# Yet Another Out-Of-Order RISC-V Processor Design: the LC-32i

Final Report - ECE 411 Computer Organization Design - Team lc-32i

Jefferson Zhang
*Electrical and Computer Engineering*
*University of Illinois Urbana-Champaign*
Urbana, IL
jyz4@illinois.edu

Kai Karadi
*Electrical and Computer Engineering*
*University of Illinois Urbana-Champaign*
Urbana, IL
kaick2@illinois.edu

Vaibhav Mattoo
*Electrical and Computer Engineering*
*University of Illinois Urbana-Champaign*
Urbana, IL
vmattoo2@illinois.edu

***Abstract*—In this project, we introduce the LC-32i, an out-of-order RISC-V processor. We designed and implemented the processor in SystemVerilog as part of ECE 411, Computer Organization and Design at the University of Illinois Urbana-Champaign. Our processor utilizes instruction-level parallelism through dynamic, out-of-order scheduling, specifically utilizing the explicit register renaming (ERR) scheme. We further supplement the processor with a dynamic split load-store unit, 2-way superscalar frontend, and additional frontend improvements. Our processor implements the RV32IM ISA, and has been verified and validated on a variety of test programs, and achieved a top 5 ranking in the Spring 2025 course design competition.**

***Index Terms*—computer architecture, processor, RISC-V, out-of-order, ECE411**

## I. Introduction

Typical 5-stage pipelined in-order processors execute instructions first-come-first-serve. However, due to this, they suffer from significant delay during the execute (EX) stage when processing instructions with significant latency. For example, MUL, DIV, floating-point instructions, and even load-store instructions during the memory (MEM) stage occupy the processor for an extended number of cycles. Out-of-order processors allow the processor to take advantage of instruction-level parallelism– that is, instructions can freely flow from fetch to execute to reorder as long as there are functional units available and there are no data dependencies. Instructions are reordered before commit and flushed if a branch misprediction is made, just like a pipelined in-order processor. This reordering is also necessary for load/store coherence.

We implement one of the two common data scheduling methods for out-of-order processors, explicit register renaming (ERR). The other is Tomasulo's Algorithm. In explicit register renaming, we make the distinction between architectural and physical registers. We keep a physical register file and assign architectural registers (those that appear in assembly programs) to physical registers in the register alias table (RAT). We also keep a mapping for physical registers whose mappings are confirmed at commit, called the retirement register alias table (RRAT). Physical registers are allocated during the in-order decode stage (ID), and freed once they are no longer used by a mapping, typically during the commit of the next instruction that uses the same architectural register. Instructions are dispatched and sent to reservation stations (RS) where they wait until they are no longer waiting on a data dependency, linked to another physical register. Then, they are executed and placed on a common data bus (CDB) before being added to the reorder buffer (ROB). The reorder buffer is a large queue, where the head pointer represents the currently committed state, and increments on every cycle where an instruction is ready to commit. If a branch misprediction occurs, it empties further instructions in ROB and flushes all other units of the CPU, setting the PC to the correct address before continuing.

Our processor builds upon the ERR out-of-order baseline by implementing a split load-store queue structure, discussed later, and a 2-way superscalar architecture, which effectively allows two instructions to be fetched, committed, dispatched, and dispatched every cycle, achieving a maximum possible instructions per clock (IPC) of 2. Furthermore, we build upon the baseline frontend configuration, implementing a 16KB 4-way set-associative pipelined read-only instruction cache and a GShare branch predictor. Developing such a processor is an important step in learning computer architecture, as OOO cores are the foundation of many modern-day CPUs. At the same time, working on a larger project taught our team the importance of organization, clean code, and the process of synthesis and optimization.

## II. Project Overview

Our project was split into four checkpoints: one, two, three, and the advanced features checkpoint, and we were given two weeks for each checkpoint. Work was split as evenly as possible per checkpoint among our three team members, also depending on our work schedules with other assignments. We would each develop expertise in certain areas and build on these areas as the weeks progressed. Some examples of areas of expertise are frontend, load store unit, branch predictors, the ROB, power optimizations, and critical path optimizations.

