ECE 411 Final Project Report: mp_OoO

Team OoOps

Abhi Alavilli
Department of Electrical and Computer Engineering
University of Illinois,
Urbana-Champaign
Urbana, IL, USA
abhia3@illinois.edu

Alexander Gallagher
Department of Electrical and Computer Engineering
University of Illinois,
Urbana-Champaign
Urbana, IL, USA
alexg5@illinois.edu

Pratyay Gopal Reddy Rudravaram
Department of Electrical and Computer Engineering
University of Illinois,
Urbana-Champaign
Urbana, IL, USA
pratyay2@illinois.edu

***Introduction*: This report details our group's implementation of an out-of-order RISC-V 32 IM processor "OoOps" in RTL and our findings while optimizing its performance. Being able to execute instructions out of program order, while maintaining true dependencies for correct execution, allows the core to take advantage of instruction-level parallelism (ILP), which is an inherent exploitable property of most computer programs.**

This paper details the high-level (and some granular) design details for our core, as well as observations and tuning we conducted from benchmarking it across various programs.

I. Project Overview (*Advanced Features*)

For this project, due to the soft 300,000 µm^2 area limit we decided to implement a dual-issue core to increase throughput without hitting diminishing returns. We suspected that the increased pipeline throughput would provide an edge in the competition, as well as playing nicely with all of the different memory optimizations we had planned. Alex had experience with caches from a previous class, and was able to get a lot of memory-related work done before we even had a working core. As for the division of work, we created an initial division of labor - Pratyay on the execute units, Abhi on the renaming structures and Alex on the issue queue, caches, and dual-port FIFO, which was an essential part of our 2-way superscalar design. Later on, we ended up all working on several modules (and adding several more) during integration, and many were rewritten several times.

II. Design Description

A. Overview

The OoOps processor is a two way superscalar, speculative, out-of-order core following the RISC-V 32 bit integer ISA with the Multiplication extension. At a high level, our core was split into six pipeline stages:

- Fetch
- Decode
- Rename
- Dispatch
- Execute
- WriteBack/Commit

Coming from a project where we each wrote a one-way in-order pipelined core, we found it less complicated to build the core keeping everything in both ways in order until dispatch, where instructions sit in reservation stations until their operands are ready. Since we used an explicit renaming scheme, we needed ERR-specific structures (RAT, RRF, Freelist) as well as a reorder buffer to ensure commit order was maintained correctly. The execute units were organized as ALUs, an AGU, a MUL/DIV unit, and the Load/Store queue for memory operations. Each execute unit has a bus line (Common Data Bus) going into the register file to write results . This design allowed us to eliminate the overhead of bus arbitration by simply checking each line for published results in the register file. It also made it easier to add more functional units (our final design incorporated two ALUs) later on. The reorder buffer was built to commit two instructions at a time to RVFI, which was verified with a Spike golden model.

B. Milestones

(i) Checkpoint 1

For the first checkpoint, we began laying the foundation of our core - we built a simple fetch unit, a dual-ported FIFO to make it easy to move to superscalar, and wrote the line buffer for our cache. We also made an arbiter (later rewritten) for DRAM, since the memory model we were provided was single-ported. This set us up for the rest of the project with a basic instruction queue.

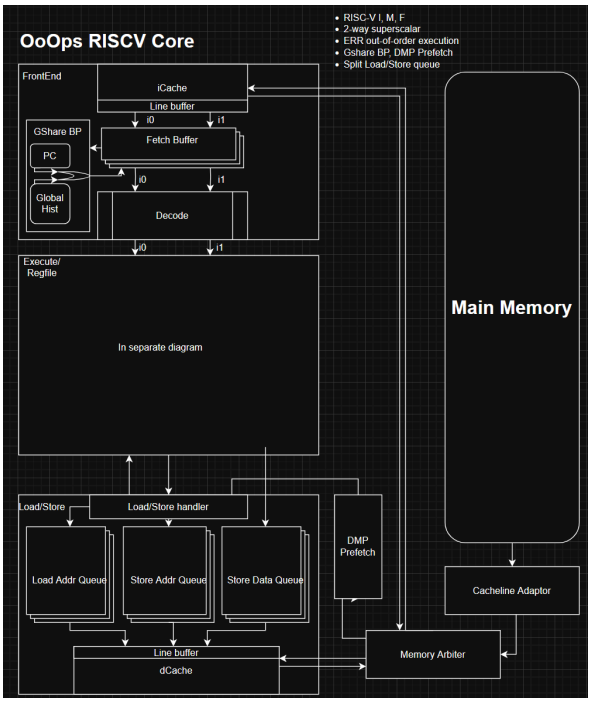

Fig. 1. Initial high-level draft of OoOps core.

