Tate: An N-Way Superscalar RISC-V Computer Architecture

Shourya Bansal
bansalsh@umich.edu

Ayan Chowdhury
ayanc@umich.edu

Aaron Rahman
cyber@umich.edu

Yunxuan Tang
yunxuant@umich.edu

Akshay Tate
artate@umich.edu

Ivan Wei
ivanwei@umich.edu

***Abstract*—We introduce Tate, an N-way superscalar RISC-V out-of-order computer architecture designed to balance performance and modularity. Building on the foundational principles of the MIPS-R10K architecture, Tate incorporates advanced techniques, including Early Branch Resolution (EBR) and Early Tag Broadcast (ETB), significantly reducing pipeline stalls and enhancing instruction throughput. The architecture supports flexible superscalar configurations. Through systematic analysis, we demonstrate that a superscalar width of two provides optimal performance, yielding a weighted CPI of 1.64 and a clock period of 7.2ns. Additional features include a robust memory subsystem featuring non-blocking, multi-banked, set-associative caches with advanced prefetching, sophisticated load-store queues capable of out-of-order memory operations, and a modularized branch prediction framework employing GShare. Comprehensive benchmarking confirms Tate's correctness, modularity, and performance.**

I. Introduction

Our team has developed an out-of-order processor designed to balance performance, robust architecture, and a rich set of advanced features. Building upon core out-of-order execution principles, we implemented a scalable N-way superscalar pipeline enhanced with Early Branch Resolution (EBR) and Early Tag Broadcast (ETB) to reduce pipeline stalls and improve efficiency. Through careful design and feature integration, we aimed to create a processor that is not only performant but also thoughtfully engineered and adaptable.

This paper is focused on evaluating the performance the processor's architecture through detailed analysis and benchmarking. We present key performance metrics including weighted Cycles Per Instruction (CPI) and clock period to analyze the impact of various architectural parameters. Additionally, we elaborate on more complex structures like the store queue and caches as well as the design challenges we encountered.

II. Analysis Overview

Our analysis focused on optimizing both the clock period and weighted CPI across our benchmark suite described in Section X.B.[1] Our final processor had a weighted CPI of 1.64 and a clock period of 7.2ns. A full perspective of CPI can be seen in Fig. 1, where our CPIs range from 0.6 to 2.4. These 15 programs were selected for our benchmarks as they represent a realistic set of different computational problems.

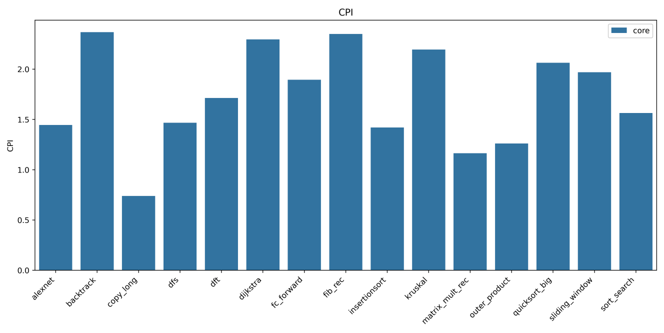


Fig. 1: CPIs of final processor across benchmark suite.

III. Core Processor

A. Superscalar Width

Our processor supports a parameterizable super scalar width. We are able to fetch, decode, issue, dispatch, and retire an arbitrary number of instructions per cycle. We have also made some of these parameters independent of each other. For example, we have an instruction buffer positioned between dispatch and fetch, and we are capable of tuning the number of instructions we dispatch separately from the number of instructions we fetch and decode. All of our modules are tunable, including the number of resolved loads and retired stores. We ensure every stage's width is at least the Common Data Bus (CDB) size. However, we only allow a single branch to resolve each cycle.

The tuning of our super scalar width greatly affected CPI and clock period. The majority of our testing was done with $N \in \{2, 3, 4\}$ as we found larger widths increased the clock period while keeping CPI the same due to memory limitations and the inherent lack of instruction level parallelism in the programs. We found the critical path was highly dependent on the super scalar width, particularly in fetch and dispatch. We achieved clock periods of 7.2ns, 7.8ns, and 8.2ns for widths 2, 3, and 4 respectively. The following graph shows CPI comparisons for representative programs:

[1]We weight the program CPI by the square root of program length. The square root was used to increase the weight of shorter programs.

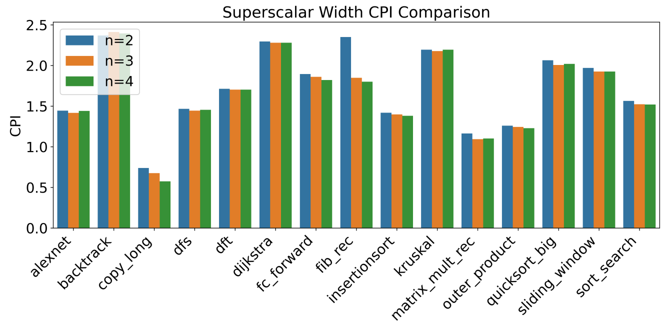


Fig. 2: CPI comparison of different superscalar widths.

In our testing, $N = 2$ yielded the fastest execution time.

B. Core Parameters

Our processor supports arbitrary sizes for the reservation station (RS) and reorder buffer (ROB). As expected, we found wider super-scalar architectures were able to take more advantage of larger sizes; however, when limiting our width to two, we found performance plateaus at a ROB size of 32 and RS size of 16.