Our goals for this assignment were to design a processor that is as performant as possible, given the constraints of the assignment. Our performance metric was $PD^4$, where $P$ is power in milliwatts and $D$ is delay in milliseconds. We had to constrain our design to an area of $300,000\mu\text{m}^2$. Teams were ranked in their performance metric as a geometric mean across all testcases, and our goal was to place as high as we could given the time we had. Our goal as a team was to place on the podium, but our final rank was 5th place. Though the official rank is 5th, we believe 4th would have been more realistic for reasons explained in the Additional Observations section. We also achieved the highest IPC for any benchmark on compression (though this was partly due to our lower frequency).

Another one of our goals was to learn as much as we could about micro architecture, which involved implementing a large variety of advanced features. Our team implemented TAGE and attempted EBR, though these were not graded as they did not lint. We were all very motivated to learning about advanced features to learn how modern processors can perform operations so efficiently.

## III. Design Description

### A. Milestones

*1) Checkpoint 1:* For checkpoint 1, we designed the processor's high-level architecture and implemented the fetch stage of the processor and some utility modules. We used these modules to hook up the fetch stage to the burst memory model of the top-level CPU and verified that we were able to obtain instructions. In the diagram in figure 1, this corresponds to making the fetch stages (1 way) and connecting it to the icache (not pipelined at this time), which is connected to the DRAM using a cacheline adaptor. The fetch stage pulls instructions from this icache and puts them into the iqueue. The cacheline adaptor and the iqueue were tested in isolation in testbenches individually, with testcases for the fifo initially testing a basic single push/pop sequence, a test to check fifo behavior when we push it fills up, a test to check simutaneous push/pop behavior while FIFO is full (to see if it lets the operation proceed as expected), then two more tests to check behavior when we try to pop from an empty queue and also a simultaneous push pop when the fifo is empty (which is considered an invalid operation, in which case the fifo does nothing). For the cacheline adaptor, 256-bit data was sent on the burst memory side, and we analyze if the correct data shows up at the icache side at the right time. This allowed us to easily isolate faults while connecting different modules together.

*2) Checkpoint 2:* For checkpoint 2, our requirement was a fully-functional CPU aside from branches and load/store. We implemented a module containing the RAT and RRAT, which we combined for simplicity of use. We implemented decode logic for the opcodes imm, reg, lui, and auipc, with their corresponding accesses to RAT, ROB, and the register file. We implemented a merged register file design, which starts with all architectural registers pointing at hard-wired physical register 0, which is all zeroes. We also implemented the logic for the ROB, designing it to be N-wide for superscalar considerations later on. We implemented a bit-vector-based allocation set for allocating and freeing physical registers. Whenever a physical register was requested, the bit-vector would be scanned for the first free register. When a register was retired by the ROB, a register would be allocated to the retirement allocation set. When an architectural register mapping was overridden by a retirement, that register would be freed from both the allocation set and the retirement allocation set. Finally, we also implemented the reservation stations and the functional units for all the ALU, MUL, and div operations. In this checkpoint, we implemented most of the modules of the backend of the processor as shown in figure 1 (for a 1-way processor). For testing, we first wrote individual test benches to test the functional units in isolation and then connected the reservation stations to these testbenches, which helped us test of the reservation stations worked together with the functional units as expected. We then connected the pairing of the reservation station with the functional units to decode and the CDB to check if there were any timing mismatches in the expected behavior for each module. Most of the errors in this checkpoint were found during the integration of several modules, and waveform debugging in Verdi was mainly used to solve these. Some notable bugs include elements being issued from the reservation station but remaining in the reservation station, and instructions receiving ready signals too late.

