ECE 411 FINAL PROJECT REPORT

MP_OUT_OF_OFFICE

May 2, 2024

Albert Wang, Bobby Zhou, Cameron Choy

1 Introduction

For computer processors, there is more than one way to execute code efficiently. One way is to pipeline the stages of the processor, which increases both the throughput and frequency. Out-of-order processors take a further step to improve performance by executing non-sequentially, processing instructions during cycles that would normally be idle for sequential processors. For example, an sequential processor would need to wait a number of cycles for memory, but an out-of-order processor could execute other instructions while it waits for a response. As many modern CPUs use an out-of-order system, it is a very relevant field in computer architecture. As such, we designed our own out-of-order processor to explore and learn about the design and implementation of such a system.

2 Project Overview

The goal of this project is to design an out-of-order processor that finds an efficient balance between program runtime, power usage, and area. To measure the performance, the following equation is used to give a score achieved from running a testbench program (lower is better):

$$\text{Performance score} = \text{Power} * \text{Delay}^3 * \text{Area}^{1/2}$$

Based on this formula, delay has the most impact on the performance. So we aim to reduce delay from different aspects, such as increasing processor frequency and reducing instruction count. The second and third priorities are Power and Area respectively, which can be reduced with a simpler and more efficient design.

The processor we designed is a 32-bit RISC-V processor that supports the full RV32IM instruction set. This means that in addition to the base RV32I instruction set, our processor also supports multiplication and division. The processor is based on the Tomasulo Algorithm combined with a reorder buffer, which allows out-of-order execution and in-order commit. Development progress goals were divided into three checkpoints and a final submission as described in the following sections.

3 Design Description

3.1 Overview

Our processor implements the Tomasulo's organization with a reorder buffer. Figure 1 shows the basic components of the processor.

