A-SOUL: Advanced n-way Superscalar Out-of-order Unified Logic design

Ruochong Chen, Jirong Yang, Ruiqi Zhao, Yunpeng Liu, Jiacheng Xie
University of Michigan
{ruochong, yjrcs, ruiqizh, yunpengl, jcxie}@umich.edu

***Abstract*—The MIPS R10K microarchitecture, known for avoiding unnecessary data transfers and supporting true register renaming, gained popularity as a high-performance out-of-order processor design during the late 1990s and early 2000s. Building on R10K, we propose A-SOUL, an advanced out-of-order processor with a unified logic design at the RTL level. A-SOUL incorporates base functionalities of R10K out-of-order execution and integrates several advanced features, including n-way superscalar execution, early branch resolution, advanced branch predictors, return address stack, adaptive instruction prefetching, victim cache, non-blocking instruction and data caches, and load-store queue with store-to-load forwarding and speculative load scheduling.**

A-SOUL achieves an arithmetic average cycles-per-instruction (CPI) of 1.736 across benchmark a set of C programs, with a clock period of 7.8 ns. Moreover, we provide a comprehensive analysis of module utilization rates, stalling behavior, CPI optimization, and critical path improvements, alongside parameter tuning to maximize processor performance.

The processor is implemented in SystemVerilog. It is thoroughly simulated against a Golden In-order processor across 20+ benchmark programs (.c and .s) to ensure correctness, and then synthesized using Synopsys Build Tools. The project demonstrates the integration of sophisticated microarchitectural techniques for achieving high performance in processor design. A-SOUL is available at https://github.com/EECS-470/p4-f24.group10.

I. Overview

We built an out-of-order RISC-V processor modeled off the MIPS R10K processor with the constraints listed below:

TABLE I: Design constraints of the processor.

Field	Value
ISA	RV32I
Architectural Register	32
Total Cache Size (In Bytes)	512
Total Memory Size (In Bytes)	65536
Memory Addressability	Byte Addressable
Memory Bandwidth	1 transaction / cycle
Memory Latency (In Nanoseconds)	100
Memory Block Size (In Bytes)	64
Register File Write Bandwidth	Same as superscalar ways

Our processor achieved an average CPI of 1.78 on the C benchmark programs, and a clock period of 7.8ns in our final submission. Later, we found a bug in the speculative LSQ that causes it to spam load requests to the dcache, we were able to achieve an average CPI of 1.73 after fixing the bug.

A. Feature Highlights

Advanced features this project implements:

- N-way superscalar execution
- Early branch resolution
- Speculative load scheduling
- Early tag broadcast (partially)
- Various advanced branch predictors
- Non-blocking data cache & instruction cache
- Configurable adaptive prefetcher (on instruction cache)
- LSQ with load buffer and store queue
- Out-of-order issuing of memory access
- Byte-level forwarding from stores to loads in dcache and LSQ
- Non-blocking pipelined functional units
- Return address stack
- Victim cache (on instruction cache)

Beyond microarchitectural features, we have some great features at the engineering level:

- Fully parametrized data structures
- Automated testing infrastructure
- Targeted customized test suites
- Well-coveraged unit tests of (some) individual modules

B. High Level Architecture

A very high-level overview of our processor architecture can be seen in Figure 1. Notice that this architectural diagram represents the actual module organization of this project, rather than a logical diagram of pipeline stage divisions.

The following illustrates the life cycle of an instruction:

1) Fetch: The instruction is fetched from the instruction cache using the Program Counter (PC) and stored in the fetch stage pipeline register as a fetch packet.
2) Decode: The fetch packet is decoded in the decode stage, stored in the instruction buffer, and repackaged as a decode packet with additional decoded information.
3) Dispatch: The dispatch arbiter determines the number of decode packets to be dispatched from the instruction buffer. At this stage, the instruction updates the Reorder Buffer (ROB), Load Store Queue (LSQ), B-mask register, and Branch Stack. Then it is repackaged as a Reservation Station (RS) entry.
4) Issue: When operand register values and functional units become available, the RS issues its entries to the issue stage pipeline register, along with the register values of the register file. These form execute packets to the issue stage pipeline register.

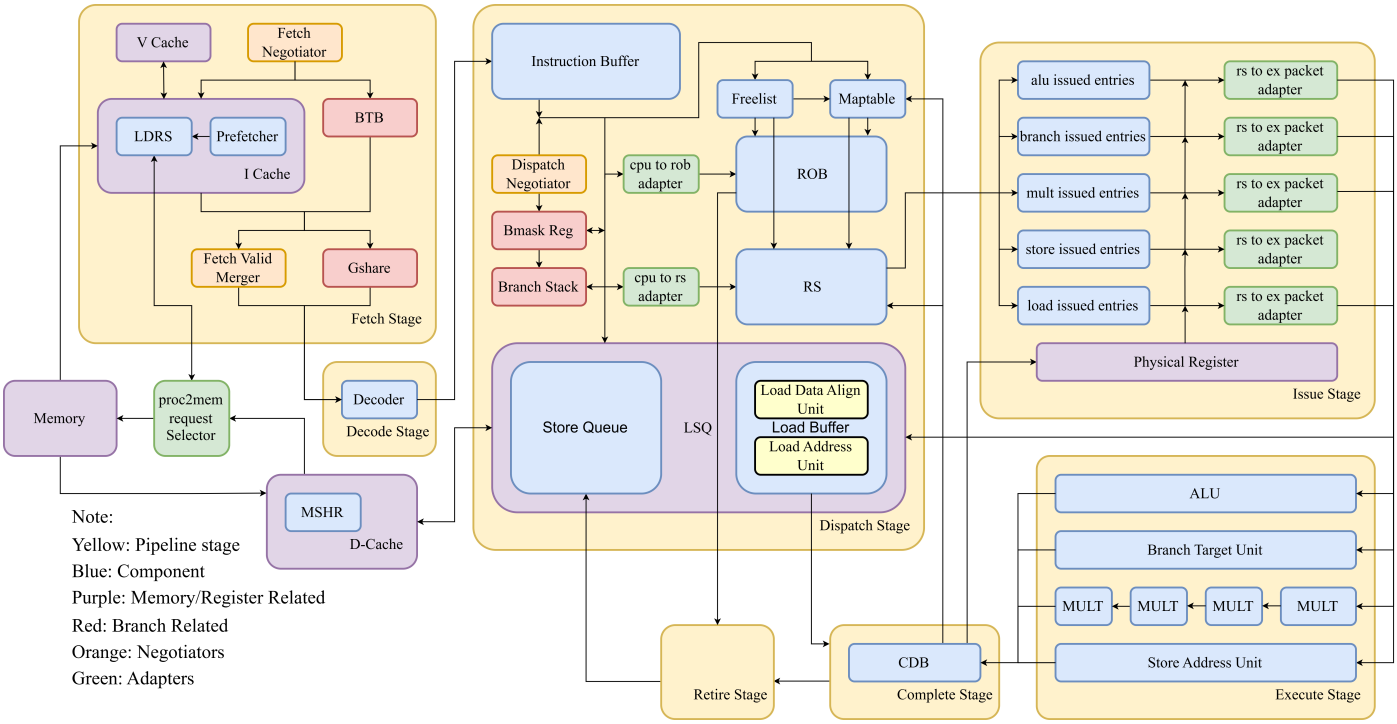


Fig. 1: Architectural Diagram

5) Execute: The instruction enters the corresponding functional unit in the execute stage, where results are calculated and stored in pipeline registers for later stages.
6) Complete: The Common Data Bus (CDB) selects results from the execute stage pipeline registers and broadcasts them to all stages. The ROB marks completed entries, converts execute packets into commit packets, and prepares them for retirement.
7) Retire: Once a ROB entry is retired, it sends its commit packets outside the CPU to confirm the completion of the instruction. For store instructions, the LSQ handles final retirement, committing the results to the data cache, and data cache will flush its dirty data to the memory at the end.

II. Design Parameter Configuration

Table II shows the configuration of our processor that yielded the best performance.

TABLE II: Parameters of the processor

Field	Value
Superscalar Ways	2
Architectural Registers Size	32
Physical Registers / Free List Size	32+24=56
I-Cache Size (in bytes)	256
I-Cache Line Size (in bytes)	8
I-Cache Associativity	Direct-Mapped
I-Cache LDRS Size	15
I-Cache Prefetch Count	19
D-Cache Size (in bytes)	256
D-Cache Line Size (in bytes)	8
D-Cache Associativity	Direct-Mapped
D-Cache MSHR Size	16 (15+1 reserved)
Instruction Buffer Size	8
ROB Size	24
RS Size	16
Store Queue Size	8
Branch Stack Size	4
RAS Size	8(not used)
ALU FU Count	4
Multiplier FU Count	2
Branch Target FU Count	1
Store Target FU Count	2
Load Target FU Count	4 (Same as LSQ Load Buffer Size)
CDB Bandwidth	2 (Same as superscalar ways)
Register File Write Bandwidth	2 (Same as superscalar ways)
Register File Read Bandwidth	26 (FU total count * 2)

III. Infrastructure

Our team prioritized building **reusable infrastructure modules** from the very beginning. While some turned out to be inflexible or redundant, discussing them provides essential context for understanding this report.

A. Array

This module is a fully parameterized, general-purpose array with configurable options:

- Depth: Size of the array.
- Data Type: Type of data stored in the array.
- Read Ports: Number of read index-data pairs handled per cycle.
- Write Ports: Number of write index-data pairs handled per cycle.
- Read Mode: Determines if write data can bypass to read

data in the same cycle.

The read mode parameter enables its over-pushing or over-popping feature, supporting the *queue_superscalar* module.

The module also includes an interface for forcefully updating the entire array, which was later deemed redundant.

Our register file is built on this module with added logic to prevent reads from PREG 0. It supports `FU_TOTAL_NUM * 2` read ports and `SUPERSCALAR_WAYS` write ports.

B. Queue_superscalar

This module is a fully parameterized superscalar queue supporting up to `SUPERSCALAR_WAYS` elements pushed or popped per cycle. Push/pop enable signals should be determined externally using push/pop space output by the module.

To maximize queue bandwidth, we implemented forwarding modes that allow over-pushing or over-popping by letting push/pop space see the pop/push count of the current cycle. This functionality depends on the bypassing mode of the underlying *array* module.

C. Parking Selector Series

These modules reuses modules from Project 1 to address the problem of mapping *x* inputs/outputs to *y* inputs/outputs.

- *psel_order_masked*: A modified version of the *psel_gen* module that performs ordered selection from requests while accounting for irregular grant masks (e.g., `101`). It selects the lowest available bits from requests and maps them to corresponding grant lines.
- *parking_selector*: A wrapper for *psel_order_masked* that extracts input-assigned masks (used request bits) and input-served masks (served grant mask bits).
- *parking_selector_input*: Extends *parking_selector* to solve the problem of mapping *x* inputs to *y* spots.
- *parking_selector_output*: Like *parking_selector_input*, but maps *x* outputs to *y* spots.

IV. Protocols

We defined various protocols to ensure consistent understanding throughout different parts, establish a standard code style, and facilitate the development of reusable components with simplified module interfaces.

A. Basic Packet Structure

An instruction transforms through fetch packet, decode packet, RS entry, execute packet, and commit packet during its life cycle. Each packet nests within the structure of the next stage, providing complete instruction information at every stage. This approach significantly simplifies debugging and module wiring.

B. Enable-Validness

All modules use an additional enable signal alongside input signals to indicate input validity. This separates concerns of data correctness from control correctness.

C. Adapter Pattern

The adapter pattern is a core design principle in our project. All modules including the Reservation Station, Instruction Buffer, Issue Stage, Execute Stage, and CDB output packets in the same format as their inputs. Packet conversions occur only at stage boundaries using adapters. This approach enables component reuse and cleanly separates data processing from control logic.