*3) Checkpoint 3:* For checkpoint 3, we added the control flow and memory subsystems in figure 1 in our processor. We support branches by adding a branch register file (BRF) and a branch allocation set, and we also added a reservation station for branches, along with a branch functional unit to calculate the branch outcome. Changes were made to fetch and decode to handle PC transactions. In our fetch, a mini decode stage was installed that would check if the instruction was a control flow instruction. If it were a JAL, we would simply increment the PC with an adder we maintained in fetch. For BRs, we would use always taken and use the same adder to calculate the new address. JALRs would use a RAS to predict their target address. For BRs, we would store the "nonpredicted target," which in CP3's case would be PC+4, in the branch register file. This could then be read in the ROB in the case that the branch prediction was incorrect to restore the correct value of the PC. JALR's would do a similar thing, where the predicted target was stored. Then, in the branch functional unit, they could be compared to the true target. The true target would also be written into the same slot of the branch register file. If there was a mismatch, a flush order was preserved in the ROB to execute on commit using the address in the branch register file.

We also implemented a naive load/store system. The system would simply fire load and store requests when the corresponding load and store was at the head of the ROB. We then replaced it with a split LSQ design which allowed out of order loads and store forwarding. Load/store operations were also integrated at the decode stage, and handling logic for

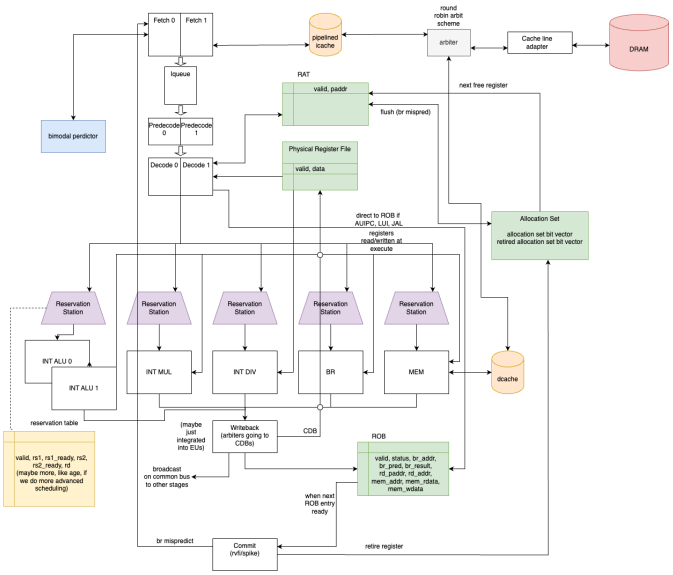


Fig. 1. Final Competition Diagram

stores was added at ROB commit. We also added a next-line prefetcher.

The final design we submitted in CP3 passed all provided tests and the random testbench, met timing at 100MHz, and we also saw a significant increase in IPC across all the benchmarks using the split LSQ. We first tested the LSQ in isolation in a testbench by testing a basic load and a basic store, then running a test where there is a load-store dependency to verify behavior, and also running tests to test full and partial forwarding. We also confirmed behavior when the load and store queues were full, and if the behavior was correct on a CDB stall. To test individual subsystems after connecting them to the top level, we wrote small test programs which consisted of only normal alu instructions along with all the possible scenarios of branch instruction to confirm functional correctness in all these cases, and we did the same for both the naive implementation of the memory subsystem and the integrated LSQ which helped us catch many corner cases which arise in the implementation of the LSQ. We then moved on to the full test bench suite provided and did manual debugging for bugs that had been a result of integration.

### *B. Advanced Features*

#### *1) Split Load-Store Queue Unit:*

*a) Implementation:* The LSQ is built around two circular buffers, the load queue and the store queue, which are managed by head-and-tail pointers and entry counters. Each entry in these buffers tracks the state of the memory operation. To break any critical paths from decode to the LSQ, new memory operations are registered before inserting them into the queues. This latch can be updated by the CDB if operands become available when the instruction is registered, by checking the physical register address of the data being broadcast on the CDB. Once the base address operands are available, the effective memory address is computed, and a read/write mask is generated. For stores, the write data is prepared using this mask. Then, before the head of the load queue issues a memory request, it checks for potential data forwarding from older stores in the store queue by checking its dependency pointer, which is a pointer to track all the stores issued before it, stored in the queue. If a store's address (word-aligned) matches the load's, and their byte masks overlap at least partially, data is forwarded from the store queue entry to the load queue entry. If all the required bytes are forwarded, then the loads are complete without accessing memory. Memory access is arbitrated, generally prioritizing loads if both can be issued. A load at the head of the load queue issues a memory request

