ECE 411: mp_ooo Final Report
RushHour, Spring 2024
Ryan Kim (minjun3)
Akhil Nallacheruvu (analla6)
Sam Ayodiran (soa3)

Introduction

This report provides a comprehensive overview of the design of our out-of-order processor. We initially built an RV32I compliant processor, and later enabled instructions within the M extension. We built this processor using the Explicit Register Renaming method, and we used this method as opposed to Tomasulo's Algorithm because it provided components such RAT and RRF, which made the translation from architectural to physical registers easier to track. This method decoupled renaming from scheduling and allowed the data to be fetched from a single register file. ERR also provided a gateway to implement more advanced features into our processors. The ones we ultimately ended up implementing were the stride prefetcher, branch target buffer, pipelined cache, 2-bit saturating counter table, gshare/gselect, return address stack, dadda multiplier, and a Synopsys IP sequential divider.

Project Overview

In mp_ooo, our team designed an out-of-order processor that is RV32I compliant using the Explicit Register Renaming method. The goal of the project was to create an out-of-order processor that provides higher performance by reducing the penalties caused by false dependencies between nearby instructions. To create such CPU architecture, our group implemented key components that include physical register file, register alias table (RAT), retirement register file (RRF), free list, reservation stations, functional units, re-order buffer (ROB), and common data bus (CDB).

Initially, our group considered using Tomasulo's algorithm to implement an out-of-order processor. However, limitations such as lower clock frequency, higher area, and higher power consumption led us to favor the Explicit Register Renaming option.

For the three checkpoints of the mp, each member of the group worked separately on writing the initial code of the requirement. Then, we merged the code together and tested for correctness. Most of the debugging process after merging was done together. More details about administrative aspects of our project and work sharing will be explained in Milestones.

Milestones

Checkpoint 1:

Akhil: Draw a design block diagram (using draw.io or similar, not handwritten!) of the basic OoO datapath [5]

Sam: Implement a parameterizable queue (depth and width) [5]

Ryan: Implement instruction fetch and integrate it with magic memory and your parameterizable queue [7.5] (Your program counter should be initialized to 0x60000000)

We tested the parameterizable queue and fetch through two test benches, fifo_tb and fetch_tb. In fifo_tb, we filled up the entire queue to the size of its depth to check it does not push to the queue when it is full. Similarly, we popped the entire queue to check it does not read anything when the queue is empty. We also test concurrent push and pop and if the queue becomes empty on reset signal.

In fetch_tb, we first check the implementation of instruction fetch and its integration with magic memory by checking the value that is being read/popped from the queue. We also use ordinary memory to test if the queue gets populated correctly when fetching instructions at varying cycles. The fmax of this design is approximately 1.534GHz as the minimum clock is 0.652ns.

Roadmap for Checkpoint 2:
Akhil: Integrate the provided multiplier into your processor as a functional unit [5]
Ryan / Sam: Your design must support out-of-order execution of the arithmetic operations (you must implement ROB, RS, CDB, etc.) [5]

For checkpoint 2, we will use the provided multiplier and make changes if necessary to integrate into the processor. We will test the implementation by writing test cases for multiply operation similar to what was done for mp_pipeline using RVFI. Out-of-order execution for arithmetic operation will be implemented through ROB, RS, CDB, RRF, and RAT. RVFI

Checkpoint 2:
Sam: Integrate the provided multiplier into your processor as a functional unit [5]
Akhil/Ryan/Sam: Your design must support out-of-order execution of the arithmetic operations (you must implement ROB, RS, CDB, etc.) [5]

We have created 2 functional units so far: a multiplication unit and an ALU.The provided shift add multiplier is instantiated in the multiplication unit in order to execute multiplication operations. Both functional units output ready signals when they're ready to issue the instruction, and the CDB chooses which functional unit's instruction to send for issue. Our CDB prioritizes multiplication instructions over ALU instructions and will send the multiplication instruction to the ROB first in the event of a conflict.