D. Memory Access Prioritization

We set memory access priorities as **data cache** $>$ **instruction cache** $>$ **instruction cache prefetcher**. Prioritizing data cache requests offers several advantages:

- It shifts the complexity of handling asynchronous memory requests to the instruction cache that simplifies data cache implementation.
- Data cache could handle smaller memory accesses than instruction cache, which operates at word granularity.
- A data cache response can serve multiple instructions depending on program memory patterns, while instruction cache responses generally serve up to two instructions.
- We note that load instruction latency is a performance bottleneck, so prioritizing regular load requests helps reduce CPI and improves overall processor performance.

E. Block-Based Memory Access

All memory transactions in our processor operate at the block level (8 bytes). Memory access sizes are fixed to a double word, aligned to the nearest block address that is calculated by setting the lowest 3 bits of the access address to 0. Each access is defined by a block address, 8-byte block data, and an 8-bit byte mask.

For loads, a 1 in the i^{th} bit of the byte mask indicates that the load accesses the i^{th} byte of the memory block. For stores, a 1 in the i^{th} bit of the byte mask signifies that the store writes to the i^{th} byte. To determine if store data can be forwarded to a load, we check: `(load_byte_mask & store_byte_mask) == load_byte_mask`. If true, the store writes at least the bytes that the load reads, allowing the load to use the stored data without accessing the next memory hierarchy level.

V. Components

This section describes all components of the processor.

A. ROB

The ROB is implemented as a superscalar queue with additional logic. It utilizes the *array* module as its underlying data structure.

On receiving signals from the Common Data Bus (CDB), the ROB scans its entries and marks those entries with matching `t` and ROB indices as completed, and creates commit packets for these entries. The ROB index ensures that instructions without destination registers are not incorrectly retired.

In each cycle, the ROB checks the first SUPERSCALAR_WAYS instructions from its head and retires completed instructions sequentially until encountering an incomplete or halt instruction. When retiring a halt instruction, the ROB enters a halted state to prevent further retirements. This protects against executing incorrect instructions if the halt instruction is not the final instruction in the assembly file.

On a branch misprediction, the ROB updates its next tail to the checkpointed ROB tail from the branch stack.

B. RS

The reservation station takes the advantage of the *parking selector series* modules. The dispatch process uses a *parking_selector_input* instance to map *x* dispatched RS entries into *y* available spots (invalid entries).

An instruction can issue only if both its t1 and t2 operands are ready (or 0) and a corresponding functional unit is available. The RS is designed to immediately issue an entry if the current cycle CDB completes t1 or t2 of that entry.

During the decode stage, each instruction is assigned a functional unit tag, which determines its functional unit readiness each cycle. Ready entries are grouped by functional unit tag and passed into *parking_selector_output* instances for each functional unit type. The output masks are merged via OR logic to form a will-issue mask that invalidates issued entries for the next cycle.

In our Conservative LSQ design, an additional constraint is added for load instructions to ensure they issue only after all dependent stores have calculated their addresses, which is achieved using a mask-based algorithm:

1) Upon dispatch, load instructions are assigned the LSQ tail index and a head-tail mask in their RS entry.
2) Each cycle, the RS updates the head-tail masks of all entries by AND them with the LSQ waiting mask, which notifies loads when their dependent stores become ready.
3) A load is ready to execute when its head-tail mask becomes 0, indicating all dependent stores are resolved.

Once the head-tail mask on the load instruction RS entry becomes 0, they are considered ready to be issued into the execute stage. This design accommodates the circular buffer structure of the Store Queue, which prevents using simple indices to detect store readiness. Updating the head-tail mask every cycle avoids conflicts when the store queue wraps around and refills the waiting mask before the load issues.

Although loads must issue in order relative to their dependent stores, they can issue out of order with respect to other instructions if their dependent stores are ready.

In our Speculative LSQ design, we eliminated this constraint, and issue loads normally on both operand register value ready and functional unit ready.

On a branch misprediction, the RS constructs a squashed bit vector to invalidate all squashed instructions, including dispatched and issued entries. On the other hand, this will still allow instructions that are not squashed to issue normally.

C. Map Table

The Map Table is a key module supporting read and update operations similar to a standard map. It updates mappings by pairing architectural registers of dispatched instructions with newly assigned physical registers from the free list. For reads, it uses architectural registers as keys to retrieve t_old, t1, t2, and their readiness for each instruction.

In a superscalar setup, managing reads is challenging because each instruction must see updates from prior dispatched instructions and must not see those after itself. To address this, we maintain SUPERSCALAR_WAYS + 1 temporary map tables. The *i*th table incorporates updates from the *i*-1th instruction, enabling parallel reads for all instructions in a cycle.

Instead of using architectural registers for the ready table, we used physical registers for the ready table to avoid associative lookups in the ready table for each instruction completed by the CDB, reducing critical path.

On a branch misprediction, the map table and ready table are reverted to the mappings from the checkpoint.

D. Freelist

The freelist is built on the *parking_selector_output* module to allocate physical registers based on its free vector, with the grant mask set by the dispatch enable signal.

On a branch misprediction, the freelist restores the checkpointed state. When ROB entries retire, the freelist updates the free vector by setting the corresponding bits for retired physical registers to 1.

E. Instruction Buffer

The instruction buffer is built upon the *queue_superscalar* module, configured with decode packets as its data type. While initially designed to use the forwarding feature, this approach connected the fetch and dispatch stages, creating a long critical path. As a result, we use the buffer without forwarding.

On a branch misprediction, the instruction buffer is reset to clear all entries.

F. Branch Target Buffer

The Branch Target Buffer (BTB) improves branch prediction by caching the target addresses of taken branches. It stores PCs of branch instructions and their corresponding target addresses. When an instruction is fetched, the BTB checks if it is a branch and provides the target address if available.

Only taken branches (i.e., those that result in a jump) are recorded in the BTB. Branches that are always not taken are excluded, allowing them to be consistently predicted as not taken without involving the branch direction predictor. This reduces unnecessary predictions and mispredictions, enhancing overall accuracy.

G. Branch Direction Prediction

We implemented various branch direction predictors (discussed later), including NT (Never Take), AT (Always Take), Bimodal, PAg, PAp, GAp, GAg, GShare, and Tournament branch prediction. We use the GShare predictor by default,

which combines global branch history (BHR) and the PC to index into the pattern history table (PHT).

A key design choice was to avoid speculative updates to the BHR and PHT. Instead of updating the BHR immediately after a prediction, updates are deferred until the branch reaches the complete stage and its actual outcome is known. This ensures the BHR and PHT reflect only the true behavior of branches.

H. I-Cache & Prefetching

The i-cache was designed to enable non-blocking instruction fetching and improve memory access speed. A key challenge was that the Instruction Fetch Unit (IFU) would stall on a single cache miss, halting instruction fetching until the miss was resolved. To address this, we implemented a prefetcher that speculatively issues memory requests for subsequent instructions.

Due to the memory prioritization protocol, the i-cache must handle pending memory requests and issue them asynchronously. To address this, we designed a mechanism called Load Reservation Station (LDRS), which inspires by the RS. This mechanism takes the advantage of the word-only and load-only memory access patterns of i-cache. On a cache miss, the memory block address of the first missed instruction is used to allocate an LDRS entry. Each entry contains a valid bit, memory block address, and an issued bit. When the d-cache is idle, an unissued LDRS entry sends a memory load request and records the transaction tag. Once a memory response with a matching tag returns, the LDRS entry is deallocated, and the data is written into the i-cache.

The LDRS also supports the design of the adaptive prefetcher, which operates using a simple state machine with two states: `IDLE` and `PREFETCHING`. While the prefetcher is in the `PREFETCHING` state, its behavior is determined by 6 scenarios: `NOP`, `ALLOCATE`, `STALL`, `RESTART`, `SKIP`, and `STOP`. To handle cases where multiple scenarios occur simultaneously, we prioritize them as follows: `RESTART > STOP > STALL > SKIP > ALLOCATE`. Each scenario is defined as follows:

- `NOP`: No operation; the prefetcher remains in its current state and does nothing.
- `RESTART`: Caused by a new i-cache miss; resets the base address and progress of the prefetcher.
- `STOP`: Occurs when the prefetcher has fetched `PREFETCH_CNT` blocks; terminates the prefetching process.
- `STALL`: Occurs when the LDRS is full, causing the prefetcher to pause.
- `SKIP`: Skips prefetching a line if it is already present in the i-cache or the LDRS.
- `ALLOCATE`: Default action; allocates an LDRS entry for the prefetched line, allowing the LDRS to manage the memory request.

This adaptive prefetcher enables prefetching of multiple lines without disrupting the normal operation of the i-cache. We observed significant performance improvements with effective prefetching. Increasing the prefetch depth from 1 line to 4 lines resulted in a 30% reduction in CPI. Extending the prefetch depth further from 4 lines to 8 lines yielded an additional 10% improvement. We ultimately settled on a prefetch depth of 19 lines for optimal performance.

I. Victim Cache

To minimize cache miss penalties and enhance instruction fetch performance, we implemented a victim cache as a small, fully-associative cache with 4 lines, serving as a buffer between the i-cache and main memory.

When a cache line is evicted from the i-cache due to a miss or conflict, it is moved to the victim cache. This gives recently evicted lines a second opportunity to be accessed before resorting to the slower main memory.

To address the limitations of the direct-mapped cache, we introduced a victim cache to handle frequent conflicts. During the optimization process, we compared three configurations: no victim cache, a victim cache that writes back to the I-Cache, and one that does not. As shown in the figure 2, we found that overall, the differences were minimal, with the non-writeback victim cache achieving slightly better CPI than the other two configurations. However, we were initially misled by this improvement. Despite observing a low victim cache hit rate, we mistakenly assumed this was inherent to its nature. It was only towards the end that we realized the victim cache should not have been added to the I-Cache. This is because the basic block size between two branches is typically around 8 instructions, while the victim cache in our design has only 4 entries. This limited capacity can lead to cascading conflicts after a branch switch, significantly reducing its effectiveness. Instead, the victim cache should assist the D-Cache, where data access patterns are more complex and random. In this context, a victim cache can significantly improve hit rates and provide a meaningful performance boost.

Implementing victim cache brought us a 0.02 CPI reduction at the time of implementation.

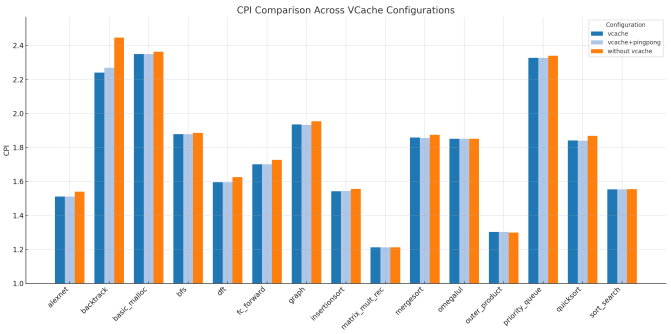


Fig. 2: C CPI for different vcache

J. D-Cache

The d-cache we implemented is a **write-back** and **write-allocate**, **non-blocking** data cache that is integrated with the LSQ. To achieve non-blocking behavior, we incorporated a Miss Status Handling Register (MSHR) into the d-cache. Each

MSHR entry tracks outstanding memory requests and contains state, block address, merged byte mask, and merged data. Each MSHR entry operates as a finite state machine (FSM) with three states:

- `INVALID`: Default state.
- `ISSUED`: Initial state after the MSHR entry is allocated and a memory request is issued. The d-cache ensures the request is always successful at this stage since it has the highest memory access priority.
- `LOADED`: State after the MSHR entry receives the memory response. The data will be written back to the d-cache in the next cycle, and the MSHR entry will be invalidated.