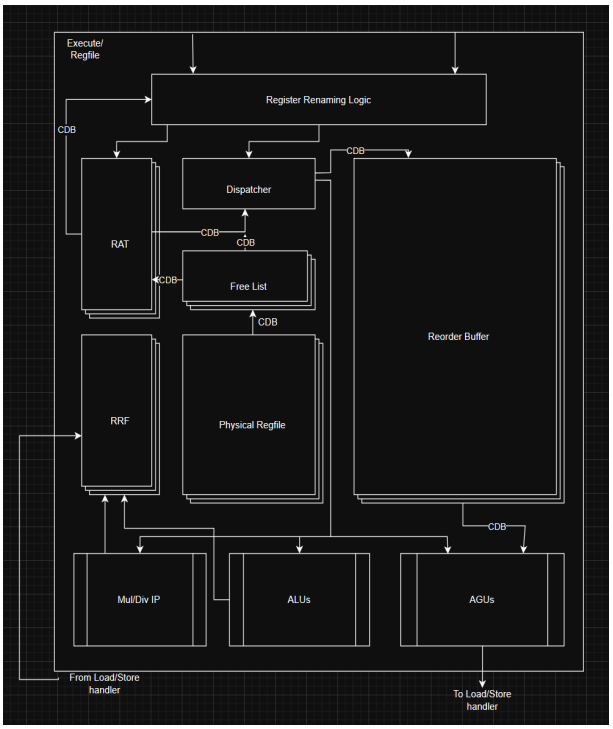

Fig. 2. Rename/Execute stage diagram.

We drew up this simpler diagram of what the core would look like, and this was used as a high-level reference for integration. As seen in the first diagram, we at first considered the F (float) extension to the ISA before we learned that it would not be worth additional advanced features points.

(ii) Checkpoint 2

In this checkpoint, we aimed to process ALU instructions with out-of-order capability. We were not able to meet the deadline for the checkpoint, but we made progress on the rename structures while Alex wrote the prefetchers, cache, and the split load-store queue. Pratyay integrated Synopsys IPs for the MUL/DIV unit into the core. At this point, we began integrating components and wanted the way modules worked to be clear to everyone, regardless of who wrote them. A major source of error and confusion was the rename stage, and the following diagram helped to resolve that.

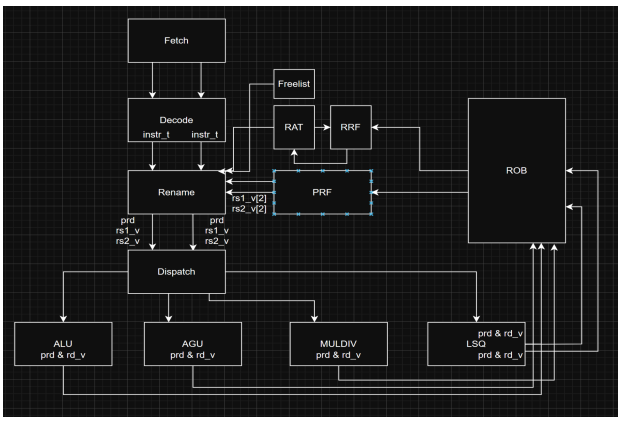

Fig. 3. Checkpoint 2 Rename stage in detail.

(iii) Checkpoint 3

During checkpoint 3, we had a lot of trouble with integration - we found several components we had written that were not written to support dual issue or were not verified with randomized and targeted test vectors. We were able to finally put everything together and get a core that would compile in simulation, but would cause a Spike mismatch on the first commit. After spending a few days debugging and getting to a few hundred instructions, our team decided to take a step back and rewrite some of our ERR modules. We verified these independently before we began integrating again, creating fewer hiccups and bugs.

C. Advanced Design Features

The two weeks remaining were what we had to work with to get our processor to work. Although we were behind on previous checkpoints, something that worked towards our advantage was that most of our advanced features had been implemented while we developed the core. As a result, getting it to work with these components gave us a very performant processor from the start. During this time, we rewrote the rename modules, created a hard ordering down to the dispatch stage to simplify logic, incorporated GShare branch prediction, greatly improving on the saturating counter we had been using prior, parameterizable functional units, and an interesting way to make the core one-way by feeding no-ops into one port of the instruction queue. Below each optimization is discussed individually.