Going off of the block diagram we created in CP1, we have created an instruction decoder that takes an instruction from the instruction queue and sends its architectural registers to the Rename stage. The Rename stage refers to the RAT to determine the mapping of the architectural register and replaces the architectural registers in the instruction with the physical registers that

correspond to them on the RAT. The Rename stage will then send this information along with the ROB ID to our reservation station, which will determine the correct functional units to send the instruction to and update the ROB accordingly. The CDB then takes the instruction of the available functional unit, and if there's a conflict between a multiplication unit and an ALU, the multiplication instruction will be prioritized, and updates the ROB with the instruction and ID, and updates the RAT with registers that aren't in use anymore. Once the instruction is committed, the physical registers will be sent to the RRF and Free Lists so that they can be used in the next instruction for renaming. The fmax of this design is 469MHz and the minimum clock is 2.13ns.

Roadmap for Checkpoint 3:
Akhil: Add support for control instructions [3]
Ryan: Add support for memory instructions [3]
Sam: Integrate with your mp_cache with competition memory (Instruction and Data) [3]

For Checkpoint 3, we will add support for the remaining control and memory instructions by building a branch predictor to enable branch predictions and enabling flushing of any branch mispredictions. We will also enable loads and store instructions in our processor and will enable the appropriate stalling logic for them. Finally, we will integrate our cache with the competition memory to increase performance.

Checkpoint 3:
Akhil: Add support for control instructions [3]
Ryan: Add support for memory instructions [3]
Sam/Ryan: Integrate with your mp_cache with competition memory (Instruction and Data) [3]

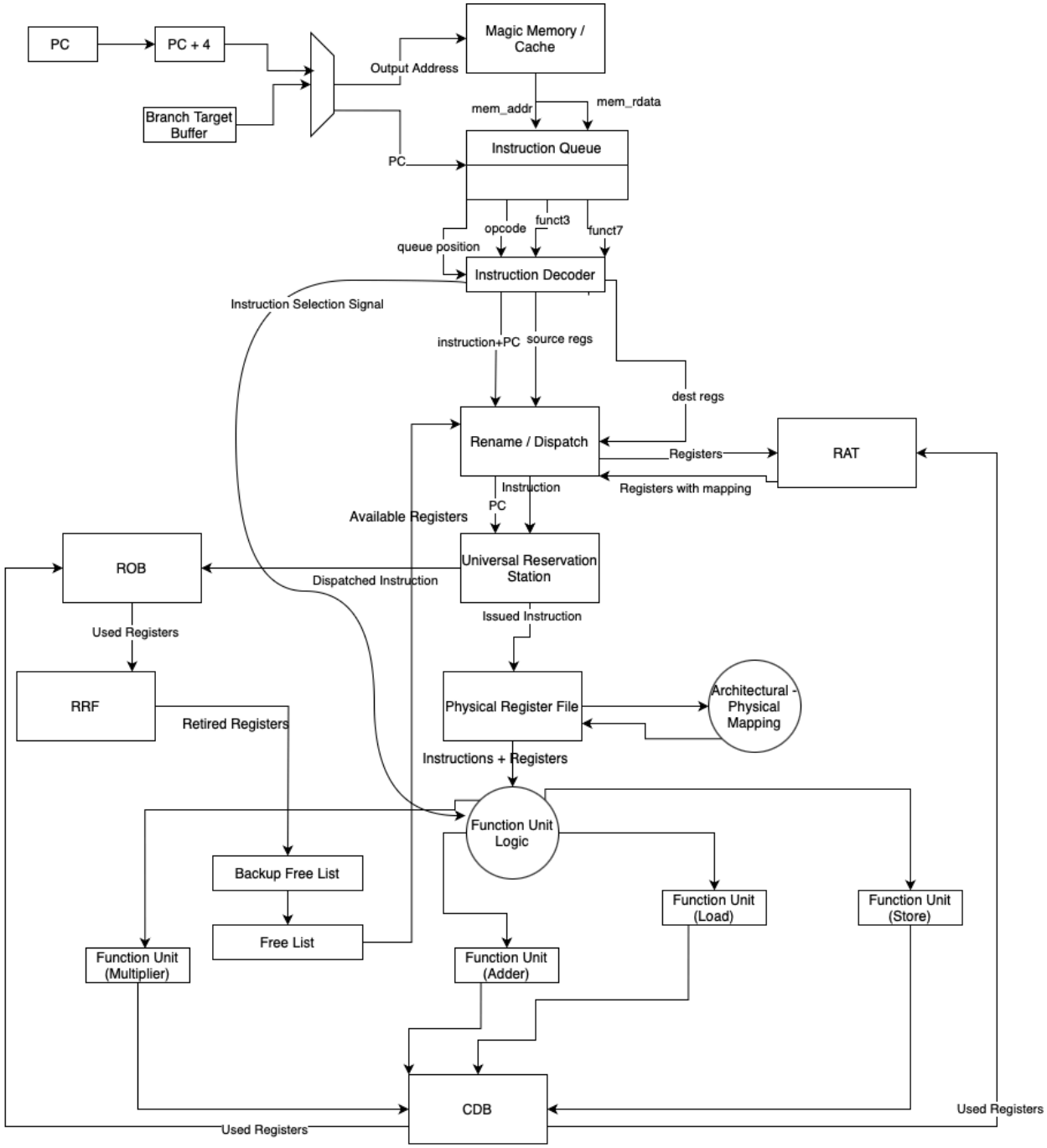
We have added onto our design by enabling load, store, jump, and branch instructions. We created separate functional units for memory and control instructions. The memory unit is responsible for taking in register and data values from the reservation station and sending the memory outputs and stall signal to the CDB. The control unit is responsible for taking in register and data values from the reservation station and sending the data and PC outputs along with a flush signal to the CDB. The CDB will arbitrate the data from the functional units and select the set of data it wants to send to the ROB. In our case, we have given highest priority to the control unit as a branch or a jump will cause subsequent instructions before the target PC value to get flushed out. After that, priority is given to multiplication instructions due to the high number of cycles they need to execute, and the lowest priority is given to the arithmetic/comparator unit.