We imposed several constraints on the d-cache to simplify its design and implementation. For example, the d-cache accepts only 1 memory request per cycle from the processor and is built around a *memDP* module with a single read port and a single write port. Based on the constraints, we identified the following scenarios for d-cache operation:

- Invalid block address: Occurs when a memory access targets an address outside the 64KB memory space, typically due to speculative instructions. The d-cache responds immediately with a 0 to clear load entries with invalid addresses.
- Cache hit: Happens when the address index and tag match a valid cache line. The cache updates or reads the data as needed.
- Cache miss, MSHR hit: Occurs when a cache miss finds a matching block address in the MSHR.
  - For load requests: The d-cache checks if the MSHR can forward merged data. If possible, it responds with the forwarded data; otherwise, it does nothing.
  - For store requests: The MSHR updates its merged byte mask and block data.
- Cache miss, MSHR miss: Occurs when cache miss and there is no matching block address in the MSHR. A new MSHR entry is allocated, and a memory request is then issued. The d-cache uses 16 MSHR entries (entry 0 is reserved) for indexing the memory transaction tag or data tag directly.
- Write-back required: Happens when an MSHR entry transitions to the `LOADED` state. The d-cache locks to prioritize write-back operations:
  - The read port fetches dirty data from the cache for writing back to memory.
  - The write port writes merged data into the cache line, updating the dirty bit based on byte mask of the MSHR.

  If the merged byte mask is non-zero, the data remains dirty for future write-backs.
- Memory response received: This can occur alongside any other scenario. A memory response updates the corresponding MSHR entry state to `LOADED` and merges the received data into the MSHR entry.

As a write-back cache, the d-cache must handle remaining unretired stores in the LSQ, incomplete store operations in the MSHR, and dirty cache lines upon processor halt. To address this, we implemented a simple FSM in the d-cache for flushing dirty data. During processor halt, the *cpu_test* driver program sets the `start_flushing` signal high and waits for the `done_flushing` signal to indicate completion. The FSM consists of 4 states:

- `IDLE`: Default state. The d-cache remains inactive.
- `WAITING`: When `start_flushing` is set high, the d-cache waits until the MSHR is empty, ensuring all memory requests are served or issued. It then transitions to the `FLUSHING` state.
- `FLUSHING`: The d-cache iterates through all 32 cache lines, checking each for a dirty bit. For dirty lines, it issues a store memory request. After processing all lines, it transitions to the `DONE` state.
- `DONE`: The d-cache sets the `done_flushing` signal high for completion. It remains in this state until reset.

K. LSQ

The LSQ is the most complex and error-prone module in our project. To reduce its complexity, our initial design imposed two constraints during the dispatch stage: (1) at most one store can dispatch per cycle, and (2) no load can dispatch alongside a store. The first constraint allows us to use a simple store queue and avoids computing multiple LSQ entry indexes for concurrent store instructions. The second constraint avoids calculating LSQ entry indexes for concurrent loads and return different head-tail masks, simplifying load-store dependency handling.

When dispatching a store, its LSQ entry index (current tail index) is recorded in its packet and carried through subsequent stages. An entry is allocated in the store queue for the store. Once the store address functional unit calculates the store address, the result is fed into the LSQ, where the entry index is used to identify the corresponding store queue entry and update details such as the memory access size and byte mask.

Each store queue entry operates an FSM with 4 states:

- `INVALID`: Default state of the entry.
- `WAITING`: State after dispatch and entry allocation, awaiting the functional unit result.
- `READY`: State after receiving the execute packet from the store address functional unit.
- `RETIRED`: State after the store retires in the ROB but is pending retirement in the LSQ.

Each state transitions sequentially from the previous state.

When dispatching a load, its LSQ entry index and head-tail mask are recorded in its packet and carried through subsequent stages. The LSQ design relies on two key concepts:

- Head-Tail Mask: Shows validity of store queue entries, where a 1 at the ith bit indicates the ith entry is valid.
- Waiting Mask: Indicates store queue entries currently in the `WAITING` state.

Each load instruction carries two copies of the head-tail mask initialized at dispatch, which are updated differently:

- `head_tail_mask`: `AND` with waiting mask of LSQ each cycle to determine if all stores before the load have reached `READY` state. If true, the load is ready to issue.
- `head_tail_mask_forward`: `AND` with head-tail mask of LSQ each cycle to track which store entries can provide forwarding data. This prevents conflicts when retired stores are replaced by new ones in the same entry.

Using the described algorithm, we enable byte-level forwarding from store entries to loads. When a load enters the LSQ, it first checks that all preceding stores have reached at least the `READY` state. The LSQ then traverses the store queue from head to tail, merging data and byte masks for store entries with a block address matching loads. If full byte-level forwarding is possible, the load uses the forwarded data. Otherwise, it is issued to the d-cache. After receiving a response from the d-cache, the load re-traverses the store queue to update its data with any relevant stores that have not yet retired into the d-cache.

We developed three major versions of the LSQ.

1) Version 1: In the first version of our LSQ, we restricted the design to handle only one load at a time, ensuring that a load would block all subsequent loads until it was completed. This approach focused on verifying the correctness of all LSQ components. The load functional unit in this version was limited to one. The load FU is divided into 3 stages:

- Address: A simple ALU calculates the address and memory size of the load instruction.
- LSQ: The load retrieves the memory block data corresponding to the block address being accessed.
- Align: Custom logic converts the 8-byte memory block obtained in the previous stage into 4-byte data and processes it based on signedness of the data.

In this version, load operates on a FSM with 4 states:

- `INVALID`: Default state.
- `CAN_FORWARD`: Initial state when the load enters the LSQ. The LSQ traverses the store queue to find data-forwarding opportunities. If forwarding succeeds, the load transitions to the `COMPLETED` state; otherwise, it moves to the `CAN_ISSUE` state.
- `CAN_ISSUE`: State where the load keeps sending requests to the d-cache, checking for a cache hit or MSHR data forwarding. The load switchs to the `COMPLETED` state only if receiving a valid response from the d-cache.
- `COMPLETED`: State where the LSQ waits for the CDB selection signal from the complete stage. Once the load is completed, it transitions to the `INVALID` state, allowing new loads to enter the load functional unit.

The load bandwidth of this LSQ version was extremely limited, as it required at least $5 + x$ (memory access time) cycles for a load to execute and complete. During this time, no other load or dependent instructions could be issued, significantly restricting our processor performance. In this version, the processor achieved a CPI of approximately **2.5**.

2) Version 2: The second version of the LSQ focused on maximizing load bandwidth. In the previous version, the d-cache served load requests only when a cache hit or MSHR forwarding occurred, requiring the load to remain in the LSQ. We identified that the only scenario requiring a load to wait for another cache hit is when the load request is served by an MSHR and the d-cache is waiting for memory data to return. To address this, we modified the d-cache to actively send a response valid signal when an MSHR entry receives the returned memory data, reducing the need for the load to wait for a cache hit.

Then, we took an aggressive approach by removing the first and third stages of the load functional units and moving all computations within the LSQ. All computations are integrated inside a load buffer, where each entry acts as a load functional unit that holds all relevant information such as address, memory size, block address, and byte mask. This design substantially simplified the LSQ, as the RS could directly issue into the load buffer, and the CDB could complete directly from it. Each load entry remained controlled by an FSM, but with an additional state:

- `INVALID`: Default state, same as before.
- `CAN_FORWARD`: Initial state when the load is issued, same as before.
- `CAN_ISSUE`: State where the load can issue a d-cache request if no forwarding opportunity is available. A d-cache miss moves the load to the `WAITING_RESPONSE` state, while a hit transitions it directly to the `COMPLETED` state, with any necessary store forwarding from the store queue.
- `WAITING_RESPONSE`: State where the load waits for the d-cache load response signal for its block address.
- `COMPLETED`: In this state, the load outputs a `done` signal and waits for the CDB selection signal to transition back to the `INVALID` state.

Although a load can still take up to $4 + x$ (memory access time) cycles to complete, adding a load buffer enabled true non-blocking load handling and maximized load bandwidth of the processor. This modification alone reduced the CPI from 2.5 to 2.0, achieving a 25% speedup.

3) Version 3: We implemented various metrics profiling the performance of the processor from all sorts of places (discussed later). We observed several facts from the data we collected: First, a large portion of all executed instructions are loads (40%). Second, 21% of the stalls at the issue stage is caused by the loads waiting for stores, which will cause temporary congestions that fills the RS, causing stalls in dispatch and fetch stage. In addition, loads took the longest in all types of instructions except for MULT. Furthermore, we noticed that only 5% of the loads are able to forward data from stores, but we use a state to explicitly do this, which makes forwarding not worth it.

Based on the above observations, we made a very adventurous decision, where we attempted to **implement speculative load scheduling in the last 2 days of the project**.

In the conservative load scheduling LSQ, we ask the load to wait for the stores to be marked ready in the store queue. However, even if the load only depend on 1 store, the store needs to wait for register values, and then go through the issue stage and mark ready for its entry in the store queue. The load can issue at the next cycle after finishing all these work. This forced all loads to wait at least 2 cycles to be issued.

We decided to implement **speculative load scheduling**, which allows the loads to not wait for the stores at all and just issue upon register values ready. After a load is issued, it will immediately issue a dcache request, as we completely gave up on forwarding data from the store queue. This also hides the latency and let the store wait for register value, issue, and become ready in store queue, while waiting for a dcache miss. After a dcache response returns, we transition the load into the newly added stage `WAITING_STORE`. During this stage, we will check the forwarding mask on the load, if the mask is not 0, then we will walk through the store queue and merge data from stores with the same block address sequentially, until we encounter a `WAITING` store or a 0 in the forwarding mask. If the mask is 0, then we will calculate the final 32bit value of the load and transition the load into the `COMPLETED` stage, allowing the load to be completed by the CDB.

Another design choice we made is we absolutely didn't want a sorted data structure (a load queue), which will prevent us from completing loads from the load queue out of order without leaving holes, and switching to a load queue will require a large amount of refactoring in the pipeline. Issues soon emerged with this design decision, but we will discuss how we solve it later.

During our implementation, a very difficult problem we encountered is that, since speculative load scheduling doesn't care about issuing loads in store order, there can be cases where a store depends on an earlier load, but this earlier load can't issue as other later loads are already in the load buffer, which are then waiting for that store, causing effectively a deadlock. We came up with all sorts of solutions such as using a semaphore, switching to a load queue, using a load done buffer, but none of them really works due to the existence of early branch resolution. The core constraint we wanted to put is that, no more than `LOAD_NUM` (load buffer size) loads should be "inflight" between dispatch and complete stage. We ended up using the simplest solution, which is to count the number of loads in the RS, issue stage, and the load buffer, and use it as a "occupied spots" in the load buffer. This ensures that all dispatched loads can enter the load buffer and be execute successfully.

Another observation we had was that, in our issue/execute stage, we implemented the "non-blocking execute pipeline", which propagates the readiness of the next stage in the issue/execute pipeline to determine the current stage's readiness. However, this can cause an issue where the actual functional units are almost full, where 1 more input will cause it to be "not ready". The RS will be able to issue 2 instructions to the same FU before the issue stage propagates the "not ready" from the FU, where the first instruction will cause the FU to lock down, and the second instruction will get stuck in the issue stage. Normally, the benefit of doing this is that we are able to allow the RS to issue 1 more instruction per FU before it locks down, and let the issue stage temporarily hold the instruction. The lock down will usually be solved in 1 or 2 cycles as this is usually caused by the CDB not having enough bandwidth. However, load instructions are the exceptions here. The loads will stay at each stage at its entry for a variable number of cycles, which can cause situations where 2 loads are issued to the same load buffer entry position, but since the previous load can usually take a large amount of cycles before the entry is freed, the load stuck in the issue stage can wait for over 10 cycles, during which there might very well have other entries that are freed but is not issued anything, as everyone is waiting for that stuck load.

Based on the discussion above, we handle loads specially in our issue stage ready logic, which uses issue_load_valid AND load_entry_ready as the ready bits passed to the RS. This enforces that loads that enters the issue stage will be guaranteed to enter the execute stage without getting stuck in the issue stage. This change along brought down our CPI by 0.05, which is considered a huge improvement at the late stage of our project.