TABLE I
PERFORMANCE STATISTICS

| coremark | iq emty | alu full | lsq full | rob full | iq full | br acc |
|---|---|---|---|---|---|---|
| base | 16.8% | 10.3% | 0.02% | 26.6% | 10.2% | 66% |
| base l | 21.8% | 6.9% | 0.0% | 12.4% | 10.2% | 66% |
| base ls | 13.3% | 12.0% | 7.7% | 1.2% | 7.1% | 66% |
| base lsg | 8.7% | 14.7% | 13.2% | 2.5% | 10.4% | 92% |

Note:
Base: the cp3 design.
Base l: the cp3 design with a split lsq.
Base ls: the superscalar design with a split lsq.
Base lsg: the superscalar design with a split lsq and G-share branch predictor.

iq emty: % of cycles the instruction queue spends empty
iq full: % of cycles the instruction queue spends full
alu full: % of cycles the alu reservation station spends empty
lsq full: % of cycles the lsq spends empty
rob full: % of cycles the ROB spends empty
be acc: the accuracy of the branch predictor

All tests were done at 100Mhz.

TABLE II
PERFORMANCE STATISTICS

| | average power | area |
|---|---|---|
| base | 8.3mW | $165048\mu m^2$ |
| base lsq | 8.5mW | $192878\mu m^2$ |
| base lsq sup | 8.7mW | $211793\mu m^2$ |
| base lsq sup g | 10.4mW | $241135\mu m^2$ |

Note:
Base: the cp3 design.
Base l: the cp3 design with a split lsq.
Base ls: the superscalar design with a split lsq.
Base lsg: the superscalar design with a split lsq and G-share branch predictor.

All tests were done at 100Mhz.

TABLE III
LOAD STORE QUEUE FORWARDING

| testcase | successful store forwards | forwarding attempts |
|---|---|---|
| coremark | 1173 | 245294 |
| aes_sha | 244 | 972142 |
| fft | 3842 | 918083 |
| mergesort | 6 | 350224 |
| compression | 0 | 322742 |

if it still requires data from memory after forwarding, and the memory interface is not currently occupied by another load or store. Upon receiving the response for a memory request, the received data from the memory is merged with any data previously obtained through forwarding. A store sends a signal to the ROB when its address and data are calculated, sending its ROB address and telling the ROB that the store is ready to be committed. Later, when the ROB commits the store at that ROB address, it sends a signal to the load store queue, as a store can only issue to memory after being committed. After receiving a response from memory, the store completes, the store queue head pointer advances, and all dependency pointers in the load queue are adjusted accordingly. When a load operation at the head of the load queue has completed, the CDB bus for load stores is driven with the load's final data and associated tags, and after this happens, the load queue head pointer finally advances. On a flush signal, the load queue is cleared entirely, and the store queue clears entries that are not committed, preserving committed stores as those are yet to access memory. Memory operations that are running during the flush are allowed to finish.

*b) Performance:* As one can see in Figure 3, migrating from the monolithic in-order memory queue to the split LSQ immediately raised the average IPC across benchmarks by roughly 35%. The difference was even more noticeable in memory-intensive benchmarks such as FFT, where the gain in IPC was 55%. This gain is mainly due to the early out-of-order issuing of independent loads. The time the ROB was full also dropped by 14% since the average number of cycles spent by a load operation before being issued to the dcache decreased, thereby decreasing the time the ROB had to spend waiting for memory operations to complete. Also, there were fewer decode stalls due to the queue for memory operations filling up since loads and stores were now stored in different queues and the ROB was full less often, so the time the iqueue spent empty also went up by 5%. A surprising observation we saw was that after implementing forwarding, there was not a huge increase in IPC. On profiling the successful store forwards vs the forwarding attempts (table 3), we found only 0.4% of loads were actually successfully forwarded on coremark, while there was no forwarding on programs like compression. The addition of the LSQ caused a 0.2mW increase in power and a 27,830 $\mu m^2$ increase in area. However, a lot of this area increase was due to the forwarding logic exponentially increasing the number of connections in the load store queue, which provided a very small gain in IPC, at least on the provided benchmarks, so this feature could have been removed. This was the major tradeoff of the LSQ. The load store queue has the best performance on workloads where there are mixed ALU and memory operations with short dependency chains, whose effect is further amplified by the super-scalar frontend as loads enjoy double the issue bandwidth. It also does well in applications where there is pointer chasing happening, with frequent pointer updates immediately followed by dereferencing the pointer, which would result in store to load forwarding. The split LSQ would perform poorly