(i) Cache features

Both the instruction and data caches were implemented with three ports - one for the prefetcher, and two ports for the two ways of the superscalar core. By allowing two ports to simultaneously access the cache, we could take advantage of the pipelined cache system, allowing for maximizing throughput of the SRAMs without increasing the number of SRAM ports to support it. Additionally, we utilized a 64 Byte cacheline size, and through the parametrization of the cache, we doubled the total size of the iCache relative to the dCache for performance reasons (8KB vs 4KB). Additionally, we implemented a linebuffer for same-cycle responses and to support up to 2 memory operations per cycle.

To test the caches, we used Coremark to see how different set and way counts performed. We found that moving from a 4-way to 8-way cache benefited both the instruction and data cache, giving us a 4% and 1.5% improvement on runtime respectively. For this reason, we chose to double the size of the iCache rather than the dCache.

(ii) Cache prefetchers

Instruction Cache: We used a simple stride (which ended up being used as a next-line) prefetcher for the instruction cache, and triggered it off of a read or write being halfway through the cache line in the line buffer. We measured a 0.4% improvement in program delay for CoreMark on the instruction cache, and no measurable improvement when tested on the data cache. We attribute

this small uplift primarily to the already well-utilized memory systems implemented through the current cache implementation and the line buffer.

Data Cache: Here, we used a DMP (Dynamic Memory-dependent Prefetcher) to look for "pointers": each line fetched from DRAM into the data cache is scanned for pointers to prefetch. This is accomplished by checking if any of the words are within a certain window of the PC, which is fed into the prefetcher from the core. This was also only using one of the PCs in the superscalar core, however. Unfortunately, with our implementation, we were unable to get a measurable performance benefit, likely due to the lack of pointer fetching in the target programs.

(iii) Superscalar

From the start, we designed our core to be superscalar. Though this would mean we were blowing up our area usage, we ended up being just below the 300k μm^2 maximum, and reached about 291k. We did not consider having more than two ways due to both design complications and the penalty for exceeding the area limitation.

As for testing, we never tried synthesizing the core as one-way, and hence were unable to extract a baseline area metric to compare it to. This is because we tested the core as one-way simply by sending no-ops into the second way.

On CoreMark-IM, (a math heavy program) we saw an IPC of 0.557850 for a 1 way, 2 ALU core and IPC of 0.4680 for a 2 way, 1 ALU core. This showed us that the benefit of going superscalar may not have been as much as we had hoped for, though we did see more notable performance improvements on benchmarks like aes_sha, which have simpler control flow and higher ILP.

(iv) Load-Store Queue

The load store queue was one of the more advanced modules in the processor, and we chose to add this since we knew memory operations would be a likely bottleneck on the core. This also played well with the cache optimizations, which meant we had a fast memory system to support the core.

Our load-store queue incorporates out-of-order loads with respect to stores. First, prior to a load or store entering the queue, it identifies dependencies existing inside of the current store queue to forward necessary data. For any load data that only partially exists inside of the store queue, the load identifies the current index of the store in the FIFO to ensure that that store completes prior to the load. Additionally, the split LSQ forwards stores that have been committed but have not been pushed to the cache. Compared to our simple, single-port LSQ with in-order execution, we obtained a 5.7% increase in performance in memory-heavy benchmarks (AES_SHAH) but slight performance regressions in other benchmarks (Coremark) due to the increased latency of testing memory dependencies prior to inserting loads into the FIFO.

(v) GShare (Branch prediction)

Our branch prediction was done in the decode stage, and we used a Global History register and a Pattern History table with the XOR hash, as is commonly used, to generate a prediction. We had an 8-entry PHT to reduce area usage, and we found that this gave us a 91.6% prediction accuracy in the compression benchmark. Compared to the saturating counter (two-bit) predictor, this was a 29% accuracy improvement. It should also be noted that the tables were implemented using standard flip-flops instead of SRAM which would have saved on area, but our implementation required multi-ported PHT for our dual-commit processor, and used combinational reads to perform branch prediction in Decode. For this reason, we used a flip flop array.

III. Additional Observations

As we noted for several of the advanced features, our core benefited from memory optimizations throughout testing, and proved extremely performant on memory-bound programs. The superscalar aspect of the core was not as useful in general, but this may also have been bottlenecked by ALUs. As soon as we added the second ALU our IPC on Coremark increased by 19%, a significant improvement. Our core also suffered from high power usage across benchmarks, which we later realized was due to not clock gating our IPs, including the Cache SRAMs and the MULDIV IP.

