Group ball.sv Out-of-Order Processor Final Report

Introduction

The final project for ECE 411 revolved around planning, designing, and implementing an out-of-order (OoO) processor. This project encapsulated the core concepts of computer architecture by requiring us to not only research the intricacies of modern processor design but also served as a pinnacle moment for us as seniors in computer engineering to demonstrate the knowledge that we have gained throughout all the semesters we've studied here. For some background, an out-of-order processor is a processor that can execute instructions out of order relative to their explicit order inside of the assembly code, however, instructions are still 'committed' or finalized in the order in which they appear. Furthermore, we implemented some different optimizations for our processor such as early branch recovery, non-blocking cache, next-line prefetcher, and a 2-level branch predictor.

Project Overview

For our take on the OoO processor, we chose to implement the Explicit-Register-Renaming (ERR) design. This was because this design was arguably more flexible than the Tomasulo design and allowed for more room to make certain design choices such as choosing to implement our processor as superscalar. This meant that our processor would need to have a free list, a physical register file, a retired register file, and a register alias table. For our main optimizations, we chose to implement early branch recovery, non-blocking cache, and split load store queue. Later we implemented smaller optimizations such as a next-line prefetcher and a 2-level branch predictor. Our design goal for choosing these optimizations was basically to try and pick the ones that have the least tradeoffs (in our initial research and also from meetings with our mentor). Overall, the largest optimization that our processor was able to benefit from was early branch recovery and branch prediction, these boosted our performance significantly. The split load store queue and non-blocking cache were only marginally beneficial at best.

For organizational aspects of our project, we looked at the objectives for each checkpoint and then picked them apart, and set tasks amongst the team, then after which we would gather together and collectively work on integrating it all. Once we got to optimizations, we chose the optimizations we wanted to work on, and from there we worked on a separate branch that was the checkpoint 3 baseline and tried to get our optimization to work standalone. After this, we would try to integrate it into the main branch which would most of the time require two or more of us to be there in person and coordinate.

For administrative aspects of our project, we had lofty goals to continuously update documentation and the block diagram from checkpoint 1, however, we fell behind on those goals as we moved forward since as we neared the end, it became a race to try to get everything finished and integrated which left almost no free time to write good documentation. This may have hindered our debugging process, but overall since all of us were on the same page and we

almost always worked together in the lab, we never faced huge problems where someone's code was undebuggable since we couldn't understand what they were writing. We also ensured that all advanced features were a group decision, so if another teammate needed assistance all team members were properly up to date on the basic functionality of their advanced feature.

Checkpoint 1

In checkpoint one, we were tasked with organizing our block diagram, thoroughly implementing and verifying a base FIFO, integrating a cache line adapter with our pipelined cache, and having a functioning instruction fetch. When beginning the design of our ERR core, we wanted to simply grasp its basic functionality and begin thinking about possible optimizations. This led to a basic block diagram followed closely by information gathered from research papers and online sources. Once the block diagram was finished, we distributed tasks of verification, FIFO design, fetch design, and cache line adapter amongst teammates. For the adapter, we decided to just have base functionality without out-of-order loads.

To test the functionality, we made separate test benches for the parameterized queue, cache line adapter, and fetch unit. The test benches for the queue include trying to read and write when the queue is full and empty and try to read and write simultaneously. The test for the adapter is simply showing the resulting cache line is the same as expected. The test for the fetch unit can show that a sequence of instructions can be fetched correctly and read from the instruction queue.

Checkpoint 2

In checkpoint two, we were tasked with being able to support all register-register instructions, which included all of the M-extension instructions, and all register-immediate instructions. In this checkpoint, we needed to implement our first 2 functional units which were the ALU to compute add, subtract, shifting, and, or, and xor instructions. This was also where we needed to construct our first reservation station and common data bus since we were finally going to have our processor commit instructions off of the reorder bus. We followed the advice of our mentor and tried our best to make everything as parameterized as possible which would enable us to play with the sizes of everything to optimize for timing and area. This proved to be valuable insight and we would stress to future students to continue following this advice. Our mentor specifically told us: that since MP_OoO is a project with fast-paced deadlines, a good processor is a working one that has flexibility.