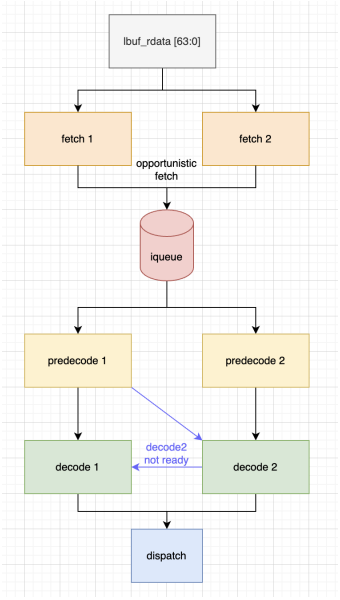


Fig. 2. Superscalar frontend

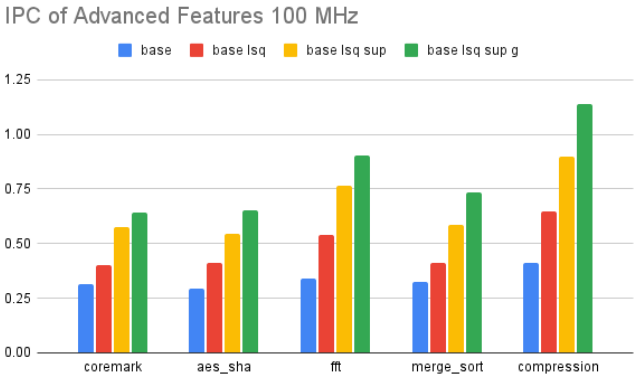


Fig. 3. Performance of Advanced Features.
Note: Base represents the cp3 design.
Base lsq represents the cp3 design with a split lsq.
Base lsq sup represents the superscalar design with a split lsq.
Base lsq sup g represents the superscalar design with a split lsq and G-share branch predictor.

on memcpy-like loops where writes are dominant, since the stores would still serialize at commit, so this would reduce the queue back to an in-order buffer (with a smaller queue size than the naive implementation).

*2) 2-way Superscalar:*

*a) Implementation:* The implementation of Superscalar involved updating almost every unit in our processor to support 2-way worth of instructions. We had settled on two ways, given the diminishing returns of multiple ways and the added complexity that comes with a parametrized way count.

During initial development, many modules had been parameterized such that the only modules that required major changes were the fetch, issue queue, decode, and ROB. Of these, the ROB and the instruction queue were the simplest,

each being modified to support two pops and two pushes to their internal FIFOs. For the ROB, it was important to enforce order for things such as flush correctness and store correctness. This was simply done by blocking a second commit if the first element did not commit.

Special consideration had to be made for the fetch given the problem of cacheline boundaries and timing. The final design consists of a pipelined imem cache with the ability to return two adjacent words. If the address requested was at the boundary of a cacheline, the second word would be invalidated. Our team had tried using a banked cache, which could pull two words from adjacent cachelines, but we had found this to have little effect. This was due to the fact that the final design was not limited by the frontend. The fetch module would opportunistically pull PC and PC+4 from memory. The two instructions would be decoded to check if they were a BR or JAL instruction. In that case, the target address was calculated with two adders. This, combined with the branch predictor, was used to determine the next PC. If the first way was a jumping instruction (either a JAL or a BR predicted to branch), the instruction in the second way would not be pushed. In all other cases, both instructions would be pushed in order.