TABLE I: ROB Size x RS Size vs CPI

		ROB Size			
		16	*32*	*48*	*64*
RS Size	*8*	1.79	1.71	1.71	1.71
	16	1.78	1.64	1.63	1.63
	24	1.78	1.64	1.63	1.62
	32	1.78	1.64	1.63	1.62

Our final processor had sizes 32 and 16 for the ROB and RS, respectively, which balances both the performance and size, as can be seen in Table I.

C. Early Tag Broadcast

We broadcast tags for completed operations one cycle early to minimize dependency hazards when issuing instructions. Consider two sequential and dependent instructions in a base out-of-order design. Imagine the issue logic places each instruction into a register, and in the next cycle, our functional unit computes its result. If we broadcast our first operation in the cycle it will be completed, then the second instruction will complete two cycles later. However, the value of the first computation is not needed to issue the instruction into a register, it is only needed when executed by the functional unit. We can thus broadcast our tag a cycle early, making the sequence of instructions take three cycles to compute as opposed to four.

Our issue stage takes a single cycle to move instructions from our reservation station into a pipeline register before our functional units. Clock period concerns, which will be highlighted later, have led us to pipeline each of our functional units. This led to our ETB implementation being concentrated on the first stage of each of our functional units. We ensure issue stalls when necessary via structural hazard detection and back-propagation. Each functional unit broadcasts its tag during an operation's penultimate cycle. In the following cycle, dependent instructions are issued. Lastly, the dependent instructions begin execution, reading the correct tag values from the physical register file (written the previous cycle). We were able to benchmark our processor with ETB enabled and disabled, and the average CPI of our benchmarks decreases from 2.1 CPI to 1.6 CPI, giving us a 31% performance boost.

D. Early Branch Resolution

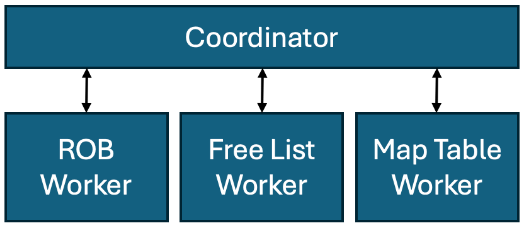


Fig. 3: Simplified diagram of EBR Architecture. More components are checkpointed (not shown).

Without EBR, the pipeline is flushed when a mispredicted branch retires. This costs a lot of cycles. We use early branch resolution to avoid this. We restore key data structures to a checkpointed state during commit when a branch mispredicts, greatly improving CPI.

Our implementation follows a coordinator-worker architecture. The singular coordinator handles high-level control, whereas each worker handles interfacing with a module that needs checkpointing.

The coordinator is responsible for a few things. First, it stores the active checkpoint mask, which is assigned to instructions on dispatch (indicating which branches they are dependent on). It also provides the voided checkpoint ids on a branch mispredict so other modules can clear invalid entries, like the ROB and functional units. Second, the coordinator is responsible for taking checkpoint requests and assigning them ids. Thirdly, the coordinator keeps track of checkpoint dependencies (i.e., nested branches). Each worker listens to the coordinator, which issues commands to save or restore data to the worker's subscribed module. The worker is responsible for storing the checkpoint data of the subscribed module, such as the map table.

This architecture makes the checkpointing system very modular. We have fine-grained control over who gets checkpointing. This flexibility has enabled checkpointing for some other advanced features like the Return Address Stack (RAS) and GShare predictor history.

TABLE II: Checkpoint Size vs CPI

2 Entries	**4 Entries**	**6 entries**	**8 entries**
1.85	1.64	1.64	1.63

The number of checkpoints each worker can hold is parameterizable. As seen in Table II, the CPI decreases plateaus at a size of four. We ultimately set our size to six as differences beyond that were negligible.

IV. Control Flow

A. Branch Predictor

We ended up using Gshare as our branch prediction algorithm. We also tested a variety of other algorithms. These include backwards-taken forwards-not-taken (BTFNT) and a tournament predictor that gradually used larger gshare predictors as they warmed up (cpi ≈ 2.3). We found that using a single 8-bit gshare predictor provided the best accuracy (cpi $= 1.64$) of the three options. The performance of gshare versus BTFNT and the control (always not taken) is shown below.

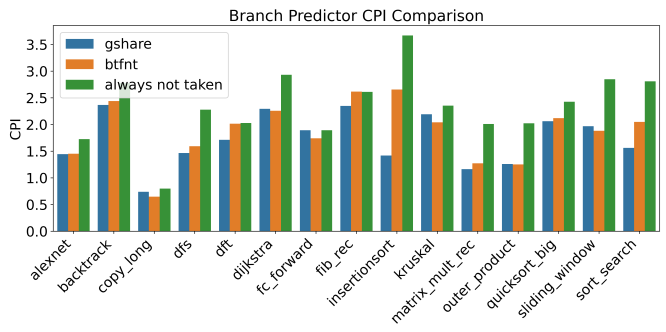


Fig. 4: CPI comparison of different branch predictors

The accuracy rates of the three branch prediction algorithms are also shown below.

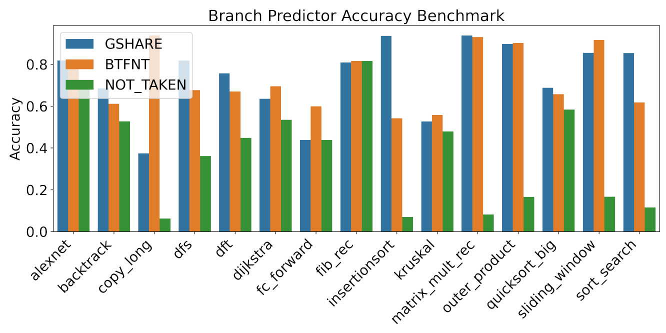


Fig. 5: Accuracy comparison of different branch predictors

B. Branch Target Buffer

The Branch Target Buffer (BTB) implementation recorded statistically insignificant (< 0.01%) changes in weighted average CPI compared to the use of a full adder for target computation in fetch. The BTB itself was directly mapped with 64 entries. Since target computation of branches and JALs can be done without any register reads, the alternate BTB implementation resolves branch targets with a full adder in parallel with the decode stage.