Another design decision we were faced with was how many common data buses (CDB) we should make in our processor. We had a choice between 1) making a unified CDB along with an arbiter in the case that two different functional units want to put their result on the CDB on the same cycle so that we can arbitrate one over another and prevent one instruction from getting lost and thus never being committed off the ROB and 2) making separate CDBs for each functional unit which would solve the arbitration issue at the cost of increased area and power and a

potential critical path. After further discussion with our mentor, we determined that we were willing to take the tradeoff since the performance hit that we would suffer as a result of stalling certain functional units (stalling and not committing to the ROB, which could in turn stall dispatch since the reservation station could be full) was too big to overlook. Our mentor also mentioned that the arbitration logic could potentially cause more bugs especially as we add more functional units later on (for memory instructions and control instructions).

The final design choice we faced was how granular we should make our functional units, the main one being multiply and divide. If we split them into separate units, we could potentially see an increase in performance since these two instruction types can execute in parallel at the cost of increased power and area. The other option would be to combine them into one reservation station so that we save area and power at the cost of potentially being able to run a multiply and divide together, however opting to only execute a single one. We ultimately chose the latter since we assumed that there wouldn't be a lot of multiply and divide instructions in the benchmarks they would run on our processor so splitting them into separate units would be a net negative for our processor. Additionally, the only way to generate a benefit would be to increase ROB size, something which greatly dimensioned overall performance when the area was a consideration.

The correctness of the design at this stage is justified by the random testbench. We randomly generated instructions using mp_verif's random rand_inst.svh and verified it using a modified covergroup, also from mp_verif. The main modifications we made to the rand_inst were that we constrained the random generation to only generate lui, reg-reg, and reg-imm instructions which we ran until timeout.

Checkpoint 3

In checkpoint three, the goal is to make the processor be able to execute the entirety of the RV32IM, including memory and control instructions. Overall, we chose to make our processor as simple as possible per the advice of our mentor since again, the best processor is to first get a working baseline and to go from there as opposed to thinking about all the advanced features and then getting overwhelmed.

The biggest challenge was integrating branches and memory instructions and making sure when they interact with each other, they work as intended. For us, we chose to make loads and stores into a single queue so that we force in-order-ness for all memory instructions to avoid memory dependencies. Furthermore, specifically for the load-store functional unit, we chose to respond the CDB a cycle later than the actual response due to our experience with MP_pipeline where if you broadcasted the data read from memory to the other reservation stations, you would almost certainly have a critical path here.

For branches, we also faced a design decision of whether or not to make branches occupy two different functional units or not since branches need both a comparator to compute the branch flag and an adder to compute the branch target address. We proposed to save area, we

could allow for the branch to also be dispatched to the ALU reservation station to potentially save an extra adder, but then we thought of a case where the ALU and the comparator may not be able to execute simultaneously on the same cycle, so we would need additional logic for the branch ALU to stall if the comparator isn't ready yet, etc. This proved to be a lot of work and we ultimately decided to give an extra adder to the control instruction function unit.

When integrating the two, we ultimately chose the easiest and simplest way just so we could get a working baseline processor first. We chose to have branches only flush when they're at the head of the ROB to ensure that all instructions before the branch have been committed and therefore we can safely just reassign all the queues and reservation stations to be empty as soon as we flush. Overall, these design choices allowed us to have a baseline processor that could execute the entirety of the instruction set we were expected to support, also at a frequency of 500 MHz.

After this checkpoint, our processor can handle all RV32IM instructions, so the correctness of the design can be justified by successfully completing the coremark and other complicated test cases.

Advanced Features

EBR

Doing branch recovery when the branch instruction is at the head of ROB is a straightforward approach that suffers from a large misprediction penalty. With early branch recovery, the processor can flush all of the incorrectly fetched instructions, therefore reducing the misprediction penalty and increasing the IPC. The main challenge to implementing this feature is to correctly tag each ongoing instruction so that their dependency on each ongoing control instruction can be known when the flushing happens.

To implement EBR, the control instructions should be dispatched to a control queue, so that they can be resolved in order. Inside the control queue, a control bitmap will keep track of which entry is storing an ongoing control instruction, and the purpose of the bitmap is to make later instructions know which control instructions they are dependent on. When dispatching each instruction, the bitmap of the control queue will be recorded in the reservation station it is dispatched into. When a control instruction is resolved and is found to be mispredicted, all the reservation stations with the corresponding bit of recorded bitmap to be high will be flushed. If it is found to be not mispredicted, the corresponding bit of recorded bitmap will be set to low for all reservation stations. The behavior of ROB on flush is to set the write pointer to be the ROB index of the control instruction plus one. To recover the RAT on flush, a copy of the RAT needs to be saved along with the valid map of RAT, when control instructions are decoded. The copied valid map also needs to be updated based on the content on the CDB when the content from CDB is from an instruction before the corresponding control instruction, and the CDB therefore

needs to contain the control bitmap of the instruction. The copied content will be copied back to the actual structure when flush happens and will be discarded if that control instruction is not mispredicted. Similar things need to be done with the freelist read pointer and store queue write pointer, so that these queue structures can be recovered by simply copying the recorded ones.