This last version of LSQ brought as a total of 0.15 average C CPI decrease, going from 1.85 to 1.71, which is considered a breakthrough in our project. In addition, we observed that 21% of the stalls in the issue stage are caused by loads waiting for stores before implementing speculative LSQ, and this number is reduced to 0 after implementing speculative LSQ. In addition, load instructions used to take on average 12.5 cycles from the dispatch stage to the issue stage, this is reduced to around 2 after implementing speculative load scheduling.

Figure 3 shows the CPI difference between different LSQs. The "Baseline" LSQ was our conservative LSQ, the "Improved" LSQ was our conservative LSQ with the issue stage tweak added, and the "Speculative LSQ" refers to our last version of the LSQ, which was speculative LSQ.

In all versions of the LSQ, the store queue is restricted to retiring at most one entry per cycle to simplify d-cache design. Also, initially, store accesses are prioritized over load accesses when both need to issue d-cache requests. This is because we think that issuing a store to d-cache guarantees its removal from the store queue, while a load may remain in the load buffer if a cache miss occurs. Later, we tried changing the LSQ to prioritize loads over stores. This gave us a 0.01 CPI improvement, so we switched to that design.

L. Early Branch Resolution, Branch Stack, Bmask-reg

EBR utilizes a global `br_complete_enable` and `br_complete_packet` signal, along with a branch stack and bmask register module. When a branch is dispatched, the bmask register provides a new branch tag (e.g., 0010) and branch mask (e.g., 0011), and the branch stack records the current state of components such as the Map Table, LSQ,

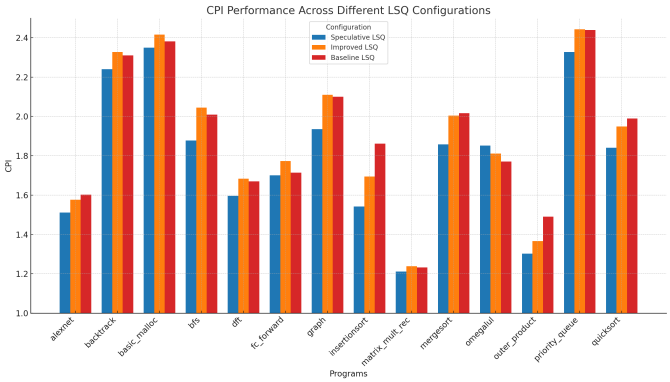


Fig. 3: C CPI for different LSQ

ROB, and Free List, including any updates caused by the branch (e.g., `jalr` instructions with destination registers).

On a branch misprediction, the branch stack identifies the checkpoint corresponding to bmask of the mispredicted branch and restores the recorded state for recovery. During branch resolution, the RS, issue stage, and functional units either update bmask of their packets (on correct predictions) or squash packets with the mispredicted bmask (on mispredictions).

One notable issue occurs when instructions that were dispatched prior to a branch instruction do not complete until after that branch instruction has been dispatched. This causes the recorded checkpoint to miss these instructions, leading to recovery with an incomplete map table. Similarly, instructions that retire after the branch is dispatched can result in recovery with an incomplete free list.

To address these issues, CDB and ROB signals are wired to the branch stack, allowing it to update checkpoints as instructions complete on the CDB or retire in the ROB.

M. Return Address Stack

To improve return prediction accuracy, our processor design incorporates a Return Address Stack (RAS). During the fetch stage, we pre-decode instructions to identify `jal` (Jump and Link) and `jalr` (Jump and Link Register). For a `jal`, the return address is pushed onto the RAS. For a `jalr`, the RAS pops its top address, replacing the address provided by the BTB.

Because the branch mask (bmask) is not generated in the fetch stage, the RAS does not integrate a full misprediction recovery mechanism. Instead, we only record the RAS tail upon detecting a branch that progresses into the decode stage. This approach ensures that the RAS can be partially restored if needed without relying on speculative information unavailable at fetch time.

However, integrating a fully correct RAS mechanism would introduce complexity in the critical path, potentially increasing the clock period. We ultimately decided not to include this module in our final design as the CPI improvement doesn't offset the clock period increased by adding the RAS.

N. Early Tag Broadcast

To speedup handling of RAW dependencies, we implemented an additional arbiter named ETB arbiter that functions similarly to the CDB arbiter but operates one cycle earlier. For instance, in a non-blocking mult functional unit with eight stages, we broadcast the tag to the ETB arbiter at the end of the 7th stage instead of broadcasting the actual data at the end of the 8th stage. When a reservation station (RS) entry receives the required tag, it issues a request. In the following cycle, the forwarding logic from the CDB to the issue pipeline register ensures that the correct packet is passed to the execute stage. The reason this approach works is that, apart from instructions squashed due to a misprediction, the instructions selected by the ETB arbiter are identical to those that will be selected by the CDB arbiter, with only a one-cycle difference. If the ETB arbiter selects an instruction that is later squashed, any dependent instructions will also be squashed in the subsequent issue stage, ensuring no disruption to the pipeline.

However, due to the trade-off between CPI and clock period, we decided not to include the ETB in our final implementation. While it only slightly reduces the CPI, it significantly increases the clock period. Additionally, there were minor synchronization errors in the ETB implementation that we did not have sufficient time to debug.

O. Fetch Stage

The fetch stage is bounded by the capacity of the instruction buffer and icache hit rate. Since the decode stage sits between fetch and dispatch, fetch uses the next push space of the instruction buffer as its fetch enable signal. With this signal, the fetch stage accesses the BTB and i-cache in parallel. BTB identifies whether the fetched instruction is a branch; if so, it provides the branch target for i-cache to retrieve the fetched packet. The fetch stage will stall if it fetches a branch (BTB hit) or if there is an i-cache miss. To simplify the critical path, we limit branch predictions to one per cycle.

On a branch misprediction, the `next_PC` of the fetch stage is reset to the correct branch target.

P. Decode Stage

The decode stage uses a modified decoder from Project 3 to convert fetch packets into decode packets and send them to the instruction buffer.

On a branch misprediction, the ROB simply updates its next tail pointer to the ROB tail stored in the branch stack checkpoint.

Q. Dispatch Stage

The dispatch stage determines how many instructions to dispatch each cycle, which involves several constraints:

- Instruction buffer has enough pop space.
- ROB has enough push space.
- RS has enough dispatch space.
- Branch stack (Bmask-reg) is not full.
- Store Queue has enough dispatch space.
- Free List has enough dispatch space.

- No misprediction has occurred.
- Branch must be the first instruction and the only branch per dispatch.
- Store instructions must not be dispatched with loads.

On a branch misprediction, dispatch is stalled as the current instructions are on the incorrect path.

We limit the dispatch stage to at most one branch instruction per cycle to avoid the complex calculation of multiple checkpoints and pushing multiple branches into the branch stack.

We also limit the dispatch stage to avoid dispatching both stores and loads in the same cycle to simplify the calculation of head-tail masks (discussed in LSQ section) for each load.

In later stages of our project, we gradually eliminated the restriction on load/store at the dispatch stage. However, we did not observe any significant performance gain from doing so, and this might be caused by bottlenecks elsewhere.

After implementing speculative load scheduling, we added another constraint which prevents the dispatch stage from dispatching more load instructions when the number of currently inflight load instructions is equal to the number of load buffer entries.

R. Issue Stage

The issue stage is responsible for issuing instructions from the RS to the Functional Units (FUs). Serving as the bridge between instruction dispatch and execution, this stage ensures instructions are issued efficiently while meeting pipeline constraints. The main constraints include:

- The functional unit required by the instruction must be idle or non-blocking.
- Multiple instructions in the same cycle must not compete for the same functional unit.
- The operands of the instructions must be ready, either from the physical register file or forwarding paths.

The issue stage utilizes the `psel` priority selector to maximize instruction issue based on functional unit and operand availability. When RS constraints are met, entries are packed into packets containing `FU_TOTAL_NUMBER` entries and their corresponding valid bits, then sent to the issue stage.

The design of the issue stage involves a trade-off between CPI and clock period. A simple approach passes packets and valid bits directly to the execute stage in a single cycle, enabling one-step instruction transfer from RS to FU, but this significantly increases the clock period.

We adopted an alternative approach by introducing a register buffer between the issue and execute stages to split the cycle. However, this introduces a new challenge: data entering the issue stage from the RS in the current cycle may encounter structural hazards in the CDB (e.g., due to FU stalls from delayed CDB selection), preventing packets in the buffer from being sent to the next stage. To handle this, the complete stage sends CDB selection signal to the execute stage and it's propagated back to the issue stage with FU availability of the next cycle. If a structural hazard occurs, the data is held within the buffer, and a conflict signal is sent back to the RS. This mechanism ensures the RS does not overwrite the same issue entry in the next cycle. Furthermore, entries in the issue stage must update branch and LSQ masks each cycle on correct predictions, and corresponding entries must be discarded when a misprediction occurs. Such optimized instruction handling minimizes stalls and enhances throughput, making this stage critical to processor performance.

S. Execute Stage

The execute stage hosts all functional units. The RS issues instructions to the execute stage based on the type of functional unit required. The complete stage then utilizes the *parking_selector_output* module to select `SUPERSCALAR_WAYS` of completed functional units for retirement.

The *parking selector series* modules prioritize selecting the lowest active bits, enabling prioritization of functional units. In the execute stage, functional units are arranged in the order: `ALU < STORE < MULT < LOAD < BRANCH`. This arrangement ensures higher priority for instructions that are more cycle-intensive or prone to structural/data hazards. For example, load instructions can take up to 13 cycles for a cache miss, while multiplication instructions require 8 cycles in the 8-stage multiplier.

Branch instructions are given the highest priority to ensure quick resolution, minimizing the significant cost of branch mispredictions even with early branch resolution. To avoid adding complexity to the pipeline, we limit the number of branch functional units to one, preventing simultaneous branch resolutions. Also, the lowest-priority functional unit occupies a unique position where, if selected by the CDB, it is guaranteed to appear in the first line of the CDB, which simplifies the implementation of selecting the branch completion signal.

We unified the interface for all functional units with the following protocol:

- *start*: Command to begin computation using the current input execute packet.
- *input execute packet*: Input packet from the issue stage.
- *done*: Signal to the complete stage indicating result availability.
- *output execute packet*: Output packet to the complete stage.
- *next_ready*: Signal indicating the functional unit has been selected by the CDB or that the result can move to the next stage.
- *ready*: Signal to the issue stage indicating availability for new inputs.

To maximize pipelined bandwidth, we introduced **Non-blocking Functional Units**. This mechanism ensures a stall in the last stage does not block earlier stages until the pipeline is full. It adds a parameter, `STAGES`, and specifies that each stage moves results forward if **the next stage is empty or if its output can progress further**. Each functional unit combines this sequential non-blocking framework for data flow control with a custom combinational module for calculating result from input execute packets. This design reflects our principle of separating control and data concerns.

T. Complete Stage & CDB

The main logic in this module revolves around the *parking_selector_output*, which selects x outputs to fit into y available spots. Here, x represents the number of completed functional units, with a width defined by `FU_NUM_TOTAL`. This logic determines which outputs are sent to the Common Data Bus (CDB).

To analyze branch prediction accuracy, we integrated a *branch_correctness_buffer* into the module. This buffer generates statistical files for executed programs, which are later visualized as graphs. Moreover, the module includes a recovery mechanism to efficiently handle branch mispredictions, ensuring accurate recovery and maintaining system performance.

VI. Testing

At the early stages of this project, we wrote lots of testbenches focusing on unit testing each of the modules, mostly in the out-of-order core such as Map Table, RS, ROB, Free List, Instruction Buffer, Branch Stack, Bmask register, etc. They were made with good test case coverage. We made a template for benchmarks to standardize the testing of modules as well.