Our group planned to use the BTB instead of the full adder if fetch were on the critical path and if we wanted to push the clock period lower, but due to time constraints, we weren't able to make that decision in time. The implementation remains present in the final submission and can be enabled by using a macro to define the flag `USE_BTB`.

C. Return Address Stack

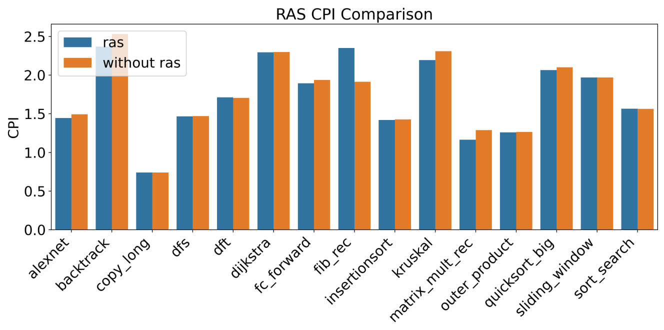


Fig. 6: CPI comparison of RAS

The RAS is responsible for predicting the jump target of JALRs. It does this by maintaining a stack of "call" pseudoinstructions (aka JALRs with rd=x1). On a ret (JALR x0 x1 x0), it predicts the target of the JALR as the top of the stack (and pops it).

Notably, fib_rec is an assembly program that does not follow the calling convention specified in the RISC-V ABI. This led to our RAS doing poorly on it. Excluding fib_rec, enabling the RAS decreases our CPI by 2.0% (with fib_rec, instead decreases by 0.88%).

V. Functional Units

We found that doing a functional unit computation in the same cycle as reading from our physical register file led to greatly increased clock periods. When optimizing, we found that performing addition along with a Physical Register Files (PRF) read would require a clock period of at least 8.3. We thus chose to pipeline each of our functional units to read the PRF in the first cycle and carry out the remaining computations in future cycles. Having each functional unit take multiple cycles would generate data hazards on dependent instructions. This did result in a CPI increase, with our benchmarks CPI going from an average of 1.3 CPI to 1.64 CPI. However, we found the clock period decrease to be too lucrative to pass up. Having each functional unit be pipelined also allowed our ETB implementation to be unified, removing the need to broadcast tags in issue, and allowing all broadcasting to be done in the first or later stages of our functional units. We discuss a different, faster, version of our processor without this pipeline stage in Section IX.C.

Aside from our multiplier, each functional unit only has a single pipeline stage for PRF reads. This includes the ALU for arithmetic operations, the control unit for handling JAL, JALR, and branch instructions, and the load and store units for computing addresses and routing those addresses to the appropriate entries in our load buffer and store queue. Our multiplier was pipelined to be eight stages to fit our clock period.

VI. Memory Subsystem

Our Memory Subsystem was one of our most sophisticated parts of our processor. It involved a few major components, each of which impacted performance in different ways. Our instruction cache and prefetching had a significant impact on instruction throughput from the frontend, greatly decreasing CPI. When it came to the core processor, we noticed that the largest challenge came in the form of the data cache interfacing with our load buffer and store queues. The complexity in the number of places that correct data could be led to some challenging forwarding situations, but after all debugging was complete, our overall memory system was robust in all test cases and minimized our load and store instruction latency.

A. *Instruction Cache and Prefetching*

The instruction cache contains two cache banks and two prefetch stream buffers of size two each.

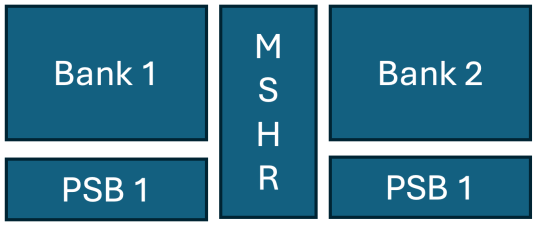


Fig. 7: Diagram of Instruction Cache Architecture

a) *Servicing Reads:* Fetch sends the current program counter (PC) to the instruction cache. The instruction cache aligns the PC by cache line. Then, starting at the aligned PC, the cache calculates two consecutive cache line addresses, one sent to each bank.

In parallel with the bank reads, queries for both of these "bank read addresses" are sent to the prefetch stream buffers. The prefetching stream buffers store a contiguous stream of prefetched instructions. Each stream buffer is a FIFO queue, and each entry contains an allocated bit, a data valid bit, an address, and space for a cache line. If the minimum bank read address (equal to the line-aligned PC) matches the minimum-address allocated stream buffer entry, then that stream buffer is considered to be *active*. An active prefetch stream buffer will pop all data-valid entries corresponding to queries and serve the data in those entries.

Now, in consecutive order starting from the aligned PC, the instruction cache serves data granted by either the banks or the stream buffers up until the first address that isn't present in either. If a prefetch stream buffer contained the data but the bank didn't, the data is promoted from the prefetch stream buffer into the bank.

b) *Fetching with Prefetch Memory Requests:* All allocated entries in a prefetch stream buffer represent some potentially outstanding memory transaction. Once the memory transaction completes, the incoming data is written to the allocated entry, at which point that entry becomes *data-valid*.