EBR never decreases IPC, because it reduces the misprediction penalty and the processor takes the same number of cycles in the case without misprediction. Implementation of EBR will increase the area of the processor, because the processor needs to keep several copies of RAT and its valid map, but it is acceptable since they are not large structures.

EBR will increase the performance of the processor a lot when misprediction happens frequently.

Local History Table & Pattern History Table & BTB

We implement a two level branch predictor (Local History Table & Pattern History Table) along with BTB. This feature will improve the performance a lot when there are many loops iterating a lot of times in the program. The extra logic to implement the branch predictor increases the critical path in our design, so the processor ends up having a deeper pipeline. If most control instructions are fetched only once and are mispredicted, then IPC will decrease a little.

The local history table and BTB are implemented as two-port SRAM, with 16 entries and each entry contains the pattern history, target pc when this branch takes, and upper bits of pc (pc[31:6]). It is direct mapped. The index that is going to be used should be pc[5:2]. The pattern history table is implemented as FFs, because it is a very small structure and this implementation can allow combinational reading. The pattern history table will take the pattern history reading from the local history table as index and it only stores 2-bit saturating counters, indicating strongly taken, weakly taken, weakly non-taken, strongly non-taken. In the fetch stage, imem address[5:2] is used as reading index to the local history table, and when imem response is high, upper bit of pc will be compared with reading data's pc upper bit. When one instruction is found to be missed in the local history table, it will be marked as weakly not taken. When a control instruction is resolved, the local history table and pattern history table will be updated, and if it is found to be mispredicted, then a flush will happen.

Table: Change in misprediction rate

Benchmark	Control inst #	Mispred # w/o predictor	Mispred # w/ predictor	Mispred rate w/o predictor	Mispred rate w/ predictor
coremark_im	60476	34822	9966	0.576	0.165
aes_sha	8136	7740	6652	0.951	0.818

compression	55477	40467	5021	0.729	0.091
fft	19964	19772	4508	0.990	0.226
mergesort	89107	45921	20962	0.515	0.235

Next Line Prefetch

We implemented next line prefetch by prefetching the next cache line into a buffer before the cache needs it. In our design, only Instruction cache has this feature because we believe fetching cache lines sequentially happens frequently when running a program. When a request is made by instruction cache to burst memory, the next line address will be recorded and an extra request on this next line address will be made when the burst memory is not busy with requests from neither caches, and the reading result will be stored in the buffer. If the next time the instruction cache makes a request on the recorded next line address, the content in the buffer will be delivered to the instruction cache, and a new next line address will be recorded.

Trade Off of this feature: Extra registers need to be added as a buffer to prefetched cache lines. In our design, the processor only prefetch one line at a time so 256-bit registers are required. The prefetched line cannot be directly put into the cache because it may not be useful until the cache actually makes a request on this address, so putting it into cache earlier can make the cache evict another useful line. This feature will not always lead to an increase in IPC, because fetching an extra possibly unuseful cache line may lead to a delay to the response of later requests. If the prefetched contents are never requested by the instruction cache next time, then IPC will increase.

Split LSQ

One of the issues we discussed for our checkpoint 3 implementation is that load instructions that are not dependent on older loads, older meaning that they appear before the load in the code, will not be able to be sent as a request down to the cache due to the store being in front of it. This kind of issue is known as Head-of-Line Blocking or HOL Blocking. For example, a store instruction that has either register depending on some time-consuming instruction such as a divide instruction will not run until it has this value. The load that comes after it will not be able to run as a result since memory instructions reside in a queue that forces in-order-ness. Our first thought was to convert the queue into a reservation station and then fire whatever instruction is ready, however, we would need some sort of dependency checking system that would not let a younger load fire until 1) all of its older stores have their addresses calculated so that memory dependencies may be checked. 2) if there are dependencies, ensure the load doesn't fire until all of these stores have left the store queue.