Timing had been a challenge for fetch. Previous versions had only one adder with arbitration for which word received calculations, but we had found this to be too hard on the crit path. Our team also found that accessing the branch predictor in the same cycle was to costly, and so it was modified to access in the previous cycle.

Decode also requires special considerations given the multiple elements it must dispatch. Our team had known that decode was on the critpath, and so that had to also be considered. Due to these two reasons, decode was split into two: a predecode, where all control signals were calculated, and a decode, where actions were executed.

The decode stage itself would pull one or two instructions, depending on its capacity. The second instruction could only be dispatched if the first could be dispatched to retain order, as seen in Figure 2. If two instructions were going to the same reservation station, only the eldest instruction would be dispatched. The exception to this was ALU, since our processor supported two ALUs. If only one instruction had been dispatched that cycle, only one new instruction could enter the decode stage, with the undispatched instruction being promoted to the eldest instruction. Our team knew of the optimization where instructions could be allocated in the ROB when they entered decode, and thus the second instruction could dispatch without the first, but we had no time to test this optimization.

*b) Performance:* As one can see in Figure 3, the jump to superscalar improved the IPCs of all benchmarks by around 50%. This is not a surprise given that the base design + split load store queue suffered with high rob occupancy and low instruction queue occupancy. By going superscalar, both bottlenecks were resolved. With the ROB able to commit two instructions at once, the time the ROB spends full dropped roughly 10%. The instruction queue was also empty for less of the time given the wider fetch stage. The biggest trade-off with superscalar is the area and the power. Area increased by roughly 20000 $\mu m^2$ and about .2mW. That said, given the lenient area and power restrictions, our team found this to be worth it. Superscalar performs best on workloads with a lot of instruction-level parallelism. This is due to the fact that high ILP(instruction level parallelism or how parallelizable a program is) workloads will have fewer dependencies to clog the frontend. This was true especially for workloads like compression, where IPC gain was large. Superscalar would perform poorly on an inherently serial program where data cannot be reordered, so issue width collapses to one instruction per cycle. In this case, all the extra circuitry needed for superscalar would be sitting idle, wasting power and adding to area.

*3) GShare Branch Predictor:*

*a) Implementation:* The Gshare branch predictor provides two independent branch predictions per clock cycle, using a Pattern History Table (PHT) and a Global History Register (GHR) to make and update predictions. The PHT is implemented using flip flops and has 1024 entries (for the advanced features branch, for power consumption reasons, in the competition, a different predictor was used). Each entry in the PHT is a 2-bit saturating counter, storing the prediction state for a specific branch history pattern. These states are initialized in the weakly taken state on a reset before any history is learned, and the GHR is cleared to all zeros on a reset. The size of the GHR is set to the number of bits required to index into the PHT. The GHR stores the outcomes(taken or not taken) of the most recent globally resolved branches. For each prediction request made by the fetch stage, which provides pc1 and pc2, an index into the PHT is calculated by XORing a portion of the program counter(PC) with the current value of the GHR. The bits chosen for the PC are from bit 2 to GHR_SIZE+1 because the instructions are word-aligned so the bottom two bits would be 0 and hence not useful while generating the index. If the PHT entry at the index corresponding to pc1 is weakly taken or strongly taken, the pred_taken1 signal to fetch is asserted, and similarly, the pred_taken2 signal is asserted if the entry at the index corresponding to pc2 is weakly or strongly taken. When a branch is resolved in the branch functional unit, it asserts a signal going to the Gshare predictor and sends it the PC of the update. The PHT index corresponding to the branch being updated is calculated using the update PC and the GHR value at the time of prediction. The two bit saturating counter at the generated PHT index is updated based on the actual branch outcome, where it is incremented if the branch was taken (unless the counter was already strongly taken, in which case it remains strongly taken), and it is decremented if the branch was not taken (unless the counter was already strongly not taken, in which case it remains strongly not taken). The GHR is also updated by shifting its current value to the left by one bit and inserting the actual branch outcome into the least significant bit, ensuring that the GHR reflects the history

of the most recent branches.