IV. Contributions

Feature	Module Contributions		
	Alex	***Abhi***	***Pratyay***
Fetch Queue	✔	✔	
Cache	✔		
Prefetchers	✔		
Line Buffer	✔		✔
Decode	✔		✔
Rename			✔
Dispatch	✔		
DRAM Mux	✔		
LSQ	✔		
Issue Queues	✔		
ROB	✔	✔	
PRF	✔	✔	
RRF		✔	✔
Execute FUs			✔
MULDIV IP			✔
Rename	✔	✔	✔
RAT		✔	✔
Freelist	✔	✔	

Feature	Module Contributions		
	Alex	*Abhi*	*Pratyay*
Branch Predictors	✔		

a.

V. Conclusion

Overall, we successfully accomplished our goal of building a high-performance, superscalar RISC-V32IM core and performed well above the baseline core across benchmarks. Given more time, we would be able to improve on the power usage, functional unit/superscalar configurations, and perhaps timing. For our autograder runs, we hit a maximum frequency of 526 MHz, which was fair when compared to other superscalar cores. Our best-performing benchmark was the memory-bound compression, and worst-performing was, like most groups, CNN. Final benchmark results are shown below.

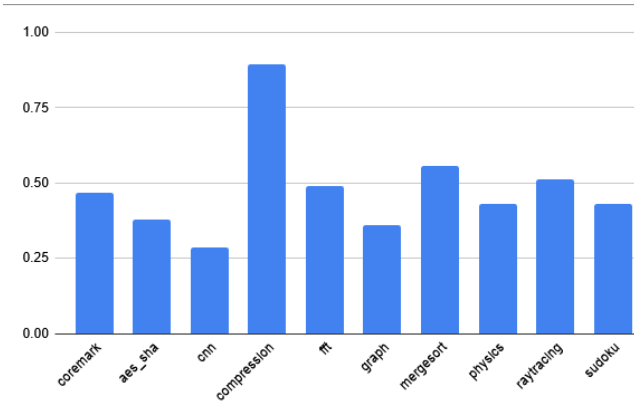

Fig. 4. IPC across provided and hidden benchmarks.

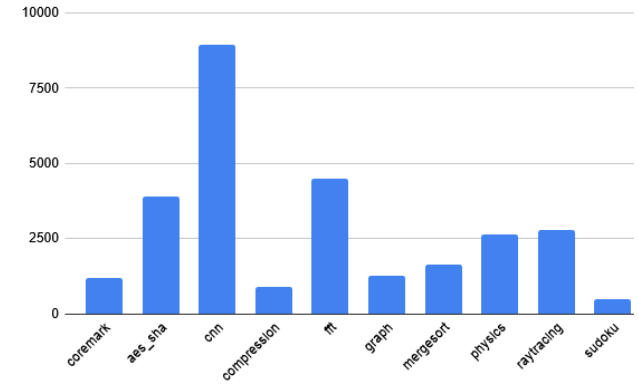

Fig. 5. Delay, microseconds, across benchmarks.

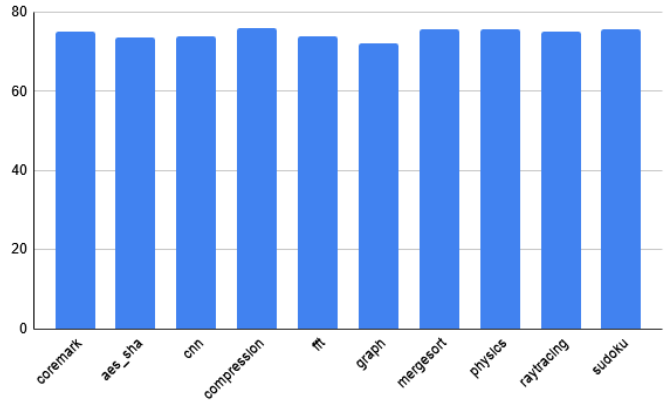

Fig. 6. Power, mW, across benchmarks.

Acknowledgments

We would like to acknowledge our mentor TA, Ethan Greenwald - meetings with him helped guide us through choosing which features to integrate, as well as lots of tips for specific implementation details that we found it hard to think through on our own. Additionally, this project was only possible through the support of our instructors, Professor Dong Kai Wang, Professor Rakesh Kumar and the rest of course staff that helped us throughout the semester. Development was supported using university provided Synopsys EDA tool licenses for simulation, synthesis and IPs, and all our work was done on Grainger Engineering EWS machines.