ECE411 MP3 Final Report

Topaz
Ben Schreiber, Neil Singh, Saurav Puri

1. Introduction

The goal of the project is to develop a fully functioning LC-3b instruction set pipelined processor. Our primary focus is to complete functional pipelined processor while keeping performance in mind. First, we cover the project overall goals and design choices we made. The next section goes into a more in-depth design description. This section includes our milestones, advanced design options, and testing methodologies. The next section covers notable design difficulties and lessons we learned throughout the project. Finally, we end the report with a conclusion of the project with our achievements and goals we met.

We created an out-of-order execution processor based on Tomasulo's Algorithm. This algorithm allows for parallel execution of instructions that would otherwise stall under other algorithms such as scoreboarding. Tomasulo's algorithm is advantageous on systems with instructions that take many cycles to complete or on systems with high instruction level parallelism. Alongside the OoO processor and branch predictor, we implemented a two-way L1 data and instruction cache, a unified four-way L2 cache, a cache arbiter, and pseudo LRU.

2. Project Overview

The project involved dividing the workload among the group modularly, then integrating the parts together. The project was divided into a handful of stages. First, we had top-level paper planning of how our overall design would look and function. This usually was the basic architecture of the pipeline with little to no controls determined. We began transferring each component into SystemVerilog code. Each component was modularly designed and modularly tested to ensure they work as intended. After ensuring modular functionality, each piece was incrementally added to again verify correct behavior. At this stage, interactions between modules are important and control signals are required to determine behavior. Finally, all the pieces were added together then tested with final given competition codes.

The division of work can be summarized as: Ben & Neil worked on ISA, datapath, Hazard handling, and forwarding, Ben worked on the load-store queue, Neil worked on Branch Prediction , while Saurav worked on datapath and cache. We used GitLab for easy version control and commit management to avoid conflicting code.

3. Design Description

3.1. Overview

Since we decided to do Tomasulo rather than a in-order pipeline, all of our checkpoints were not set in stone and we were given general deadlines by our TA. The goals completed per checkpoint are listed below.

3.2. Milestones

3.2.1. Checkpoint 1 (March 17th)

Instruction queue, alu reservation stations, and reorder buffers implemented. One instruction fetched per cycle with space allocated in the reorder buffer. The arbiter dispatches the instruction to the ALU and is then committed in order. All dependencies are resolved by the reservation stations.

3.2.2. Checkpoint 2 (April 2nd)

Added implementations for JMP, JSR, JSRR, LEA, Branch.
Began to develop load-store methodology. Initial attempt involved partial ordering with OoO loads.

3.2.3. Checkpoint 3 (April 16th)

Added implementations for LDR, STR. Began to test design with physical memory and added branch prediction. Begin work on cache implementation.

3.2.4. Checkpoint 4 (April 23rd)

Added implementation for LDB, LDI, TRAP. Added L1 instruction and data caches, L2 combined cache, and arbiter.
Rewrite of load-store queue following difficulties with forwarding. Current version is strict ordering, with stores stalled until head of ROB.

3.3. Advanced Design Decisions

Most of our main advanced design choice was doing Tomasulo out-of-order execution. This turned out to be very time consuming and so we were not able to pursue many other planned advance design decisions. We implemented branch prediction, with a BTB and local history, and a four-way L2 cache with Pseudo LRU policy. We did not measure performance without these advanced design choices because we had planned to do these from the beginning; however, a larger cache would ideally give more speedup at the cost of more power consumption. Similarly, adding a branch target buffer would achieve speedup at the cost of power consumption.

3.4. Testing Methodologies

Small test codes were used for individual instructions to ensure correct basic implementation. For example, the test code for LDI is just six lines long. Then slightly more sophisticated test codes were written to test larger features. We used a factorial program to test the effectiveness of our branch predictor.

We largely relied on provided test code to find edge cases. Most of our edge cases and challenging bugs were with instruction fetch and stalls, bus arbitration, and forwarding. Initial testing was done with magic memory to resolve basic control flow issues, with physical memory introduced while the cache was tested.

Testing is generally challenging because of the large number of machine states possible. Compared to a conventional pipeline, the use of queues and reservation stations in Tomasulo means data is harder to follow around the machine, and may appear in multiple places. We coped with this by attempting to streamline the wave review process as much as possible, using .do files for fast setup.

4. Additional Observations

4.1. Notable design bugs

We encountered several particularly challenging bugs over the course of the project. A lot of the issues stemmed from forwarding. Frequently, a value would be taken from a register the same cycle that that value was updated, at times resulting in incorrect execution. Forwarding only affects a few registers in an in-order pipeline, but do to the scattered nature of data in Tomasulo our forwarding paths were numerous.

Another stubborn bug involved instruction fetch. If a stall occurred during an instruction fetch, the values would be latched as no memory request would occur. However, the instruction queue still attempted to read in a new instruction. The cycle after the stall completed, the new value would not yet arrive due to memory latency, but the i-queue would again read in the old value. This lead to duplicated instructions. It was resolved by refusing to serve instructions from fetch during stall, instead saving a pending instruction for the cycle immediately after the stall completed.

4.2. Lessons learned

We assigned distinct roles very early in the design process, which lead to parts of the design only being understood by one group member. It would have been better to spend more time writing code in a group setting, and concurrently documenting this code for reference.

It also would have been better to plan more of our design on paper ahead of time, including a module hierarchy. Some parts of the design, like the reservation stations, became somewhat messy and hard to read.

The load-store queue was a particular challenge, as Tomasulo naturally does not lend itself well to memory operations. We originally attempted a moderately aggressive design, with loads executing when their operands were ready, so long as no earlier stores shared the address or did not yet have a calculated address. If an earlier store mapped to a later load, the value would be forwarded.

From a microarchitectural view, it was hard to reason about a load queue where any entry could potentially be popped. This was resolved using a collapsing queue that continually shifted entries down to fill openings. It was also hard to keep track of which stores came before a load, and which came after. This issue, along with forwarding headaches, lead us to abandon this design for the slower, strict-ordering currently in place.

5. Conclusion

In conclusion, we successfully achieved our goals and designed an out-of-order execution processor with many other advanced design choices. We were able to get functionality of all the example codes including provided competition code. Due to lack of time, we were not able to perfect our processor and so the speedup and power consumption may not be optimized. We learned that we should have worked more on paper designing the processor because many issues and time management issues could have been fixed by seeing how long other potions would take. This also led to unbalanced partitioning of work because certain parts were more difficult to debug and caused critical issues. We also did not compare the speedup of our design choices versus the default or other design choices which would have been a analytical way of determining speedup. In the end, we were able to complete the integration of all our parts and create an out of order execution processor.