Each stream buffer has an associated prefetch address pointer. Each cycle, each of the stream buffers requests the processor's memory arbiter. Any stream buffer whose prefetch request is granted will then advance its prefetch pointer. If the buffer is not yet full, the requested previous address will also be given its own allocated entry in the buffer.

It is an invariant that all addresses starting from the minimum allocated and non-data-valid entry and up to the address a line before the prefetch pointer have outstanding memory transactions that were granted in chronological order. When combined with the fact that the provided memory module guarantees that transactions are completed in order, the prefetch pointer can advance arbitrarily far ahead of the buffer's allocated entries' addresses. In an active prefetch buffer, once a read pops an entry, the buffer "catches up" to the prefetch pointer by replacing the popped entries with newly allocated entries that have the next several addresses. Note that if the prefetch pointer is ahead of all a buffer's entries, the buffer must be full.

If neither buffer ends up being active, one is chosen to be evicted and its prefetch pointer reset to the cache-aligned PC. This deals with cache misses because the prefetch pointer of the evicted buffer will begin to request at the missed address.

Inactive buffers were active at some point in time, and thus they store instructions that were slightly ahead of the PC and requested from memory at some point in time, before some PC change occurred; if these instructions are immediately available in the buffers when we return to the PC and the PC change doesn't occur (e.g., a branch is not taken), the inactive buffer becomes active and the data can be used. This is the reason to have two buffers.

In summary, promotion from the buffer is conservative, while memory requests are made aggressively. This simultaneously takes full advantage of memory requests every cycle, while avoiding bad evictions from the banks.

c) *Results:*

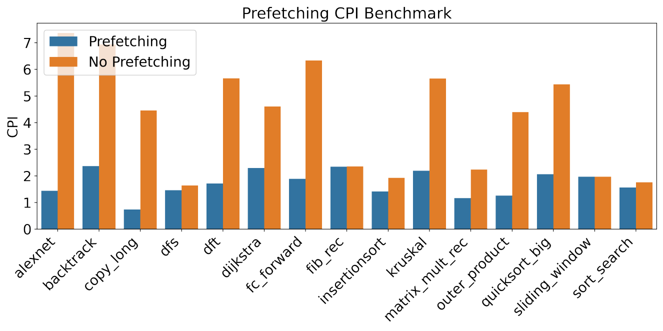


Fig. 8: The most significant performance gains from prefetching can be seen in programs that are too large to fit in the instruction cache, causing thrashing.

B. *Store Queue*

Stores in our processor are dispatched in order and assigned an entry in our FIFO store queue on dispatch. Store instructions will go through our reservation station to our store functional units to compute addresses and data. In the functional unit, addresses will be word aligned, and a mask will be generated for the bytes being stored in order to make store forwarding simpler.

a) *Store Queue States:* Each entry of our store queue has three associated bits. A bit for if the store has its address, a bit for if the store has been retired, and a bit for if the store has been committed to cache. A parameterizable number of entries attempt to commit to the cache if their retired bit is set. The same number of entries attempt to be cleared in order if their committed bit is set. The setting of each bit is not forwarded for committing to the cache or for clearing the entry. This is in part due to clock period concerns and in part to ensure certain invariants for our load buffer.

b) *Parity Handling:* One additional design decision was deciding how to determine if our store queue is empty or full if the head and tail are equal. We considered designs forcing the tail to always be ahead of the head, and other designs where each entry would be assigned some parity. Eventually, our parity idea was refined to have a parity associated with each pointer into our queue as opposed to a parity for each entry. This would be very cheap and easily fit with our EBR checkpointing system.

c) *Parametrization:* We found a store queue size of 8 to be ideal for our clock and CPI tradeoffs.

C. Load Buffer

Our load buffer is split into 3 modules with distinct tasks. We have a buffer module, containing a parametrized number of entries. Each entry is handled by a load handler module. Each load handler module then has four byte loader modules.

a) *Load Buffer Module:* The load buffer communicates with dispatch, the load functional units, the data cache, the CDB, and the PRF. It maintains the number of free entries and associated slots to give to dispatch. It will then route addresses from functional units to their corresponding load handlers. Requests to our cache from each load handler will be arbited by our load buffer to the cache, and data from the cache will be properly routed to the correct load handlers. Completed loads will be arbited by the CDB and written to the PRF and then cleared from the buffer.

b) *Load Handler Module:* Each load handler is a state machine consisting of 4 states. There is the empty state, the waiting for functional unit state, the waiting for data state, and the complete state. In the entry state, the load handler will signal to the load buffer that it is free and wait for it to be assigned to a load from the load buffer. It will then transition to the waiting for functional unit state in the next cycle. As load handlers are assigned in dispatch, there will always be at least one cycle between a load handler being assigned to a load and reviving the address of that load from its functional unit. Once an address has been computed from the functional unit, which we have guaranteed to be word aligned, we transition to the waiting for data state. In this state, we immediately speculatively ask the cache for data irrelevant of what we view in the store queue. Each byte loader is responsible for determining what the correct value of each byte is. If the byte loaders have loaded in the bytes being requested, then we transition to the completed state. Once the load buffer arbits our handler, we transition back to the empty state.