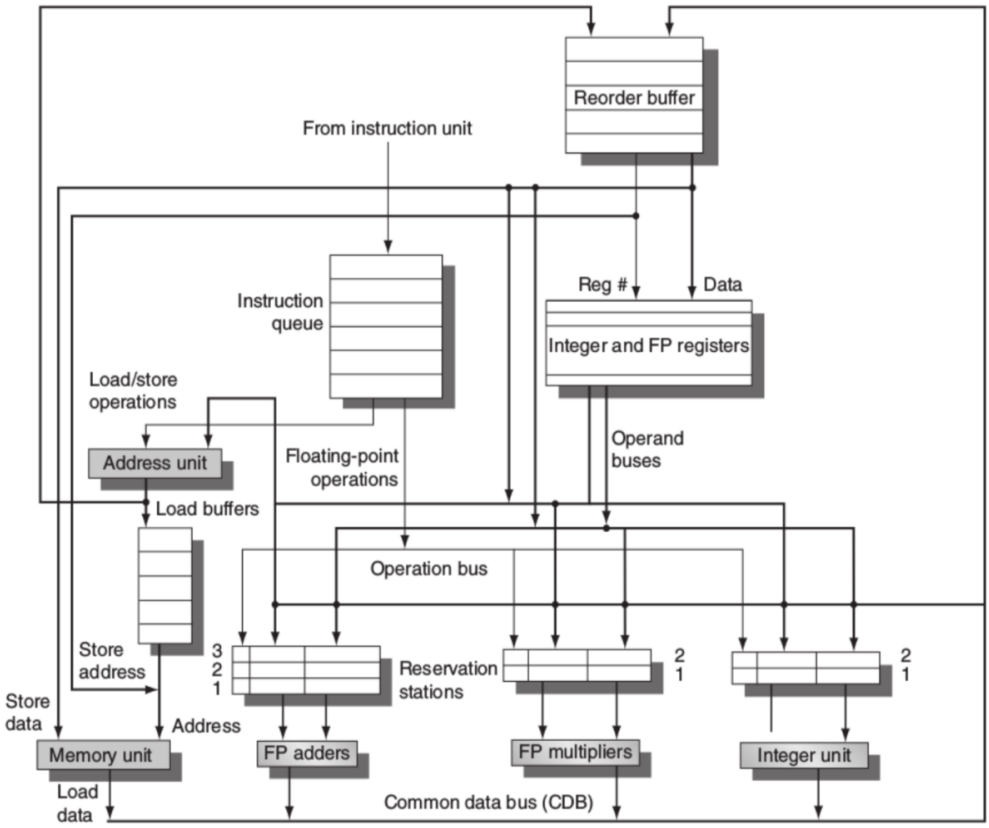

Figure 1: Basic components of the processor

The most notable components are the instruction queue, the reservation stations, the common data bus (CDB), and the reorder buffer (ROB). Here's a brief overview for each component:

- **Instruction Queue**: Holds instructions that are fetched from memory. Instructions are fetched in order and are dispatched its corresponding reservation station.
- **Reservation Stations**: Holds instructions that dispatched, which may or may not have their operands ready. During dispatching, it will check the register file, ROB, and CDB in that order. If the operand is not ready, it will snoop every CDB slot for every cycle. Each reservation station is also associated with a dedicated functional unit, which writes to the CDB when the result is ready.
- **Common Data Bus (CDB)**: A bus that broadcasts the result of the functional units. Our CDB has as many slots as the number of reservation stations so that each reservation station can write to the CDB without contention.
- **Reorder Buffer (ROB)**: Holds instructions that are dispatched and are waiting to be committed. The ROB is used to ensure in-order commit. It also holds the destination register and the value of the instruction. Once ready to commit, the instruction will write to the register file and get popped from the ROB.

3.2 Milestones

3.2.1 Checkpoint 1

For the first checkpoint, we have created a basic block diagram design for our processor. We implemented a basic FIFO instruction queue as well as a finite state machine controlling fetching logic. The memory model we use is a dual port memory that guarantees to respond in the next cycle. Figure 2 shows the block diagram of the processor for the first checkpoint.