We have also added a cache to the datapath and integrated it with the competition memory to allow for faster memory transactions. We have instantiated two caches: I-Cache and D-Cache. The instruction cache feeds data that it receives from memory to the fetch stage. The data cache has a read/write relationship with the functional units and will send them data that they have

requested. In order for both caches to be functional on one memory unit, we have instantiated an arbiter that prioritizes between I-Cache and D-Cache during memory transactions. It is implemented through round-robin scheduling and alternates between sending data to the I-Cache and D-Cache every cycle. The fmax of this design is 269MHz and the minimum clock is 3.71ns.

Roadmap for next checkpoint:

For the next checkpoint, we will be working on improving the performance of our memory for the competition by trying to pass the tests in the competition memory suite. We will do this by improving upon our existing code and adding advanced features. At this moment, the plan for advanced features are superscalar and division. We plan to implement superscalar processing by taking advantage of our current ERR design and adding the required functionality for it, and we plan to implement division by creating a module that thoroughly defines the division operation and a functional unit that will send the data of a division operation to a CDB.



Advanced Design Features

Pipelined Cache ([4] read-only): The read-only pipelined cache was implemented by first separating the cache module into i-cache and d-cache. We modified the i-cache so that it stays in the "compare tag" state when there is a hit, effectively reducing the imem_resp latency from 2 cycles to 1 cycle. Adding pipelined cache increased our processor coremark IPC from 0.252008 to 0.310616.

Stride Prefetcher [4]: The purpose of implementing stride prefetcher was to reduce the latency of receiving the next cacheline when there is a miss. When there is a cache miss and the current i-cache address is not equal to the prefetch address, the prefetcher will have the next cacheline ready by updating prefetch_data using prefetch address. However, our group decided to not include this advanced feature in the final design of the processor because it only added area and power while there was no improvement in performance because of the high hit rate and slow memory.