After finishing the out-of-order core, we started integrating the whole processor and we needed a reliable way of testing the correctness of our processor. Our first step was modifying the Makefile so that it can inject a macro `INORDER` to control whether to use the reference in-order processor or our processor. We then built on this macro to develop the `make <program>.out.diff` build target. It runs the processor under both in-order and out-of-order modes, compares their outputs using diff, and reports any differences in register/memory write-back values as well as CPI. Building on that, we also developed the `make check_all` build target, which runs all benchmarks in both modes and outputs any observed differences. This has long been our way of checking whether the processor works completely correctly.

For memory-related modules such as the i-cache and the d-cache, we wrote the testbenches in a way that uses a fake memory along with the actual memory module, we connect the actual memory module to the caches, and capture the outputs of the caches to update the fake memory based on the semantics of these signals, and we check the output of the caches against the content from the fake memory to see if they match. This approach is very effective as it helps us treat the module as a black box without worrying how it works internally, as their internal logic are extremely complicated. We only needed to verify the correctness of their semantics to ensure that users could obtain correct results from these modules.

After integrating our processor, we noticed several issues. First, although our testbenches are made with good coverage, **it often turned out that our understanding of the modules was wrong to begin with**, making the testbenches incorrect as well, and thus effectively useless. We re-evaluated the importance of testbenches and decided to discontinue writing unit tests for individual modules, as they are time-consuming and don't guarantee correctness of the whole processor. Moving on, we only tested our modules directly with end to end tests on the benchmark programs or custom assembly programs written to test the features.

Second, it's extremely difficult to debug the project by just looking at the code. We ensured that each non-trivial module included a print function to output its internal state, and we can precisely control which module's output to print. Doing this significantly improved our efficiency in debugging the processor.

It's indeed painful to debug the project by just reading the log file generated by the print functions of each module. For more complicated test programs, the log file can grow to hundreds of megabytes which can make the IDE extremely laggy and frequently crashes the IDE due to out of memory issues. To deal with this, we frequently make "simplified" versions of the test programs by reducing the parameters or commenting out code. This will usually greatly reduce the number of cycles taken to run the test programs while keeping the code that exposes the bugs in our processor.

VII. Analysis

A. Analysis Infrastructure

For easier performance analysis of our processor, we built a series of tools on top of the testing infrastructure to better identify bottlenecks.

1) Automatic Experiment Pipeline: We modified the Makefile so that it enables overriding **any macro** defined in the Verilog code by simply adding `X_<PARAM>=<VALUE>`(or `X_S_<PARAM>=<VALUE>` for string values) to the make commands. This allows for easy parameter tuning and experimentation. In addition, we wrote a simple Python pipeline that works with this Makefile modification and allows us to run multiple experiments with different parameter permutations inside a JSON file. The results of each experiment, including register and memory writeback values, CPIs, and various performance counter results, are automatically saved in an experiment folder.

2) Performance Counter: Performance counters are a major part of our testing infrastructure. We mainly track 3 types of metrics in our performance counters:

- **Occupancy**: The average occupancy of each component in the processor such as LSQ, ROB, RS, instruction queue, dcache, i-cache, etc. This can also track metrics beyond the occupancy of container modules. For example, the same counter module can be re-used to track things like cache hit rate, complete bandwidth, branch prediction correct rate, etc.
- **Stage stall causes**: For all stages that might be stalled by multiple factors (fetch, dispatch, issue, complete, retire), we wrote a stall by counter to track which factor has caused a stall in each stage in the current cycle.
- **Instruction lifecycle**: This mainly tracks when an instruction has entered each stage. This allows us to determine how long each instruction spends in each part of its lifecycle.

Each of the metrics will be printed out into their own CSV files for each benchmark and this is done through SystemVerilog's `$fdisplay` function.

The CSV files will then be downloaded to a local computer (the CAEN environment does not allow the installation of Python packages), and our automatic experiment analysis framework will analyze each experiment and produce figures on each metric per benchmark.

Disclaimer: At the time this section was written, our processor was still under active optimization. Performance counter figures, along with some straightforward parameter ablation experiments, were generated using an earlier version of the processor. For key configurations, we performed extensive ablation experiments based on the latest version.

Table III shows the configuration of our processor that yielded the results of the experiment and the analysis.

TABLE III: Parameters of the analyzed processor

Field	**Value**
Superscalar Ways	4→2 (final version)
Architectural Registers Size	32
Physical Registers / Free List Size	64
I-Cache Size (in bytes)	256
I-Cache Line Size (in bytes)	8
I-Cache Associativity	Direct-Mapped
I-Cache LDRS Size	15
I-Cache Prefetch Count	16
D-Cache Size (in bytes)	256
D-Cache Associativity	Direct-Mapped
D-Cache MSHR Size	16 (15+1 reserved)
Instruction Buffer Size	8
ROB Size	32→24 (final version)
RS Size	32→16 (final version)
Store Queue Size	8
Branch Stack Size	4
RAS Size	32
ALU FU Count	4
Multiplier FU Count	2
Branch Target FU Count	1
Store Target FU Count	2
Load Target FU Count	4 (Same as LSQ Load Buffer Size)
CDB Bandwidth	2 (Same as superscalar ways)
Register File Write Bandwidth	2 (Same as superscalar ways)
Register File Read Bandwidth	26 (FU total count * 2)
Other Notes	Without→adopted: RAS, Victim Cache, and Speculative LSQ

Note: Configurations with "→" indicate changes between earlier and final versions of the processor.

B. CPI

Figure 4 shows the CPI per benchmark for both the C and assembly benchmark programs.

The average CPI for the assembly (except for halt) benchmarks is around 2.09, slighter higher than that of the C benchmarks. This is primarily because assembly programs have a much lower instruction count compared to C programs. A lower instruction count amplifies the pipeline overhead, resulting in a higher CPI. For instance, halt.s consists of only one instruction, yet takes 19 cycles to execute due to memory latency and pipeline overhead. Besides, btest1.s and btest2.s have exceptionally high CPIs as both of them involve many branches that aren't visited again. This will cause the branch predictors to become almost useless. In addition, both have very short basic blocks, which means that very few instructions are executed between branches, limiting the bandwidth of the processor.

In contrast, the C benchmark programs have an average CPI of 1.7798. C programs are long enough for the processor to warm up enough and can reflect the performance of the processor more faithfully. Even the shortest benchmark program has thousands of instructions to execute.

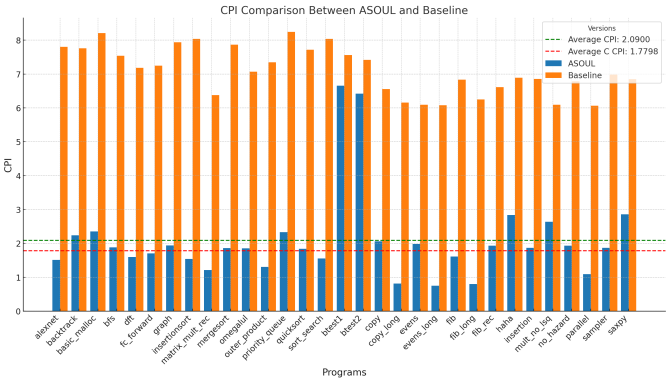


Fig. 4: C and Assembly Benchmark CPI

C. Occupancy Counters

The occupancy performance counters, illustrated in Figure 5, focus on the C benchmarks as they best represent the processor's real-world performance. Several key observations can be made from this figure.

First, the *rs_will_issue* counters represent the average number of RS entries issued per cycle, including entries for each type of functional unit and overall. These values are generally below 0.1, meaning that only 2 to 3 of the 16 RS entries are issued per cycle on average, despite the issue bandwidth matching the number of functional units. While this design aimed to maximize issue bandwidth, the data suggests that limiting it could be more beneficial by reducing the critical path caused by RS priority selectors. When analyzed alongside the *rs_ready_to_issue* counters, the data further confirms that RS utilization rates are generally below 0.1.

Another metric of interest is the *rs_not_issued* counters, which represent the average proportion of RS entries ready to issue but unable to issue in each cycle. In other words, they indicate how often the issue stage is constrained by the availability of functional units. As expected, this value is negligible across all functional units, suggesting that the functional units are rarely a bottleneck. This presents an optimization opportunity to reduce the number of functional units and shorten the clock period.

Additionally, the *rs_size* counter shows that the RS is used only about 30% on average, suggesting an opportunity to further reduce its depth to shorten the clock period. After analyzing RS utilization, we've already reduced its size from 32 to 16 with almost no CPI penalty. However, reducing it

further to 8 resulted in an unacceptably high CPI penalty of 0.04, making it impractical.

Regarding the *rob_size* counter, the average ROB utilization is around 60%, suggesting an opportunity to reduce its size. In subsequent optimizations, we reduced the ROB size from 32 to 24, which significantly shortened the critical path with almost no CPI penalty. However, further reducing it to 16 resulted in a CPI penalty of approximately 0.05, outweighing the benefits of a shorter clock period.

The *rob_retire* counter measures the average number of instructions retired per cycle. Our processor utilized only about 34% of the retire bandwidth, which is relatively low. However, this is expected, as instructions must retire in order. A single incomplete instruction at the head of the ROB can block all subsequent retirements.

The *lsq_store_queue* and *lsq_load_buffer* counters indicate the utilization rates of the store queue and load buffer within the LSQ, representing the number of valid store and load entries. The utilization rate of the store queue is relatively low. Consequently, we optimized the store queue size, reducing it from 32 to 16 and eventually to 8, as further reduction to 4 incurred a noticeable CPI penalty. Similarly, we reduced the load buffer size from 8 to 4, which eliminated a critical path caused by the excessive number of load functional units. However, reducing it further to 2 entries introduced a CPI penalty, making 4 entries the optimal size. We found that the utilization rate of the load buffer to still be around 40% after this. After implementing speculating load scheduling, the utilization rate has increased to around 60%, which is pretty good.

We used to have a *lsq_load_will_forward* counter that indicates the percentage of loads in the `CAN_FORWARD` state that can directly receive data forwarded from stores. This took 1 cycle in the conservative load scheduling LSQ. As expected, this percentage turned out to be very low, with only about 4.9% of loads benefiting from forwarding. This later motivated us to completely give up LSQ store to load forwarding when implementing the speculative load scheduling.

The *lsq_load_waiting_store*(added in speculative LSQ), *lsq_load_waiting_response*, *lsq_load_can_forward*(deleted in speculative LSQ), *lsq_load_completed*, *lsq_load_can_issue*, counters reflect the distribution of loads in various states within the load buffer. The *WAITING_STORE* was the most frequently seen state for loads in the speculative LSQ(50%), which makes sense as most loads still need to wait for all previous store addresses to be known in the load buffer. This latency is only shifted from the issue stage to the execute stage. 35% of the loads are in the *CAN_ISSUE* stage, which is due to the 1 request per cycle limitation on the dcache.

An interesting observation is that almost no loads are in the *WAITING_RESPONSE* state, which happens when the load had a cache miss in the dcache. This number used to be 20% in the conservative load scheduling design. Theoretically, this decrease is caused by a decrease in dcache hit rate, but the dcache hit rate turned out to be lower as well. We were confused by this data, and a later inspection of our code after submitting the project found this to be a bug in our speculative LSQ code that caused the load buffer entries to not transition into the WAITING_RESPONSE state correctly, causing the LSQ to spam load requests to the dcache, which explains the low dcache hit rate. Since we implemented the working speculative LSQ with only 2 days, having small bugs like this was almost inevitable, but thanks to the data we collected, we were able to identify this bug.

The *icache_prefetch_will_stop*, *icache_prefetch_will_stall*, *icache_prefetch_will_allocate*, *icache_prefetch_will_restart*, *icache_prefetch_will_skip*, counters show the percentage of each scenario occurring during the i-cache prefetching process. The `WILL_ALLOCATE` case dominates, accounting for 83% of prefetching cycles, as conflicts with the prefetcher are rare. `WILL_STALL` is nearly negligible (0.8%), indicating sufficient LDRS entries in the i-cache. `WILL_STOP` occurs in approximately 4% of cycles, which is reasonable since it represents the normal termination of the prefetcher. For a prefetch count of 19 lines, we would typically expect around 5% `WILL_STOP`. However, `WILL_RESTART` accounts for 1.4% of cases, indicating over-prefetching that leads to exiting the current basic block. This is a rare event, occurring only 1.4% of the time.