We chose to implement a pipeline-esque design for our memory instructions in order to enforce this order of events:

1) After dispatch, the memory instruction enters a structure known as the address reservation station
2) At the earliest, its register 1 is already valid and already leaves the address reservation station into either the load reservation station or the store queue.
3) From here:
   Loads will take another cycle to check for dependencies against the older stores using a bitmap system. Only after its dependencies have been checked, can it be sent as a request and at the earliest it will respond on the next cycle. Then finally, one cycle after the response, it goes onto the CDB and its ROB entry will be ready to commit. In total, the earliest loads can finish in is 5 cycles after they are dispatched. This is quite expensive as opposed to our earlier implementation where the fastest loads could potentially complete is in 3 cycles. The added cycle overhead due to having an address reservation station and a cycle to check for dependencies ultimately outweighed the benefit provided by the split LSQ, this cut back the performance of some of the benchmarks and only marginally improved other benchmarks.

   Stores will send as a request to the cache as soon as their data is ready and their entry is the head of the ROB. If memory responds next cycle, then the fastest a store can commit is in 4 cycles.

Overall these design choices helped us resolve this HOL blocking problem, but these situations weren't as common as anticipated and the performance gain that we should've gotten was overshadowed by the longer pipeline for loads. However, an upside to this is that deepening the pipeline allowed us to push our frequency up to 600 MHz and this in turn allowed us to shorten the cycle length of our multiply and divide IPs which allowed us to achieve better performance.

Non-Blocking Cache

Non-blocking cache is a straightforward idea that in practice becomes much more complicated. Essentially, the goal of a non-blocking cache is to prioritize servicing hits by ignoring other tasks a cache must do. On a cache miss, the cache must fetch from memory data and allocate the read data into its own SRAM array to service future hits, the issue is during this fetch we must stall all possible loads or stores. A non-blocking cache aims to solve this problem by allowing continued memory requests regardless of cache state. This idea ended up being difficult to implement and seemingly impossible to make it a feature that benefits.

For my implementation, I had a few goals. The first was to essentially have it so that the cache at minimum kept pipelined cache performance, that is with dependencies it ends up performing the same as a pipelined cache. Next, I wanted to make more non-blocking than required. This meant in addition to loads from main memories, I would be performing stores in the background. The main structure I used to achieve this was a queue, where on misses I simply

push to queue and attach any necessary dirty information. Then, the queue services these to bmem and in the midst allows hits to follow. In the case a write hit occurs on the dirty information I also had forwarding logic in place to update the information on the queue. Also to ensure the same waveform as pipelined cache, I had an early start feature in the situation where if the queue is empty to immediately issue.

For handling the dirties I wanted to better use the out-of-order load feature of bmem, so I pushed dirties onto the queue after a read response. This made it so I could execute consecutive misses more easily. Within the queue, I also had a ghost read pointer which essentially moved to service memory, and a read pointer which moved to service processor requests.

In addition to the cache HDL, there also was the arbiter and adapter which received changes. The adapter changed from sending a single dfp resp to sending a dfp_write and dfp_read to reduce logic in the queue itself. The arbiter changed from having states that serviced caches individually to having states that served the caches simultaneously, with priority on the instruction cache. This allowed instruction requests to be sent to memory out of order with the dcache.

Non-Blocking Cache Drawbacks

Unfortunately, after all the work, the non-blocking cache did not even increase IPC. This is because, before starting I fully realized the potential of nonblocking cache but not the drawbacks. The two main drawbacks were that the codebase didn't take advantage of our cache without needing there to backend logic to ensure loads were coalescing. If we had anticipated the major drawbacks of non-coalescing loads we may have been able to have a cache with extreme performance benefit. A sad end to the story, but a story to potentially continue over break.

Performance Analysis

Baseline CP3 Processor Frequency: 500 MHz, Area: 199420 microns squared

Benchmark Testcase	IPC (instructions/cycle)	Latency (microseconds)	Power (milliwatts)
coremark_im	0.3563	1636.56	21.971
aes_sha	0.4084	3197.55	20.042
compression	0.4632	1813.51	22.366
fft	0.4443	2315.89	20.720
mergesort	0.5147	1813.97	22.919

Isolated EBR Processor Frequency: 500 MHz, Area: 187257 microns squared

Benchmark Testcase	IPC (instructions/cycle)	Latency (microseconds)	Power (milliwatts)
coremark_im	0.4371	1334.12	24.419
aes_sha	0.4411	2960.40	23.201
compression	0.7167	1172.06	25.707
fft	0.4961	2073.92	23.245
mergesort	0.6491	1438.36	26.049