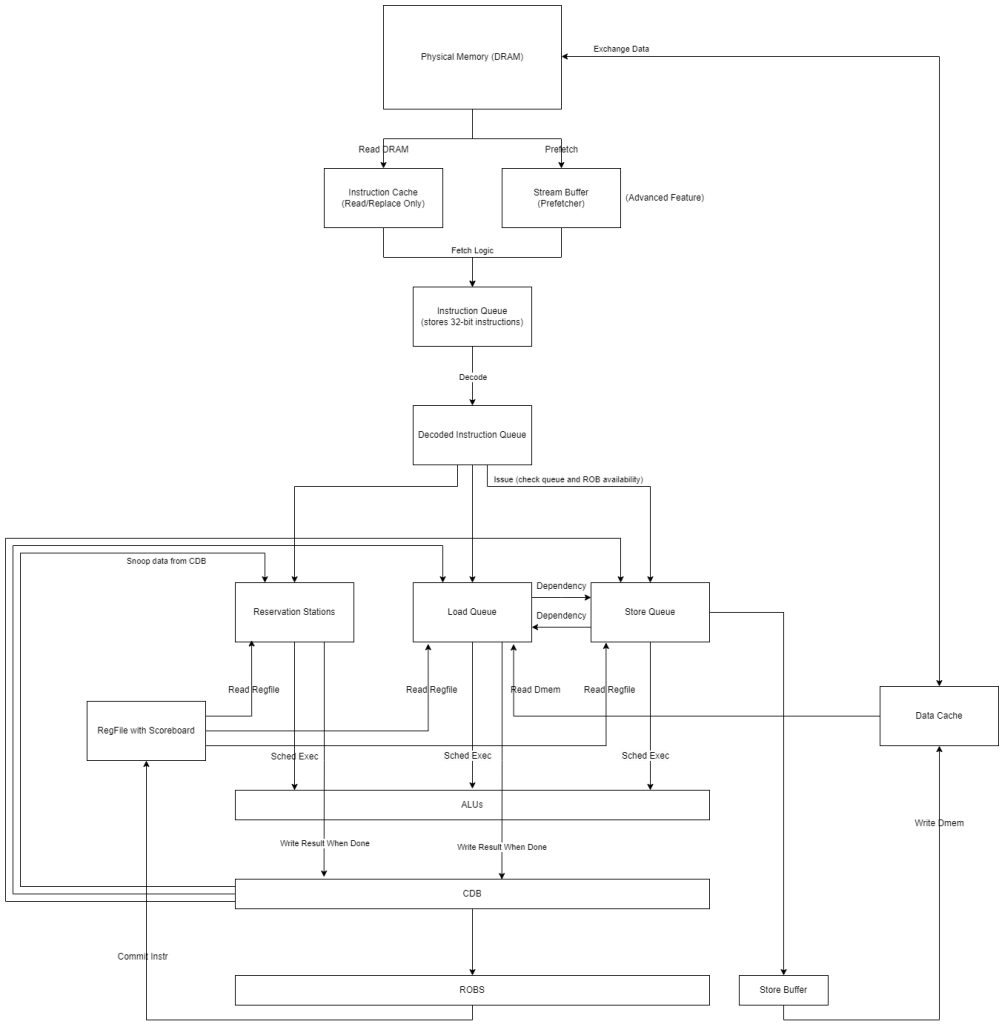

Figure 2: CP1 block diagram

To test this partial design, we manually populated the memory and checked the waveform of the signals, such as the internal states of the instruction queue, and the memory interface signals.

3.2.2 Checkpoint 2

Building on the first checkpoint, we implemented every instruction in the RV32I except the memory and control flow instructions. We have implemented the rest of major components

of the processor, such as the reservation stations, the functional units, the CDB, the ROB, and the register file. In addition, we also integrated a shift-add multiplier so that we can support the multiplication part of the M-extension. Figure 3 shows the block diagram of the processor for the second checkpoint.