The *icache_prefetch_will_issue_mem* counter measures the percentage of cycles where the i-cache issues memory requests without being blocked by d-cache. The percentage is around 97%, indicating that i-cache requests are rarely blocked by d-cache requests.

The *icache_prefetch_is_prefetching* counter shows how often the i-cache is prefetching. We expected this to be high, but it is only 23%, likely due to a high i-cache hit rate reducing the need for prefetching.

The *icache_ldrs* counter tracks the LDRS utilization rate i-cache, shared between normal cache misses and the prefetcher. The relatively high utilization rate of around 60% is expected, as the prefetcher runs in the background to fill the LDRS even when there is a cache hit.

The i-cache has a hit rate of around 90%, heavily influenced by the superscalar width. Theoretically, wider superscalar ways will cause more icache misses as it fetches much faster than the icache can prefetch. In our earilier design, a 4-way superscalar configuration achieved a significantly lower icache hit rate (70%).

The *ib_queue_size* counter tracks the instruction queue utilization, which is relatively high at around 75%. Based on this metric, we reduced the queue size from 32 to 16, and then to 8. However, further reducing it to 4 resulted in a CPI penalty.

The *execute_fu_last_ready* counters track how quickly instructions are completed by the CDB for each functional unit. A higher value indicates a larger proportion of that unit's operations are being completed. The branch target functional unit scored a perfect 1, as expected, since branch completion is always given the highest priority. Mult instructions, with the third highest priority, scored 64%, matching their priority. Similarly, store instructions, which follow mult instructions

in priority, performed well. In contrast, loads and ALU instructions scored below 30%, despite loads having the second highest priority. This is primarily because these instructions have more functional units, but in practice, only 1 or 2 load or ALU instructions typically need to be completed at a time.

The *execute_fu_dones* counters measure how many instructions complete execution in the execute stage per cycle. When analyzed alongside the *execute_fu_last_ready* counters, we observe a similar score distribution across functional units. This indicates that the completion bandwidth of our baseline processor is primarily limited by the execution stage's capacity rather than the CDB's bandwidth.

The *dispatch_cnt* counter reveals that the dispatch stage has relatively low bandwidth (35%), similar to the retire stage. This is expected, as the dispatch stage faces the most restrictions. We will discuss these limitations in detail later.

The *dcache_request_valid* counter shows how many dcache load or store requests are responded with a valid request. In our LSQ design, a request valid on a store allows it to retire from the LSQ, and a request valid on a load allows it to move out of the CAN_ISSUE state to no longer need to send another request to the cache. This score is extremely high (96%), demonstrating the strong non-blocking nature of our dcache. The *dcache_request_valid_load* and *dcache_request_valid_store* counters show the request valid percentage of loads and stores separately. Both of them are pretty high (96%).

The *dcache_mshr_forward* counter shows how many of the "cache miss mshr hit" load requests that retrieve data directly from the MSHR. This used to be relatively high at 34% for our conservative LSQ design, highlighting the benefits of implementing byte-level forwarding in the MSHR. However, after implementing speculative LSQ, this has reduced to around 9%, which is expected as many dependent stores were able to be retired before the load that depend on them in conservative load scheduling, but loads can usually issue earlier than its dependent stores in speculative LSQ, resulting in a low forwarding rate.

The *dcache_mshr* counter shows the utilization rate of the MSHR in the cache. Its low score, averaging 1 valid MSHR entry, was somewhat unexpected. However, this needs to be analyzed alongside the *dcache_miss_mshr_miss* counter, which also scored a 6%, meaning only 6% of the time the MSHR will allocate a new entry. Despite this, reducing the number of MSHRs is not feasible, as our design depends on the 15+1 memory tags.

The *dcache_hit* shows a relatively low dcache hit rate of 55%. This used to be very high 85% in the conservative LSQ design, due to stores loading the memory block earilier.

The *dcache_miss_mshr_hit* scored a 40%, meaning 40% of the requests will hit on an MSHR entry. This used to be very low(4%) in the conservative LSQ design. Our hypothesis is that a large portion of the requests that would've caused a dcache hit have become a hit on the MSHR, which explains the high request valid rate along with low cache hit rate.

The final counter to examine is *bp_correct*, which measures the branch prediction accuracy. This scored a 76%. The baseline processor uses a Gshare branch predictor by default. On benchmarks such as alexnet, matrix_mult_rec, backtrack, sort series, and graph, the predictor achieves an average accuracy above 85%. However, this is offset by poor performance on fc_forward (37%), basic_malloc (57%), and priority_queue (76%). The poor accuracy on fc_forward is due to its two non-nested for loops with only four iterations, providing insufficient time for the predictor to warm up. basic_malloc calls the tj_malloc function just four times, and without an RAS, the predictor likely fails to predict return addresses. In priority_queue, the loop breaks after varying numbers of iterations, introducing inconsistency that the predictor cannot easily learn.

D. Stalled by counters

We are interested in what exactly stalled each stage, so we graphed the cause of stalls in all the stages in the processor. We count the conditions that limit the bandwidth of each stage the most as the reason for stall at each cycle. For example, if this cycle only 1 instruction is allowed to dispatch, the conditions that caused `SUPERSCALAR_WAYS - 1` instructions to stall will be considered as the cause of the stall and +1 in our metric.

Figure 6a shows the breakdown of causes of stalls at the fetch stage. We can see that 2 major causes of stalls at the fetch stage are i-cache and instruction queue. It is expected to have most of the cycles stalled by i-cache misses at the fetch stage as a single i-cache miss takes many cycles to resolve, and the fetch stage cannot proceed until the current instruction is fetched.

Figure 6b shows the breakdown of causes of stalls at the dispatch stage. The dispatch stage is where we have the most restrictions. From the figure, we can observe that the stalls at the dispatch stage mainly come from the RS, the ROB, the IB, and the Bmask-reg. This result seems very odd to us. It does makes sense that a good portion of stalls come from the Bmask register being full, which can be a result of branch instructions being very close to each others, which seems to be the case for C programs as a lot of them have very close `jalr` instructions.

Stalls from the instruction queue (ib) mostly come from the fetch stage not having enough bandwidth. Given that 1 icache miss can cause the fetch stage to stall for many cycles, it does make sense that this will also starve the dispatch stage with an empty instruction queue.

Stalls from the RS largely come from instructions being stuck in the issue stage, which we will explain next.

Stalls from the ROB largely come from instructions being stuck in the retire stage, which we will also explain next.

Figure 7a shows the breakdown of causes of stalls at the issue stage. And the issue stage is mostly stalled by ready_t1/t2, which comes from data-dependent instructions, and that also explains the large portion of stalls caused by the RS in the dispatch stage. Another major cause of stall in the

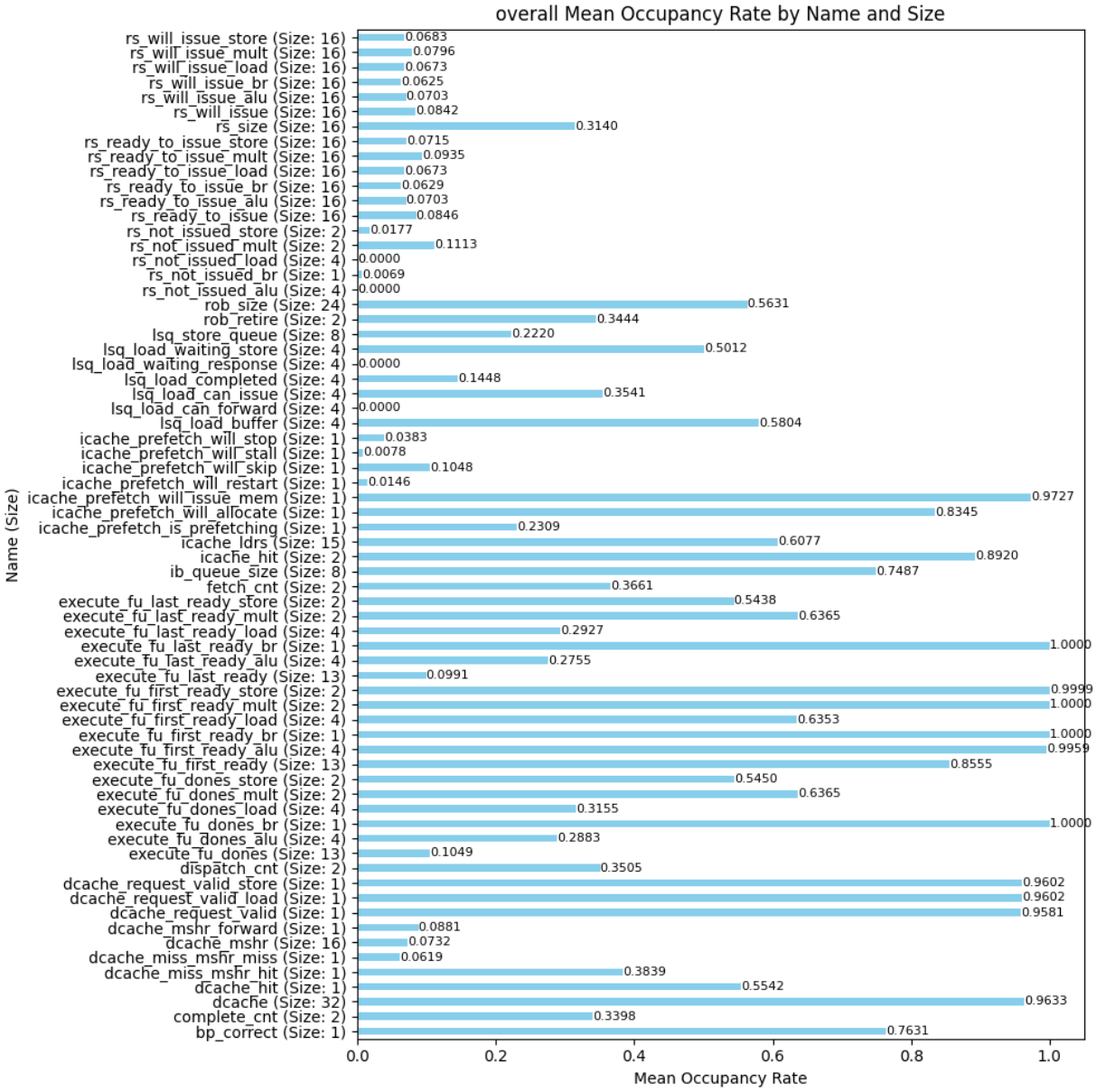
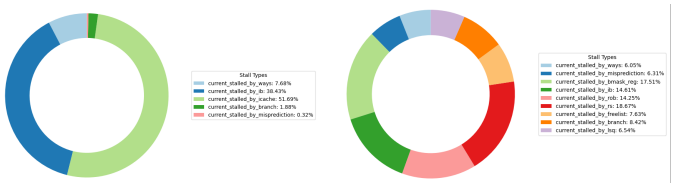
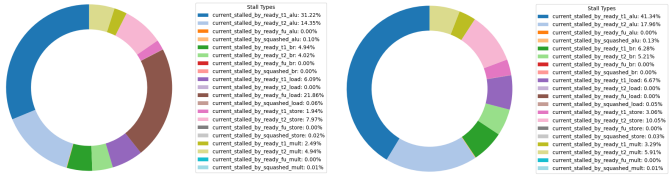


Fig. 5: Occupancy and Hit Counters (Only C benchmarks are considered)



(a) Fetch Stall Causes (b) Dispatch Stall Causes

Fig. 6: Fetch And Dispatch Stall Causes



(a) Issue Stall Causes (Conservative LSQ) (b) Issue Stall Causes (Speculative LSQ)

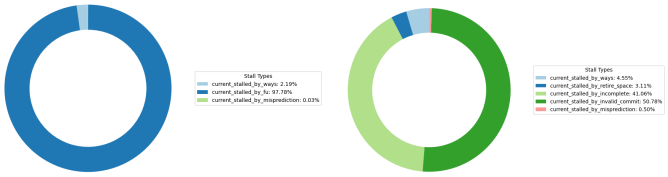
Fig. 7: Complete and Retire Stall Causes

issue stage is the readiness of the functional units. However, as discussed before, the occupancy rate of the execute stage is actually not very high, which means that most of the functional units should be ready for most of the time. The only reason for a high percentage of stalls coming from fu readiness is that

we calculate the ready_fu signal for load instructions by not only the readiness of functional units but also whether all the stores before the load have calculated their addresses. A very frequent situation we observed while debugging our project was that many of the C benchmarks have very dependent interleaving loads and stores which fill up the RS frequently. It is also shown in the figure that the 2 major reasons of stalls are ALU with unready register values (44%) and loads waiting for stores (21%).