c) *Byte Loader Module:* Four byte loaders are responsible for loading the data for each handler. Each byte loader has three state bits. A bit for if the store queue has been explored, a bit for if the loader has data from the cache, and a bit for if the loader has data from the store queue. Cache and store queue checks are handled concurrently. If a byte loader receives data from the cache, it will store that byte of data and flip the associated bit. Each byte loader will receive the current store queue head, the tail associated with its load, and the current state of the store queue. Masks over the store are created for entries that don't have addresses, as well as entries that correspond to the data we are attempting to load. These masks will be shifted to align the associated store queue tail with the highest bit of the mask. The highest set bit will then be selected. Based on the selection of our masks, we populate a byte with data from the store queue and mark our store queue bits accordingly. One additional optimization we discovered is that loads could be completed with unknown stores in the queue if those stores are later than known stores to the correct address. After our store queue and cache checks have been concurrently processed, we say that the byte has been loaded if the entire store queue has been explored, and either we have data from the cache or the store queue. Each byte loader can then be reset when a handler receives a new load.

d) *Parametrization:* The main points of parametrization in the load buffer were its size and which states could forward to other states. Forwarding leads to large clock period increases, so we elected to have all state transitions take a full cycle. For load buffer size, we decided on 8 due to clock period concerns.

D. Data Cache

Our data cache is a non-blocking, (parameterizably) multi-banked, (parameterizably) set-associative, write-back cache with modular eviction policies. This structure allows us to easily configure and try different configurations and extract optimal performance. Fig. 9 gives an overview of how the cache would be laid out in hardware to achieve the abilities mentioned. It consists of a) a Miss-State Holding Register Table (MSHR), b) Multiple SRAM Banks, and c) a Write-Back Buffer for Lazy Memory Write-Back.

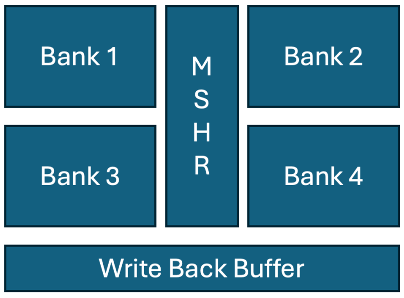


Fig. 9: Overview of the Data Cache.

a) *General Overview:* When the cache receives a memory read request or write request from the Load Buffer or Store Queue, it CAMs all the places the address and its corresponding data may be stored—i.e., the Cache Banks, the MSHR, and the

Write-Back Buffer. An address may only exist in a single place at a time. If there are any matches, the relevant data will be masked and then returned to the querying process. If no matches were found, the Cache will attempt to process whichever memory request it can, prioritizing requests based on maximizing throughput of blocking instructions (i.e., loads).

b) *Non-Blocking (MSHR Table):* For the sake of performance, one of the most important components of the data cache is the miss-state holding register table. Because it is critical to the performance of our system, our group did not consider building a blocking cache, and we do not have data to calculate its benefit. However, a non-blocking data cache with our 7.2ns clock period would allow us to have 13 in-flight memory requests at a time, as opposed to just one per cache. We estimate that removing the MSHR would increase our CPI by an order of magnitude. In the MSHR, we support partial write forwarding. The biggest shortcoming of our MSHR is that the final implementation does not take advantage of all the MSHR entries. Instead of treating the MSHR like an additional cache buffer (which would lead to an effective increase in cache size by 50%), it is currently used as a "pass-through", where as soon as a memory transaction completes, the data immediately enters the cache and leaves the MSHR table. Our rationale was that we want our processor to be as realistic as possible—in a real memory system, memory responses are not guaranteed to be FIFO, so creating a FIFO for the MSHR would have been unrealistic.[2] We did test how a FIFO would impact performance, and noticed that it decreased CPI marginally, but by less than 0.05—which was our threshold for making the change.

c) *Write-Back Buffer:* For our write-back buffer, we attempted to employ multiple strategies in order to increase its performance. We treat the write buffer as a pseudo-victim cache, without promotion. Any data that is in the write buffer remains there until the buffer is full and one of the cache banks evicts dirty data.[3] The "lazy eviction" from the write buffer allows us to serve a memory request that has already been written to memory. In tandem, the Write-Back Buffer also uses its own modular eviction policy (like the cache banks), which lets us fine-tune performance and eviction based on what workloads we expect the processor to be used for. Table III demonstrates how the size of the Write-Back buffer impacts cache hit rate. Our final buffer was size 8.[4]

[2]One area we hope to explore in the future is also using an eviction policy in the MSHR, and see if that would improve performance at all; this would also help us start to build a system that would interface with a real-world multi-leveled memory system.

[3]The fact that initially clean data is not stored in the write buffer is a big reason that this is different than a victim cache, beyond the lack of promotion. This is an area we hope to further analyze and optimize.

[4]We are aware of the 4-line limit, but with approval from Bradley and Jonah (April 4th), we combined our write buffer and victim cache into a single larger write buffer without promotion to the main cache. This was deemed acceptable, as a write buffer is strictly less effective than a victim cache.