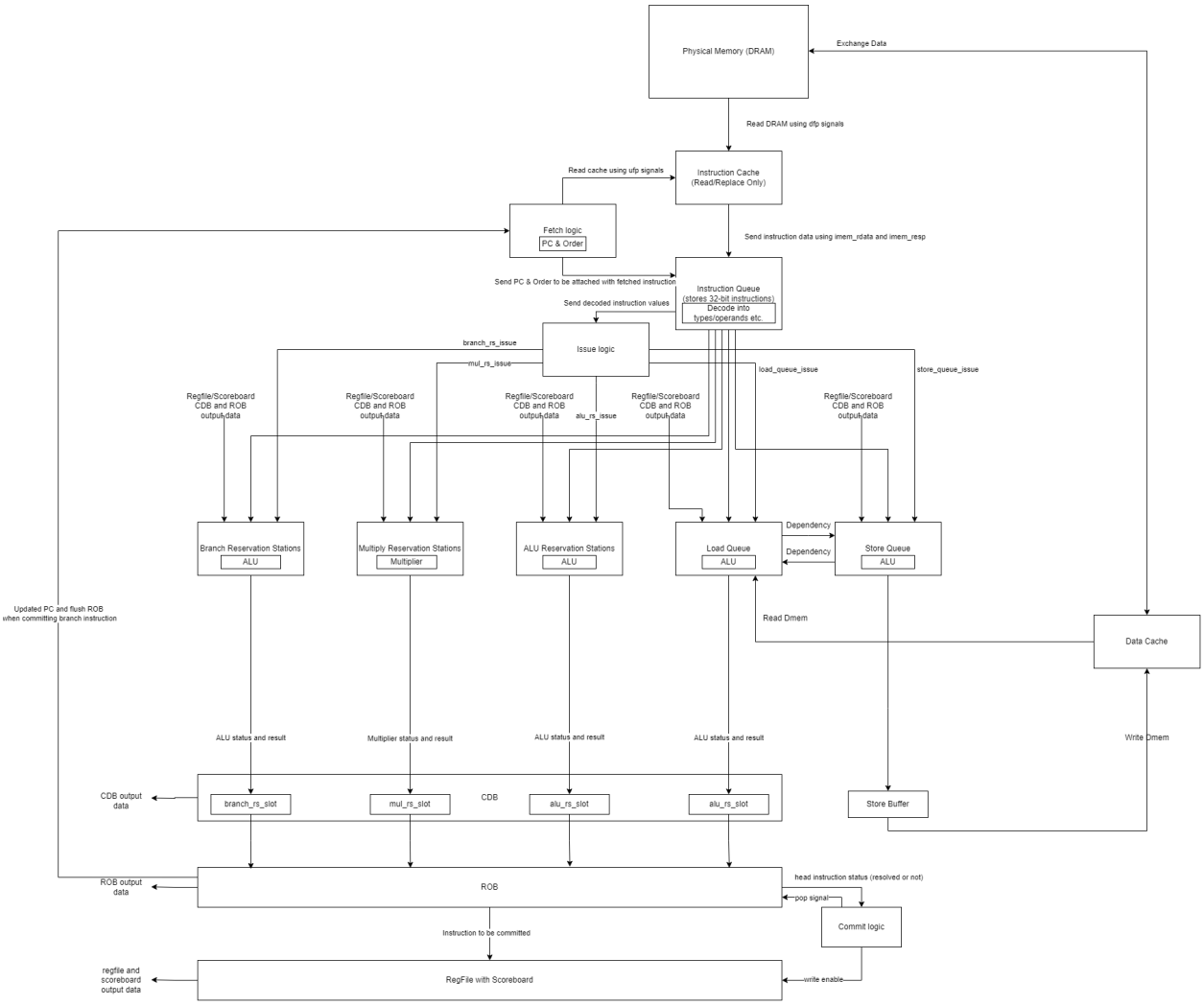

Figure 3: CP2 block diagram

To test our processor so far, we cannot use coremark yet, since we have not implemented the memory and control flow instructions. Instead, we used a series of targeted tests to verify the correctness of our processor. For example, we have a dependency test that checks if the processor can handle data dependencies in adjacent instructions. We also have an out-of-order test that checks if the processor can execute multiplication and other instructions concurrently while maintaining the correct commit order.

3.2.3 Checkpoint 3

For the third checkpoint, we have implemented the memory and control flow instructions. We also moved from the magic dual port memory to a more realistic banked memory, along with integration with the cache system implemented earlier in the semester. To implement the control flow instructions, we have added a branch reservation station that calculates both the branch target and the branch condition. To implement the memory instruction, we decided to use the naive approach to deal with memory hazards, which is to stall dispatching load instructions until there is no in-flight store instruction. Figure 4 shows the block diagram of the processor for the third checkpoint.