Branch Target Buffer [2]: The purpose of implementing a branch target buffer is so that we know where to jump to if we decide to implement any prediction scheme. Since we implemented a dynamic predictor this was necessary. We implemented the table using a direct mapped cache-like setup with SRAM. We ignored the bottom 2 bits of PC since our address space is byte addressable and used the next 7 bits of the PC address as the set index, and used 23 bits for the tag. We indexed into our BTB on every cycle and used it as our next pc if we predicted branch-taken. We wrote to our BTB on every committed branch that was taken. Adding a BTB didn't directly affect performance positively but it allowed us to implement our varying branch prediction methods which definitely improved performance. BTB doesn't work as well if there are many fresh branches that overwrite an already seen branch that has yet to be seen again.

2-Bit Saturating Counter Table[1-2]: We implemented a 2-Bit Saturating Counter Table as our main scheme for dynamic branch prediction. The upper bit tells us if we are predicting taken/not-taken and the lower hysteresis bit tells us our "conviction" essentially. On every branch instruction we indexed into this table using a 6 bit set index (64 entries) and looked at the upper bit to see if we should take or not take. Once a branch was committed we updated the status of the 2 bits ranging from "Strongly Taken," "Weakly Taken," "Weakly Not-Taken," and "Strongly Not-Taken" based on if said branch was taken or not taken. This predictor gave us a considerable IPC boost (roughly 25-30% on average), since every program benefited from some form of prediction other than static not-taken.

GShare/GSelect [4]: This feature utilizes a N-bit left shift shift register (N being the number of PC set index bits used) called a Global History Register (GHR) to keep track of the last N branches taken/not-taken status. Depending on if we wanted to use GShare or GSelect we either XOR'd or AND'd our set index from PC with the GHR to index into our BP table. The idea of GShare/GSelect is to utilize the idea of Branch Correlation, where past history of branches are a good indicator of future branches. In programs with low branch correlation it is not as useful.

Return Address Stack [3]: The purpose of this feature was to leverage the calling convention of RISC-V function calls. It works by maintaining a stack to track return addresses pushed by JAL/JALR instructions. When another JALR instruction is encountered, the RAS can pop and use the previously stored return address. This feature led to a very moderate performance improvement for most programs. On the negative side it can cause unnecessary area and power expense if the JAL's and JALR's in the program are not used for function calls.

M Extension (Dadda) [6]: The purpose of implementing a Dadda multiplier was to support faster multiplication. The original shift-add multiplier took 64 cycles to execute a multiplication

instruction. The implementation of a Dadda multiplier brought that number down to 16. This was done by following the algorithm for Dadda, which took two 32-bit values as inputs and generated 4 32-bit partial products through a combination of half adders and full adders, which are built up from partial products of lower values. The 4-bit partial products that are generated by multiplying 2 bits of each of the inputs are generated from half addition, and the subsequent partial products are generated from performing a full addition. Each addition is performed in a separate state, and the multiplication moves on to the next state only when it gets a ready signal from the previous state. Implementing this multiplier resulted in a decrease in delay.
Divider [3]: The divider that was implemented in this processor is a Synopsys IP sequential divider. The number of cycles that this division takes has been set to 16, and one instantiation of the DW_div_seq module is used in the division functional unit to determine the quotients and remainders for RISC-V division and remainder instructions. If the instructions are divu and remu, then the quotient and remainder are used as the results to be stored in the destination registers of these operations. If the instructions are div and rem, then the two's-complements of the quotient and remainder are fed to the destination register. This feature allows for greater functionality within the processor as it is now capable of taking in division and remainder instructions.

Conclusion
The design objective of mp_ooo was to create an out-of-order processor using Explicit Register. Some specific features we implemented include register alias table, retirement register file, free list, reservation stations, functional unit, re-order buffer, and common data bus. For optimizing our base design, our group decided to include pipelined cache, branch target buffer, 2-bit saturating counter table, return address stack, Dadda multiplier, and divider. With all the advanced features implemented, the coremark IPC increased from 0.252008 to 0.435099. Decreasing the size of the reservation stations and branch predictor also decreased the power and size, allowing us to increase fmax, which resulted in a noticeable increase in performance.