TABLE III: Write Buffer Size vs Hit Rate

2 Lines	**4 Lines**	**8 Lines**	**16 Lines**	**32 Lines**
96.75%	96.92%	97.22%	97.38%	97.73%

d) *Banking Cache:* The primary purpose of our cache banking was to allow us to serve multiple loads at the same time—essentially increasing the number of ports that the LSQ was able to use to query the Cache. In addition, each cache bank had multiple separate SRAM modules—one per way. This allowed us to then, in parallel, query each way to extract the relevant data. This also has the benefit of completing more than one cache operation in a cycle: if a load and store were to access the same cache bank, but hit on different ways, each of those is able to happen concurrently. When doing operations on a single cache bank way, the ultimately selected prioritization scheme (built from first principles) was a) prioritizing any memory responses (because of the MSHR Design), b) then prioritizing loads, and finally c) completing stores.

e) *Associativity:* The cache is parameterizable and associative. The following table shows our hit rate versus associativity:

TABLE IV: Cache Associativity vs Hit Rate

2 Way	**4 Way**	**8 Way**
97.035%	96.929%	97.218%

Based on the data in Table IV, we determined that associativity did not have a statistically significant impact on cache hit rate, and seemed to be more correlated with the specific test case than the associativity at large. Thus, there were not many conflict misses, which highlights effective eviction, which we will expand on next.

f) *Modular Eviction Policies:* The emphasis of this processor was modularity. On top of the overarching design (i.e., associativity and banking) being modular, we also emphasized modular interconnects. Namely, the eviction strategy that our data cache uses is completely modular, with an easy-to-use interface. We built four different eviction strategies, and thoroughly tested three of them: Least Recently Used LRU), FIFO with Second Chance (Clock), Static Re-reference Interval Prediction (SRRIP), and Bimodal Re-reference Interval Prediction (BRRIP). As both the data cache and the write-back buffer can employ disjoint eviction policies, we compared hit rates when using each of them:

TABLE V: DCache Eviction Policy x Write Buffer Eviction Policy vs Hit Rate

		Cache Bank Eviction Policy		
		SRRIP	*LRU*	*Clock*
Write Buffer Eviction Policy	*SRRIP*	97.22%	96.74%	96.30%
	LRU	97.11%	96.26%	96.17%
	Clock	97.15%	96.42%	96.15%

Ultimately, as Table V indicates, the best policy tested was Static Re-Reference Interval Prediction, which is what is used in our final submission.

One important note was that some of these eviction policies are atypical—for each of the policies in our final design, the "eviction index" was a cycle delayed. This was done to ensure that the cache was not on the critical path[5].

g) *Cache Sizing:* For this project, the cache size is fixed; however, we did some analysis on how our hit rate might change based on cache sizing, to help differentiate between Compulsory and Capacity Misses[6].

TABLE VI: Cache Size vs Hit Rate

256B	512B	1024B	2048B
97.218%	98.237%	98.565%	98.792%

As Table VI demonstrates, of the 2.8% miss rate that our cache had, about 1.2% of those were capacity misses, which indicates our effective cache performance. Overall, our capacity-conflict miss ratio was quite low, demonstrating effective overall cache performance.

E. Memory Bus Arbiting

One of the more innovative aspects of our design is the memory bus arbitration mechanism. While our initial approach followed the lab recommendation—prioritizing all data cache operations over instruction cache accesses—we observed that this strategy limited prefetching effectiveness, especially in store-heavy workloads.

To address this, we developed a more nuanced arbitration scheme:

1) **Highest Priority:** Forced memory reads triggered by hazards in the write-back buffer. These are critical for ensuring correctness and are serviced immediately.
2) **Next Priority:** Memory loads from the data cache, as these directly impact execution progress and need timely handling.
3) **Prefetching / Instruction Fetches:** If no immediate data loads are pending, the memory bus is allocated to instruction cache prefetching, ensuring a steady supply of instructions.
4) **Lowest Priority:** Memory stores are deferred and handled lazily. Given our support for infinite prefetching, this ensures that store operations do not block the pipeline unless absolutely necessary.

This strategy allowed our processor to balance correctness, performance, and prefetching efficiency, particularly under store-heavy conditions.

[5]The rationale here was that the eviction index being a cycle delayed *shouldn't* impact performance significantly, as that was analogous to every load taking an extra cycle due to pipelining or some hazard.

[6]The associativity section should demonstrate the impact of Conflict misses.

VII. Testing

A. Unit Testing

Throughout the development of the project, we have written unit tests for nearly all modules. This greatly simplified later integration testing, but also served as a place to verify individual modules' synthesis performance.

B. Integration Testing

Throughout integration, we developed specific test programs to test new features of the project. Initially, we focused on simple programs without branches and memory operations, then expanded to programs with branches to stress our EBR system, and finally expanded to full C programs with intense memory operations to specifically test our store-forwarding and cache correctness. Specific test programs can be found in Table VII.

TABLE VII: Test Program Descriptions

Program	Description
simple	Tests core processor
func_call	Tests EBR and RAS
cache_slam	Tests store forwarding and data cache
stress_mem	Tests data cache

We also compiled our C programs with various optimization levels using the `-O` flag, and ran with a variety of clock periods and parameter adjustments in modules, to generate different interleavings of program execution to expose bugs in the processor.

C. Visual Debugger

Our visual debugger was an incredible debugging resource, additional information can be found in Section XI.

VIII. Project Management

A. Workflow Style

From the beginning of the project, our team strived to implement as many advanced features as possible. During our milestone one meeting, our original goal was to include N-way, ETB, and EBR. During our second milestone meeting, we expanded our goal to include nearly all of the advanced memory subsystem features. Our final milestone sprint focused mainly on correctness and prefetching. The last week was targetted towards optimizing our clock period.

We used a variety of organizational tools to help keep the team on track. Discord was where most of our communication happened. We allocated one channel to each major component (i.e., branch prediction, the cache, etc.). We also kept a channel with a running list of future optimizations we wanted to do. For important issues and bug tracking, we used the Linear issue tracker to understand their status. When building complex docs on new modules, we used Slab to write specs and implementation logic out. Finally, all work was developed on separate GitHub branches. We required a formal peer review and evidence of passing a test suite before a branch could be merged into the main branch.