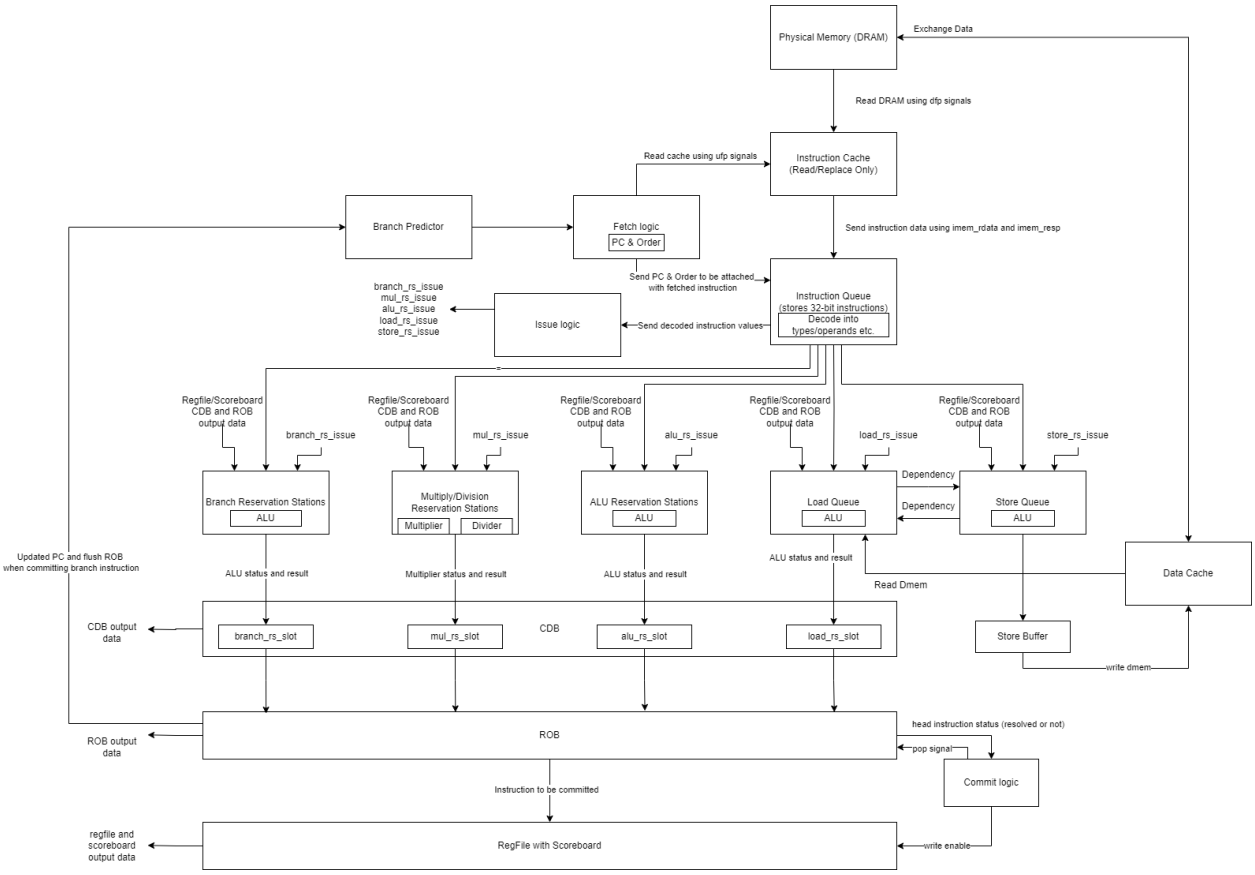

Figure 4: CP3 block diagram

With all of RV32I implemented, we can now run coremark to verify the correctness of our processor. We used a combination of RVFI and spike to compare the output of our processor with the golden reference. We also used the waveform to verify the correctness of the processor.

3.3 Advanced Features

At the end of checkpoint 3, our processor can run coremark with IPC of 0.08, which is not very efficient. We noticed a lot of areas that can be improved:

- branch predicton: our processor uses a static-not-taken branch prediction, which is not very accurate. The frequent misprediction can cause a lot of pipeline flushes and reduce the IPC.
- load store hazard: our processor stalls the dispatching of load instructions until there is no in-flight store instruction. This can cause a lot of pipeline bubbles and reduce the IPC.
- multiplication: our processor uses a shift-add multiplier, which is not very efficient. We can replace it with a more efficient multiplier such as the Dadda multiplier.
- banked memory and cache: banked memory access time is orders of magnitude higher than the processor execution time, which means the processor spends a lot of time waiting for memory. We can improve this by adding a pipelined read-only cache to compensate without adding too much complexity.