*b) Performance:* As one can see in Figure 3, inserting the Gshare branch predictor on top of the superscalar + split LSQ pushed IPC up by a further 15-20% across the benchmarks with control-heavy workloads like compression, gaining as much as a 27% increase in IPC. This improvement comes straight from a drop in global-branch misprediction rate, which, as shown in table 1, goes from a 66% accuracy using static taken, to a 92% accuracy using Gshare. We also tried using a TAGE branch predictor and found that the accuracy increase of 2% did not justify the huge increase in power and area (since the geometric history tables were only being used 5% of the time over the base predictor for TAGE). This decreased the time the iqueue spent being empty by 5% since the effect of clearing out the iqueue on a mispredict flush was significantly reduced. The trade-off was mainly the power and area. Power rose by about 2.2mW and area rose by almost 30,000 $\mu m^2$. This was mainly due to the PHT being implemented as flip-flops and being size 1024, which significantly increased the power on the clock network. However, this was deemed to be a necessary cost since the main bottleneck in increasing IPC at that point was the high number of flushes. An SRAM-based implementation would have helped partly remedy this, but we faced errors from the autograder while trying to add the SRAM, which discouraged us from exploring this further. Gshare works best on branch-dense, high ILP code with irregular control flow where the pattern of consecutive branch outcomes is deterministic, which explains the relatively higher increase in IPC in tests like aes_sha and mergesort. Gshare performs poorly on workloads where branches depend on local rather than global history since the entire GHR is XORed with PC, and the bits that represent other branches act like random noise.

TABLE IV
PERFORMANCE STATISTICS OF AGE ORDER ISSUE

| | age order IPC | no age order IPC | Concurrent ready |
|---|---|---|---|
| mergesort | 0.325784 | 0.320803 | 45249 |

Note: Concurrent ready signifies the number of cycles where a reservation station had multiple elements ready.

*4) Age-ordered Issue Scheduling:*

*a) Implementation:* The reservation stations are all designed to track operand availability for the instructions and issue instructions to their functional units based on their "age", which refers to their dispatch order. The main data structures in the reservation stations are the rs_entries array, where each entry holds a valid bit to see if the entry is occupied and instruction-specific data like physical register addresses and valid bits for source operands. There is also a ready array that indicates which entries are ready to issue, that is, which entries and valid and have both operands available. Along with this, an empty array indicates which entries are available to accept new instructions, with an entry being considered empty is it is invalid or if its instruction is being issued to the functional unit. The reservation station monitors all the CDBs to update the availability of source operands, looking for a match between the physical register address of the result on the CDB and the reservation station entries, and setting the source operand validity bit when it finds a match. The age-ordered issue policy iterates through the reservation station entries to identify ready instructions, then selects the earliest ready instruction for issue, based on its position in the reservation station array (lower index means older instruction). If multiple instructions are ready, the one with the smallest index is selected. To implement the actual age-ordering mechanism there is a fall array where fall[i] is asserted if the entry at i+1 is empty or can also fall through, allowing older instructions to shift forward to fill empty slots, preserving the age-ordered issue policy even when instructions complete out of order. To terminate this falling sequence, the last fall signal depends directly on the last empty signal.

*b) Performance:* Age order issuing was the most difficult to benchmark, given that we had developed it in CP2 and so had no baseline to compare to. That said, with some slight modifications to the current module, one could make a youngest instruction prioritizing module. When comparing this with the classic age order issuing, we can see the difference in performance. This was done on mergesort as this best displayed the effect. As seen in table 4, the IPC of the age ordered module is roughly 0.005 greater. The effect itself is not larger, which is most likely due to the fact that the reservation stations themselves are not deep, around 4 for the tests. That said, it did result in a performance gain. To further illustrate this, our team logged the number of cycles where multiple instructions were ready to be issued, a circumstance where age-order issuing is helpful. One can see this happens often in about 45 thousand cycles.