B. Division of Work

Contributors are listed alphabetically by first name. While all members worked with all portions of the processor, each individual took specific ownership of individual components to maximize throughput of features[7]. Aaron contributed to the branch predictor, EBR, RAS, project management, and testing. Akshay contributed to the caches, clock period optimization, decode, dispatch, instruction queue, PRF, ROB, and VCD parser. Ayan contributed to the ETB, functional units, issue, load buffer, RS, and store queue. Ivan contributed to the BTB, branch predictor, fetch, prefetching, and RS. Shourya contributed to the caches, clock period optimization, EBR, ETB, functional units, and issue. Yunxuan contributed to the decode, instruction queue, ROB, functional units, store queue, and VCDIVE.

IX. Future Work

A. Multiple Reservation Stations

While optimizing the clock period of our programs, we found that issuing out of a large reservation station to a large number of functional units became our clock period bottleneck. Ultimately, we reduced the number of functional units to our superscalar width, eliminating this issue. However, we hypothesized a solution that uses two reservation stations with disjoint sets of functional units. Such a solution would allow us to issue twice as many instructions at the clock period cost of our original reservation station.

B. Split ALU

As mentioned in Section V, we introduce a pipeline stage in our ALU to reduce our clock period. This came at the cost of an increased CPI. We hypothesize that splitting the ALU into multiple smaller ALUs, some with pipelining and some without[8], could increase the CPI of our processor while keeping our extremely low clock period. A similar technique could potentially be applied to the load and store units.

C. Taylor's Version

In our optimization process, we had focused very heavily on the clock period. We realized that this resulted in a heavily sacrificed average CPI. In our current design, our major CPI gains are from EBR, ETB, GSHARE, and a variety of features from our memory system. However, we realized that dependent instructions could substantially reduce our CPI, even with ETB, due to all of our functional units having an extra pipeline stage to read from the PRF. While this pipeline stage remained in our submitted design, we were able to play around with removing this stage after the deadline.

Removing the PRF read pipeline stage led to a large increase in clock period, as our functional units would now be on the critical path. Our clock period went from 7.2ns to greater than 8.5ns. However, this extra clock period afforded us room to increase the size of our buffers and add further forwarding paths throughout our processor. Tuning parameters resulted in a processor with a clock period of 8.75ns and an average of 1.2. This would be an improvement of around 10%, with larger gains on bigger programs. Unfortunately, we were not able to submit this design, but we found it enlightening to see the benefits of optimizing for both CPI and clock period, instead of optimizing for just one.

X. Appendix

A. Advanced & Interesting Feature List

Throughout the development of our Architecture, we implemented multiple advanced features and other interesting additions that we thought would be nice to share. Below, we have listed an overview of the advanced features that we have implemented.

Core Processor
- N-Way Superscalar
- Early Branch Resolution (EBR)
- Early Tag Broadcast (ETB)
- Branch Prediction
  - GShare (Parameterizable)
  - Backwards Taken, Forward Not Taken (BTFNT)
  - Tournament Branch Prediction (GShare, BTFNT)[9].
- Return Address Stack (RAS)

Load Buffer and Store Queue
- Out of Order Load Issues and Memory Requests
- Store Forwarding
- Internal Speculation

Cache and Memory
- Non-Blocking Data and Instruction Caches
- Multi-Banked (Parameterizable)
- Set Associative (Parameterizable)
- Write-Back Caches
- Instruction Cache Prefetching
- Lazy Write-Back Buffer with Data Forwarding
- Modular Eviction Policies
  - Static Re-Reference Interval Prediction (SRRIP)
  - Bimodal Re-Reference Interval Prediction (BRRIP)
  - Least Recently Used (LRU)
  - FIFO with Second Chance (Clock Algorithm)

Miscellaneous
- Modular Visual Debugger (VCDive)[10]
  - Supports VPD Files for Large Struct Breakdown
- Lighting-Fast VCD Parser[11]

[7]At the cost of a small latency hit.

[8]Based on the latency of instructions (e.g. , shifting operations would not be pipelined).

[9]We noticed that there was no performance improvement with the tournament prediction strategy, so our final synthesis run does not use the tournament prediction strategy

[10]Due to the modularity of the infrastructure that we built, we have open-sourced VCDive here.

[11]You can find the parser here

- Robust Unit Tests through the development process
- SFloat: A fixed-point division-free number library designed for scientific computing
  - Supports Basic Arithmetic Functions, as well as Square Root, Power, Exponentiation, Logarithm, and Basic Trigonometry
- Distributed Synthesis Sweep Automation
  - Supports host finding and clock period sweeps.

B. *CPI & Benchmark Suite*

Individual program CPI (cycles per instruction) is calculated by taking the total number of cycles, inclusive of necessary flushing, and dividing it by the number of instructions retired.

Our benchmark suite is a mix of both provided C and ASM programs, along with additional C programs such as dfs, dijkstra, kruskal, and sliding_window.

TABLE VIII: Program Descriptions

Program	Description
dfs	Implements a depth first search on a 12×12 grid
dijkstra	Implements Dijkstra's algorithm on a 7×7 grid
kruskal	Implements Kruskal's algorithm on an 8-vertex graph
sliding_window	Implements a sliding-window sum of 1024 integers

XI. Visual Debugger (VCDive)

Inspired by Deric Dinu Daniel's visual debugger from Fall 2024, our team decided to develop **VCDive**, an open-source web-based visual debugger for own own processor as well. After communication with Deric Dinu Daniel, our team decided to use a similar toolset with Next.js for the frontend, Python Flask for the backend requests, and a novel C++ based parser using PyBind.