Therefore, we start to implement a series of advanced features:

- **Branch Prediction**: We implemented a Gshare branch predictor, which is a simple and efficient branch predictor. The Gshare predictor uses a global history register to index a table of 2-bit saturating counters. The global history register is updated with the branch outcome and the saturating counter is updated based on the actual branch condition. We also implemented a branch target buffer to store the target address of the branch instruction. The branch predictor is integrated with the fetch stage of the processor, which overwrites the PC with the target address if the branch predictor predicts a taken branch.
- **Memory disambiguation**: Store is always being committed in order and only writes to the data cache when it is at the top of the ROB. The load could be executed out-of-order. Therefore, the load can only be executed when the address of the load reservation station is solved and all store instructions younger than the load in the store queue are solved.

  One challenge is to make sure load/store works with branch flushing. To accommodate the flush signal, we should not send new read/write requests to the data cache when there is a flush signal. Since our flush finite state machine will wait resp signal for all inflight data/instruction cache requests.
- **Dadda Multiplier**: The multiplier was changed from using shift-add algorithm to a dadda multiplier. Dadda multiplication is an algorithm that calculates the product by summing partial products to reduce the Dadda tree until the product is achieved.

  Our specific implementation also used vedic multiplication to achieve 32x32 multiplication with 16x16 multipliers, which are created from 8x8 dadda multipliers. This was

done as it is simpler to implement and to meet the deadline on time. However, this takes a larger area and slightly more stages. A 32x32 dadda multiplier pipelined by stages can complete its operation in at least 8 cycles, however, our implementation with vedic multiplication requires an additional cycle to finish. Additionally, to meet the processor's timing requirements, additional stages were added to avoid a critical path, resulting in our multiplier taking either 10 or 12 cycles, depending on whether the resulting product is positive or negative. This is still a large improvement over the shift-add multiplier, which takes 32 cycles to finish.

- **Read-only Pipelined Cache**: Even though we support sending instruction read requests at the resp cycle, the instruction cache finite state machine will go to the idle stage again. We do the read-only pipeline cache by accelerating this situation. When there is a resp signal (hit in the compare stage), the cache also receives a request from the fetch module, we will change the set value in that cycle. For example, if the first address is sent to SRAM in the first cycle, the hit/miss status and read data for the first request will be available in the second cycle (suppose it is a hit), we could change the first read address to the second read address in the second cycle, and get the hit/miss status and read data for the second request at third cycle.

- **Divider (Synopsys IP)**: After these advanced features, we noticed that our processor still has high latency compared with other groups when running the physics.elf and dna.elf program. Noticing that these two programs are the only ones that use the division instructions, we decided to use the Synopsys IP divider to see if it can improve our performance.

  We decided to use the sequential divider IP with 16 cycles to complete such that it doesn't create a critical path and lower our processor frequency. After integrating the divider correctly, we are able to run the physics.elf and dna.elf program with full M-extension enabled. Although the IPC barely changed, the instruction count is significantly reduced, which improves the latency.

Using coremark as an example, the performance of our processor proceeded as follows:

- Implementing the basic processor: IPC = 0.08
- Implementing the branch prediction: IPC = 0.08 $\rightarrow$ 0.24
- Implementing the Dadda multiplier: IPC = 0.24 $\rightarrow$ 0.40
- Implementing the pipelined cache: IPC = 0.40 $\rightarrow$ 0.53

At the end, our processor can run at 444 MHz. It has an area of 197753 μm^2. Its power usage is around 35 mW depending on the workload.

4 Conclusion

We were able to design a 32-bit RISC-V out-of-order processor that supports the full RV32IM instruction set. The processor is based on the Tomasulo Algorithm combined with a reorder

buffer, which allows out-of-order execution and in-order commit. We have implemented a series of advanced features to improve the performance of our processor, such as branch prediction, memory disambiguation, Dadda multiplier, pipelined cache, and Synopsys IP divider. Our processor can run coremark at 444 MHz with an area of 197753 μm^2 and a power usage of around 35 mW.