The tradeoffs for age-order issuing are plentiful. Given the fact that our age-order issue was collapsing, lots of shifts occurred, which consumed a lot of power. This was manageable in our design given the small sizes, but deeper reservations stations would suffer more from this problem. Another tradeoff is the long critical paths that result from requiring priority. The third tradeoff is that age-order issuing may not be the most optimal method of issuing instructions. Theoretically, the most pertinent instruction to issue is the one with the most dependencies, and age-order is only an imperfect heuristic for this. In general, age-order issuing is better for workloads with low ILP, as it theoretically helps resolve dependencies quicker. This makes it worse in situations where there with lots of parallelism as the added power and critical path could be used to increase performance in other areas.

## IV. ADDITIONAL OBSERVATIONS

One observation our team has is about our placement in the competition. The final standings had put us at 5th, but we believe we could have placed 4th. The day before the last day, our team had submitted the run that landed us in 5th place. During the last day our team spent a whole 24 hours trying to optimize the frequency of our processor. The issue

was, to meet these frequencies, shortcuts had to be taken in the load store queue, tanking our IPC. This, in addition to the increased power, had made our final score the same as the day before. That said, we could have simply pushed our frequency from our initial submission to 540, which we believe would have pushed us to 4th place. The lesson learned here was to do the simple, easy fixes first, test them, then take on larger improvements.

During our fruitless optimization, we tried to build a large number of additional features that didn't end up being useful for the processor and added, not reduced, from the critical path. First, we implemented a banked instruction cache, which made a negligible impact on our performance due to our efficient pipeline cache and opportunistic fetch mechanism. We also attempted a 4-wide fetch, but this led to critical paths accumulating in the fetch stage, slightly increasing our IPC in exchange for a huge amount of frequency. We tried adding pipeline stages and buffers to the LSQ unit, but ultimately hurt the performance of our LSQ in an attempt to reduce the critical path only slightly. We even tried unusual techniques like non-power-of-2 sizes for the LSQ and unbalanced LSQ, and while this was promising for the critical path, lowered our IPC on load-store heavy testcases by up to 0.1.

After submission, we recognized that the issue with our final optimization attempt were several structural issues that appeared early on that we didn't address. For one, though we had implemented a complex split LSQ with some benefits, the entire module was very unruly. The Verilog was not written with the cleanest RTL, and the complexity resulting from forwarding logic made critical paths difficult. If we could return to the drawing board, opting for simpler, performant modules rather than modules that met advanced features points would have been a better move. After we had made the split LSQ, we did not want to rewrite it, even though much of the forwarding was not often used. The other issue our team faced was with our caches. At the end, our most advanced cache was a pipelined instruction cache. We believe if we had invested more time at the beginning into a proper pipelined cache or a nonblocking cache, our LSQ would perform better without requiring complex forwarding logic. The lesson learned for next time is to be aggressive in seeking simplicity. When a module gets too long or bloated, we ought to consider a rewrite.

## V. Contributions

Below are the contributions of each team member:
Jefferson Zhang: fetch, decode, branch functional unit, 2-wide decode, critical path cutting, debugging
Kai Karadi: allocation set, physical register file, ROB, 2-wide fetch, 2-wide ROB, pipelined imem cache
Vaibhav Mattoo: reservation stations, ALU/MUL/DIV functional units, Split LSQ, gshare, TAGE, memory arbiter, cdb

## VI. Conclusion

In conclusion, our team believes we met many of the goals we set out to achieve in the beginning. Though we were hoping for a podium spot, we are still proud of our 5th position. The last two weeks of the MP had our whole team living in DCL, with many sleepless nights, and we believe we put in everything we had. Some of the issues we faced were more long-term and required more planning from the beginning. Though we may not have met the podium goal, we did learn a lot about microarchitecture and, perhaps more importantly, the importance of RTL style. We attempted and completed many advanced features, and we also learned the problems that come with unclean RTL. In the end, we produced a performant RISC-V CPU that, while clocked at a lower frequency due to long critical paths, could outperform other cores at the same frequency in IPC due to some of our design decisions and advanced features, which were the core of this project, and came out successful.

## Acknowledgment

We thank our course assistant mentor, Shiv Gohil, for his support and relentless discussion over advanced features throughout this project. We thank our TAs, Pradyun Narkadamilli and Stanley Wu, for benchmarking & toolchain resources. Finally, we thank our professor for the course, Dong Kai Wang.