After implementing speculative LSQ, we completely eliminated stalls caused by loads waiting for stores, and this number has reduced from 21% to 0.

Each load/store can usually be issued after the one load/store they depend on is completed. This has indeed caused a lot of stalls in the pipeline. A store takes at least 2 cycles from issue to complete, and a load takes 2 to 12 cycles, which can potentially block everyone after them. If we had more time, we would've implemented early tag broadcast(for speeding up dependent instructions) and speculative load issuing(for reducing stalls caused by dependent load-stores) and see how much performance we could gain from them.



(a) Complete Stall Causes (b) Retire Stall Causes

Fig. 8: Complete and Retire Stall Causes

Figure 8a shows the source of stalls at the complete stage. We can see that almost all the stalls come from the execute stage not having enough bandwidth and the number of functional units that are done is less than the WAYS. However, based on our experiments, reducing the number of functional units for `LOAD`, `STORE`, and `MULT` instructions will all incur a CPI penalty, so what's likely here is that the issue stage doesn't have enough bandwidth in issuing instructions to the execute stage, which was also proven before, where the will_issue counter only has a utilization rate of around 10%, 1-2 instructions issued per cycle. Another interesting observation is that, we originally wanted to optimize for the CDB bandwidth such that squashed instructions will not be considered for CDB psel for broadcasting to ensure maximum CDB bandwidth on mispredictions. This turned out to form a circular circuit and we changed that to squashing the already selected instructions, which can potentially hurt the CDB's bandwidth on mispredictions, but it turned out to have negligible impact on our CDB's bandwidth.

Figure 8b identifies the sources of stalls at the retire stage, which are influenced by several factors, including incomplete instructions, insufficient instructions in the ROB, branch mispredictions, and the unavailability of a commit packet for the retiring instruction. A significant source of stalls is attributed to incomplete instructions and invalid commit packets.

E. Instruction Lifecycle Analysis

Instruction lifecycle analysis tracks various details of retired instructions.

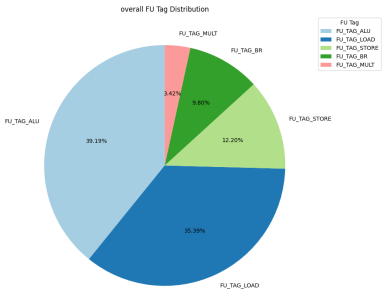


Fig. 9: Instruction By Functional Unit Type

Figure 9 illustrates the percentage of retired instructions executed by each type of functional unit. The data indicates that the ALU and LOAD instructions dominate, each comprising roughly 40% of all retired instructions. STORE instructions follow, with BRANCH instructions accounting for slightly less. MULT instructions represent the smallest proportion. This distribution aligns with the final configuration of functional units: 4 ALU, 4 LOAD, 2 STORE, 1 BRANCH, and 2 MULT.

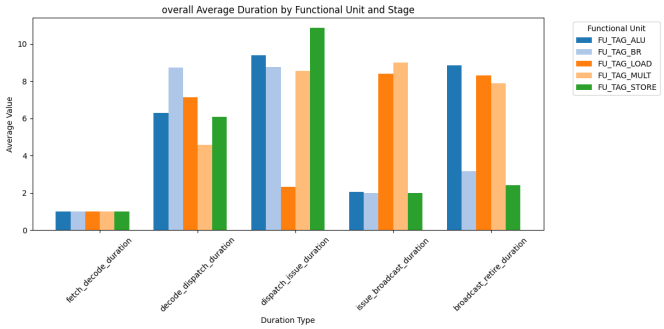


Fig. 10: Average Duration of each stage

Figure 10 indicates the average number of cycles each instruction spends in each stage. The 1 cycle latency from fetch to decode is intuitive, as there are no stalls at the decode stage. However, the delay from decode to dispatch is substantial for branches due to the restriction of dispatching only 1 branch as the first instruction per cycle.

The delay from dispatch to issue is considerable across all instruction types, averaging 8-10 cycles. Initially, we attributed this bottleneck to dependencies in load/store instructions. It's worth noting that switching from conservative LSQ to speculative LSQ reduced this latency by an average of 2 cycles, and reduced the latency of loads by over 10 cycles!

The graph also shows that MULT instructions take an average of 9 cycles, consistent with expectations since we use an 8-stage multiplier. Similarly, load instructions average 7 cycles, which is reasonable given that a cache-hit load takes 3 cycles, while a cache-miss load can take anywhere from 3 to 15 cycles.

Stalls at the retire stage are around 8 cycles, though they likely have less impact on performance compared to the dispatch-to-issue latency. Our earlier analysis indicated that most retire-stage stalls result from incomplete instructions.

Figure 11 illustrates the time each stage occupies in an instruction's lifecycle, classified by instruction type. As previously mentioned, the dispatch-to-issue and complete-to-retire latencies are the primary contributors to overall instruction delays.

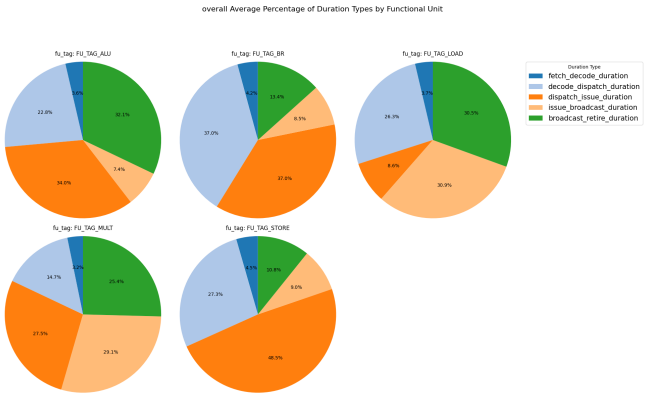


Fig. 11: Duration percentages of each instruction

F. Branch Predictor Comparisons

In the analysis of branch predictors, we implemented and compared several direction predictors, including Bimodal, GShare, GAg, GAp, PAg, PAp, and Tournament predictors. The core idea of the Tournament predictor is to use a fast warm-up GShare predictor for early prediction results, while a more accurate PAg predictor provides predictions for longer programs. When the later-stage predictor shows higher accuracy than the early-stage predictor over the most recent n predictions, we switch to the later-stage predictor for all subsequent predictions. The detailed analysis and data results of each branch predictor are as follows:

1) PAg, PAp, GAp, GAg and GShare: We compared the prediction accuracy of Bimodal, PAg, PAp, GAp, GAg, GShare, and Tournament predictors on all test programs, all using 2-bit saturating counters. The results in Fig 12 shows that the Tournament predictor achieved the best performance, slightly outperforming GShare and surpassing the other branch predictors. This demonstrates that the Tournament predictor combined the strengths of GShare and PAg, surpassing their individual performances.

However, the Tournament direction predictor is relatively large and more complex, increasing the critical path length in the fetch stage, which impacts the program's clock period. Based on the trade-off between prediction efficiency and hardware overhead, we ultimately chose the GShare predictor.

2) Speculative Update vs. Non-Speculative Update: In branch prediction, speculative update refers to immediately updating the predictor structure when fetching a branch instruction, assuming that the current prediction is correct. This

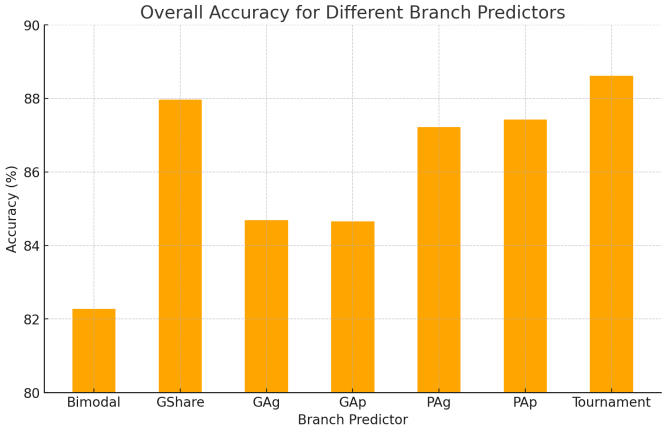


Fig. 12: Overall bp Accuracy Rate

approach allows the predictor to quickly warm up. In our design, we performed speculative updates only on the Branch History Register (BHR), not on the Pattern History Table (PHT), and included a recovery mechanism after obtaining the actual branch outcome. Although our branch predictor has high accuracy, introducing speculative updates still slightly reduced the prediction accuracy (from 87.97% to 87.20%).

This reduction may be caused by too many in-flight branches, the likelihood of multiple branch mispredictions increases exponentially even if the individual branch error rate is low. Consequently, the BHR is more likely to require recovery, resulting in a decline in prediction accuracy.

We also tested different GShare BHR sizes and initial saturation counter values, and found that the best performance was achieved with an 8-bit BHR and a saturation counter initialized to **Weak Taken**.

Based on the above data, we ultimately selected GShare (8-bit BHR, PHT initialized as Weak Taken, non-speculative update) as our branch predictor, achieving a balance between performance and hardware overhead.

G. Experiments

We also ran a series of experiments tweaking the configurations of the processor.

The first experiment we ran was impact of different branch predictors. The average CPI of each branch predictor on the C benchmarks is show in figure 13. We implemented a total of 8 branch predictors: Never Take, Always Take, Bimodal, Gshare, GAg, PAp, PAg, GAp. The figure doesn't show NT and AT due to space contraints. From the experiments, we can see that different branch predictors had very small impact on the average CPI of the processor.

The second experiment we ran was the impact of changing superscalar ways of our processor. The average CPU can be seen in figure 14. It seems like the superscalar ways doesn't improve the performance of the processor more after having a 3 way machine. Most of the times the machine is just processing 1 to 2 instructions in parallel. In addition, a wider

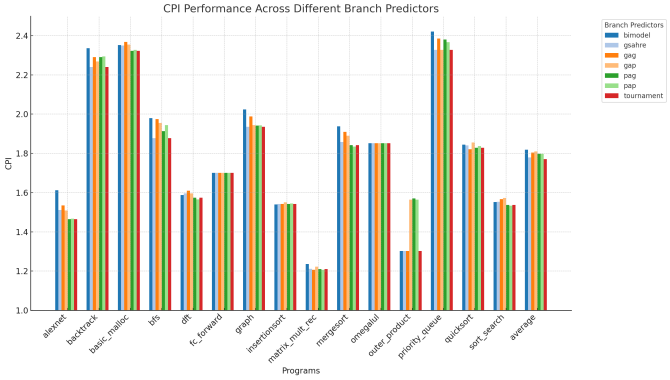


Fig. 13: C CPI for each BP

machine will cause a higher icache miss rate, causing more stalls in the fetch stage.