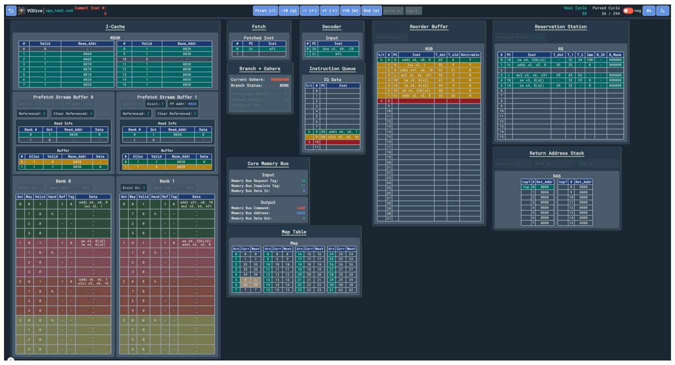


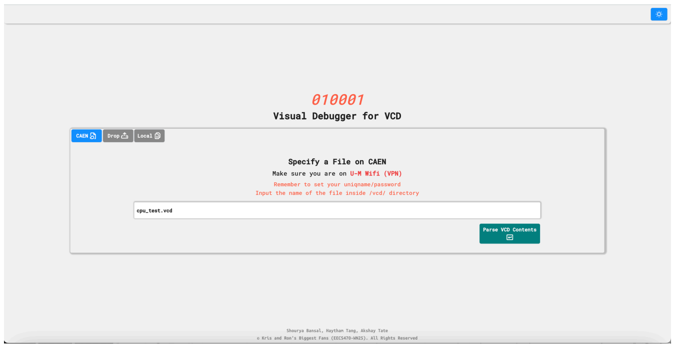
Fig. 10: Snapshot of VCDive

As the name suggests, VCDive will parse the Value Change Dump (VCD) files and extract meaningful signals at different timestamps as the CPU is running. With structural and color-encoded rendering, we can see the state of each component in a particular cycle. For example, we can see the head and tail of the Reorder Buffer and what instructions have been dispatched to it so far. The ability to display information nicely allows us to step into cycles and trace the instructions in-flight and changes happening to each module. We can index into any cycle by inputting the cycle number. Later, when we wanted to use the visual debugger in a more advanced way for difficult bugs in LSQ and Cache, we would track the line-count write-back file first differs with the ground truth, and look up several cycles' states around that cycle based on the *Committed Inst #* feature.

Although the major functionalities of VCDive share similarities with Deric Dinu Daniel's visual debugger, we have made several significant contributions that improved the debugging experience. These improvements make VCDive more modular, portable, dynamic, and faster.

A. *CAEN Forwarding*

All simulations are run on CAEN with the VCD files generated on CAEN computers. To parse and debug them on a visual debugger hosted locally, one would typically have to manually download the files. This is a lengthy process. To address this concern, we have added the **CAEN Forwarding** feature displayed in Fig. 11.


010001
Visual Debugger for VCD

Specify a File on CAEN
Make sure you are on U-M Wifi (VPN)
Remember to set your uniqname/password
Input the name of the file inside /vcd/ directory

cpu_test.vcd

Shourya Bansal, Haytham Tang, Akshay Tate
© Kris and Ron's Biggest Fans (EECS470-WN25). All Rights Reserved

Fig. 11: VCDive CAEN Forwarding Page

We have three ways to specify a file for parsing: caen, upload, and local paths. To use CAEN Forwarding, the user just needs to set up their uniquename, password, path to the vcd directory in their environment variables, and specify the name of the file. CAEN Forwarding will automatically log into CAEN with the user's information under the hood, and prompt a DUO push for two-factor authentication. It will then send a request with the file specified and parse the file. Once the file is done parsing, it will navigate VCDive to the debugger page instantly. This feature streamlines the debugging work significantly.

B. *VPD to VCD*

Traditional VCD files will unpack structs and flatten the bits, which complicates representing custom structures or unions. Instead of using the original Value Change Dump files, we decided to use VPD (VCD Plus) files and then convert them to VCD files with *splitpacked* arguments to keep the integrity of the structs. This brought us several benefits: (1) It became much easier to parse and query the variables. In our parsing, different levels of variables are separated with ".", and structs will maintain their respective ".<data_attribute>" that the visual debugger could directly index and retrieve the value. (2) The entire computer architecture rendering is dynamic and variable on the frontend. Instead of specifying the width of each parameter of a struct and manually changing it when the CPU module changes, VCDive is capable of reading the constants and parameters, such as the size of the cache, and varying the rendering of the components accordingly. If we added a new parameter or changed the naming of a parameter within a struct, we could edit it correspondingly, simplifying the maintenance work.

C. C++ Parser

Parsing large VCD files, even the structured ones from our VPD process, can be slow, especially using only Python. This was a bottleneck, particularly with multi-gigabyte files. To make VCDive faster and more memory efficient, we built a new parser using C++.

Our C++ parser is designed to read the special VCD files (with preserved structures) and quickly extract signal values over time. We used PyBind11 to connect this fast C++ code to our Python Flask backend. When a file needs parsing, the backend calls the C++ parser, which rapidly reads the file and organizes the data efficiently.

Because of this, the backend can retrieve signal data for any simulation cycle almost instantly. This greatly reduces loading times and makes the VCDive debugging experience much smoother and faster for the user, especially when dealing with huge simulation files. For instance, when parsing the BFS VCD file with over 40k cycles, it took Deric's parser about five minutes, whereas it took at most one minute for our parser to run.