EBR + split LSQ + nextline prefetcher. Frequency: 606.06 MHz, Area: 210678 microns squared

Benchmark Testcase	IPC (instructions/cycle)	Latency (microseconds)	Power (milliwatts)
coremark_im	0.4628	1039.39	31.215
aes_sha	0.4123	2613.00	28.978
compression	0.7096	976.61	32.386
fft	0.5302	1601.17	29.512
mergesort	0.6426	1198.70	32.733
cnn	0.4635	4916.09	31.631
raytracing	0.2288	5208.78	28.434
rsa	0.2322	17482.33	28.979

Final Processor (added branch predictor and shortened multiply IP cycle length)
Frequency: 606.06 MHz, Area: 185090 microns squared

Benchmark Testcase	IPC (instructions/cycle)	Latency (microseconds)	Power (milliwatts)
coremark_im	0.6526	737.22	33.177

aes_sha	0.4633	2325.09	30.664
compression	0.8984	771.36	34.218
fft	0.6284	1350.91	30.769
mergesort	0.7186	1071.87	34.456
cnn	0.5916	3851.52	33.624
raytracing	0.5043	2363.78	33.345
rsa	0.2417	16793.9	30.971

Overall the largest performance gains were when we had integrated EBR and the branch predictor into our processor and when we optimized the multiply and divide IPs. For instance, just integrating EBR onto the baseline processor, we saw a huge improvement in latencies and IPC across the board on all benchmarks. We also changed the sizes of some structures which is why you see the area going down, however this clearly came at the expense of increased power.

Secondly, the next jump which was not as good, but enabled us to up our frequency and also pass RSA funnily enough was the split LSQ which we integrated together alongside the prefetcher. During the development process we found that aes_sha benchmark had a lot of loads and stores that we dependent on one another and this would've benefited a lot from store-to-load forwarding which we had implemented, but it we ultimately decided against it since it would've been our critical path and it would've only realize marginal performance gain on only a single benchmark. Although in certain cases such as FFT benchmark, we realized a non-marginal performance gain which we suspect were the out of order loads being utilized by other instructions which allowed our increased cycle overhead to be offset by this performance boost.

Finally, the last significant performance gain was when we integrated a simple 2-level branch predictor and also optimized our IPs such that our multiply cycle length was allowed to come down to 4 cycles (from 20 cycles) per multiply and our divide was allowed to come down to 16 cycles (from 20 cycles) per divide. The latter change realized significant boosts in performance to our hidden test cases, especially raytracing. We were able to jump from being stuck around 0.22 IPC to around 0.5 IPC for raytracing.

Speedup

Benchmark	Baseline CP3 Processor	Isolated EBR Processor	EBR + split LSQ + nextline prefetcher	branch predictor and shortened multiply IP cycle length
coremark_im	1636.56(1)	1334.12(0.82)	1039.39(0.64)	737.22(0.45)
aes_sha	3197.55(1)	2960.40(0.93)	2613.00(0.82)	2325.09(0.73)
compression	1813.51(1)	1172.06(0.65)	976.61 (0.54)	771.36(0.43)
fft	2315.89(1)	2073.92(0.90)	1601.17(0.69)	1350.91(0.58)
mergesort	1813.97(1)	1438.36(0.79)	1198.70(0.66)	1071.87(0.59)
Geometric mean	1.00	0.81	0.66	0.55

Conclusions

In conclusion, our final project was a challenging experience to fully flesh out a RISC-V out-of-order core, but we learned a lot about computer architecture in general. We hit all the objectives we set and managed to not only get a baseline working core, but also one with all the optimizations we initially selected. However, only some of them were beneficial to our performance. We definitely experienced the downsides of design and all the tradeoffs you must make in order to squeeze every bit of performance out of your core. The biggest thing we think that will benefit future groups is to do more prior research on the optimizations or at least talk to your mentor more. We didn't do both of these things and suffered a ton due to basically only one of the optimizations we selected having the biggest performance gain. If we were given another shot to further improve our processor, we would definitely have to either find a way to reduce power or to research alternative memory instruction queueing such as fire-and-forget. But, all in all, this MP was one of the most challenging and stressful MPs we have ever written and are satisfied with our ranking of third place in the final leaderboard.