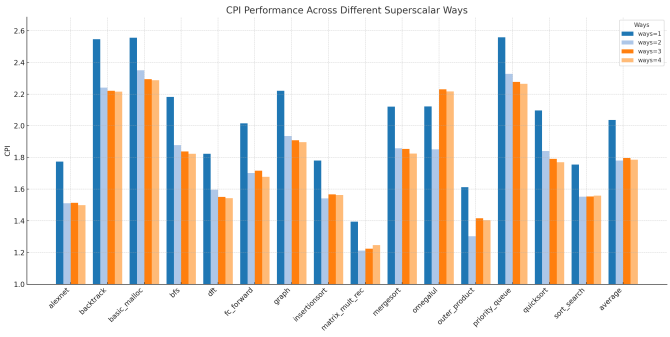


Fig. 14: C CPI for each superscalar way

We also examined the impact of changing the number of lines prefetched in the i-cache, as shown in Figure 15. The results indicate a significant benefit from increasing the prefetch count. For instance, raising the count from 0 to 4 nearly halved the CPI. However, diminishing returns are observed with further increases. Although not fully shown in the graph, we later determined that a prefetch count of 19 yielded the optimal CPI. Increasing the count beyond this, to 24 or 32, degraded performance due to over-prefetching. This is probably because over-prefetching can replace instructions which are after a branch in the icache, causing subsequent branch predicted PCs to miss in the I-Cache.

We also conducted brief experiments with varying instruction buffer sizes, as we found that the instruction buffer size significantly contributes to stalls in both the fetch and dispatch stages. Figure 16a presents the results of these experiments. Although increasing the size of the instruction buffer slightly reduced the average CPI (about 0.02), the improvement was not significant and diminishing returns were evident. Additionally, we observed that increasing the instruction buffer size beyond 8 impacted the clock period, so further experiments were not pursued.

Since the dispatch stage is also heavily bottlenecked by branches, we also tried tweaking the size of the bmask register. This is shown in figure 16b. However, it turns out that

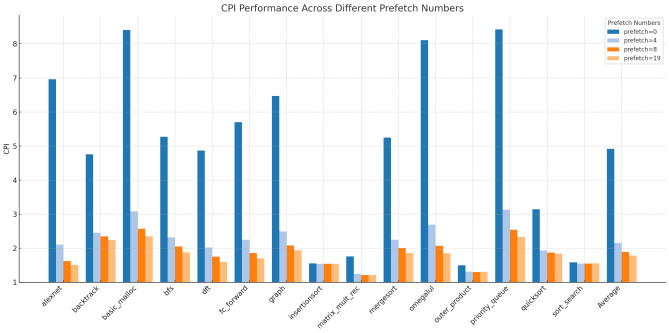


Fig. 15: C CPI for each prefetch count

changing the number of the bmask register/branch stack had nearly no effect on the average CPI.

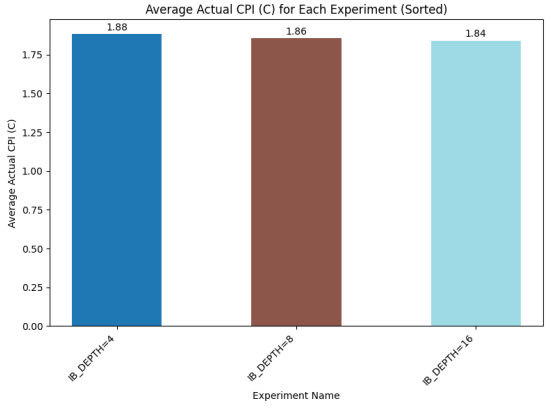
We also suspected that many stalls in RS issuing were caused by previous loads occupying the load functional units while waiting for d-cache responses. To investigate, we experimented with different numbers of load functional units (load buffer sizes), as shown in Figure 16c. Increasing the number of load functional units from 2 to 4 resulted in a noticeable CPI improvement. However, increasing the count further to 8 provided minimal additional benefit and negatively impacted the critical path due to the issue stage's priority selector. Therefore, we did not conduct further experiments. Later, we discovered that issue stalls for load instructions were primarily caused by dependent load-store pairs, rather than the latency of load instructions in the load buffer. This finding explained why increasing the number of load functional units beyond 4 had limited impact.

We also tried using different

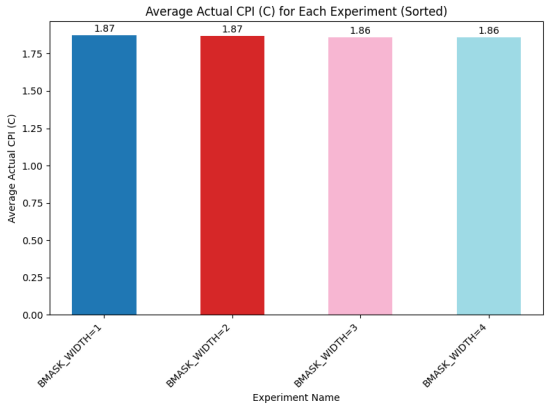
H. Critical Path Optimizations

1) Code Parallelization And Dependency Optimization:

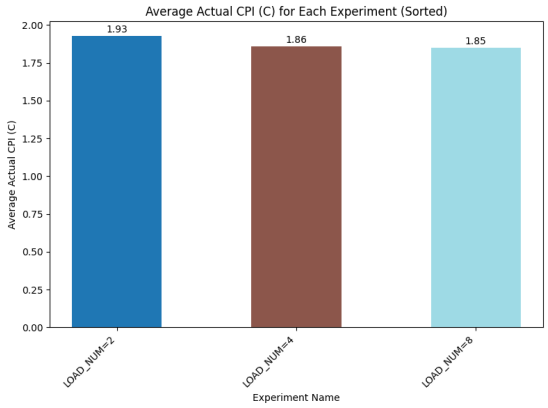
- Dcache: We restrict the dcache to only process 1 processor request per cycle to reduce the complexity of its logic and prevent long critical path.
- Same-cycle forwarding: In our early designs, lots of the modules had "forwarding" mechanisms that "forwards" the received update signals to read signals in the modules. For example, the branch predictor will first update its PHT using resolved branches, and then predict the branch with the updated PHT. This results in a connection of different stages, which is very undesirable for critical path. We removed all these "forwarding" later.
- FSM Optimization: Complex finite state machines (FSMs) handling multiple operations within a single state greatly stretched the critical paths. To address this, we divided large states into smaller, single-purpose states. Each state now only works on a single operation, reducing transition complexity and improving timing efficiency without compromising CPI.
- Dependent Control Flow: Initially, our code was highly dependent due to using next-state logic as the condition for loop iterations. This turned the loop/if statements



(a) Instruction Buffer Size



(b) Bmask Size



(c) Load Number

Fig. 16: Average C CPI Comparison for Various Sizes

highly dependent, worsening the clock period.We resolved this by basing loop conditions on the current state rather than next state. This decoupling reduced dependencies, allowing faster state transitions and significantly shortening the critical path.

- Computed to state: Many modules' sizes were computed on the fly using their valid bit vectors and the `$countones` function. This can result in a long critical path when the bit vector is very long. We changed these to be register based values that are computed at the end of each cycle instead of on the fly. This was the last piece of puzzle in reducing our critical path under 8ns.

2) Module Parameter Tuning And Performance Counter: To optimize performance, we used a controlled variable method to analyze the impact of various parameters on the critical path during synthesis. By adjusting specific parameter values, we evaluated their influence on clock period and performed trade-offs with CPI to determine optimal settings for critical modules.

- ROB: From our occupancy counters, we found that ROB had a very low utilization rate, while being a major contributor to our critical path. We eliminated this by reducing ROB's size from 32 to 24.
- RS: Similarly, the RS has a relatively low utilization rate as well, and the priority selectors at the issue stage was once a major bottleneck of our clock period. We eliminated this by reducing the RS's size from 32 to 16.
- Superscalar ways: From our occupancy counter, we found that a wider machine will cause much more icache miss, which will give us essentially the same performance for any width above 2 and icache miss is a major fetch stage bottleneck. We decided to reduce our width from 4 to 2, resulting in a much shorter critical path.

VIII. Limitations

We were proud of our time management on this project, which allows us to implement performance counters, which helped us gain valuable insights we couldn't have gained otherwise. However, there are still some shortcomings worth noting. First, we failed to fully recognize the potential impact of the victim cache on the D-Cache, resulting in minimal performance improvements from this feature. Second, we did not fix the minor bugs in the ETB implementation. Additionally, at the beginning of the project, we think that the clock period increase caused by associative caches will outweigh the potential CPI reduction, leading us to abandon attempts to implement it. Lastly, in the final hour of writing this report, we discovered a minor error in the LSQ that caused certain load instructions to remain in a specific state for a few extra cycles. After fixing this bug, the CPI improved from 1.7798 to 1.7360. Although we updated the main branch, the deadline for code submission had already passed. We were also not satisfied with the CPI we achieved, as we think our processor was bottlenecked by factors that were out of our sight even after implementing the performance counters. Almost any parameter tuning in the late stage of our project had little effect in improving the CPI. Given more time, we would conduct more in-depth analysis and identify the cause of this bottleneck.

We have reflected on these shortcomings and identified areas for improvement in future engineering projects. Before implementing features, we will place greater emphasis on reviewing relevant literature. When making tradeoff decisions, we will avoid relying solely on subjective assumptions. Finally, after completing a feature, we will thoroughly verify each state and perform more detailed debugging to ensure correctness.

IX. Project Management

We went through very dramatic changes in our project management as the project progressed.

At first, we started by assigning two people per feature each with a defined deadline. However, this management strategy suffered from constant unmet deadlines and difficulties in communications between team members.

Later, we went through a period where most work is centralized on the team's manager, including finishing and testing many of the out-of-order core modules and integrating the processor. While this did improve the overall efficiency of the team a bit, this strategy suffered from imbalanced work distribution and low average productivity, and also led to most of the members not understanding each part of the processor.

After that, the team's manager tried to help the team members grow by using a rotation manager strategy, where we planned to let each member become the manager of the project for a while throughout the project. The goal of this strategy was to allow each member to get a better sense of the codebase and how the project is structured on a high level. While the goal was well-intentioned, this strategy was executed poorly, and did not last long.

Then, the team's manager tried to hold pair programming sessions with the members and work on debugging the processor together during in-person meetings. We had very frequent collaborative programming sessions during this period, and this helped the members understand the whole codebase better and greatly improved their programming ability. This strategy has successfully improved the skill level of most members.

As the members grew, the manager was able to shift a large portion of work to the other members for better work distribution. We gradually formed the role assignment as Table IV:

At the late stage of the project, the team mostly worked by having members propose and implement new optimizations, and the members will propose deadlines to the manager to let the manager decide whether the deadline makes sense or not. On each feature, the testing engineer will check the performance gain from the optimization, and the manager will decide whether to merge the feature into the main branch of the project.

TABLE IV: Role Assignment

Member	Role	Description
Ruochong	Engineering Manager	Propose high level optimization ideas, provide implementation guidelines, maintain project progress
Jirong	Tech Lead	Provide help and work with other team members, helps implementing and debugging new features
Ruiqi	Engineer	Work mainly on the memory subsystem and help other team members debug and implement optimizations
Yunpeng	Engineer	Work on everything fetch and branch related, advanced feature optimizations
Jiacheng	Operation Lead	Test the project and collect data, maintaining progress records and testing infrastructure

Acknowledgements

We would like to express our gratitude to Professor Krisztian Flaunter and Professor Nathaniel Bleier for their insightful lectures and guidance in the field of computer architecture. Special thanks to Bradley and Mustafa, the most exceptional GSIs we have encountered at the University of Michigan, for making the class experience truly outstanding.

We also extend our thanks to Jonah, Jaccob, Advait, and Ian for their consistent support in answering questions about lecture material and project work during office hours. Your help was invaluable.

We are grateful to our classmates for fostering an active and engaging learning environment throughout the course.

Finally, we deeply appreciate the hard work and dedication of ourselves. Despite having diverse career goals, we all remained committed to the project, tackling challenges together and making significant contributions. It was a privilege to collaborate on such a demanding yet rewarding project, and we are proud of what we accomplished this semester.