CS/ECE 552 Spring 2019 Final Project

Team 10
Matthew Bachmeier, Amanda Becker, Garret Huibregste

Design Overview

In ECE / CS 552 – Introduction to Computer Architecture, we have been tasked with the project of making a working, single-core processor over the course of about 3 months. There are 36 instructions that we were required to implement, properly referred to as the WISC-SP19 ISA, which is similar to the instructions implemented in the MIPS architecture, except using 16-bit instructions rather than 32-bit. We've come a long way, having begun with a single-cycle, unpipelined processor using single cycle perfect memory, and ending with a 5-stage pipelined processor that uses a 2-way set-associative cache to access both data and instruction memory.

To begin designing the single-cycle, unpipelined processor with single cycle perfect memory, we came together to draw a top-level schematic that showed all of the basic units we would need, such as register files, an ALU, a control unit, and a memory system, along with all of the multiplexors and other simple logic units we would need. As we designed this phase of the project, we kept in mind that we would eventually be moving towards a pipelined design, and grouped our logical units accordingly. This was the simplest part of the design, and it came together quite nicely.

After our single-cycle processor was functioning, we moved on to implement a 5-stage pipelined processor, still with single cycle perfect memory. We broke the five stages down into Instruction Fetch, Decode, Execute, Memory Access, and Writeback. We added stalling logic to pause our processor when instructions had dependencies on prior instructions. The Instruction Fetch stage was responsible for updating the PC, fetching the next instruction from memory, and determining whether there were any errors involved with the fetched instruction. In the Decode stage, we broke down the instruction sent in from the IF stage and sent it to our Control Unit. From the Control Unit we were then able to determine our write-back register, sign and zero extension of passed-in immediate values, some branch and jump logic, and again if there were any errors while decoding. In this stage we also determined write data that needed to be passed on to the Register Memory in the next stage. Our Execute stage used signals to determine which of up to three registers in an instruction, or an immediate, would be passed in to the ALU as inputs 1 and 2, which kind of operation we would be performing, and any errors that occurred. Out of the ALU came the output of the ALU operation, as well as a few flags to signal if the output was zero, negative, or had overflowed, which were sent to the Decode stage to computer branch logic. The Memory stage was responsible for accessing memory at a location determined in previous stages, either by a direct immediate, the PC and an immediate, or some result of an ALU operation, and determining if we were writing to or reading from that location. The write data leaving this stage depended on if the instruction was writing from memory to a register, and error logic was again computed. Our Writeback stage was very minimal – it was predominantly used to determine if the data being written back to a register was to come from a memory access or the output of the ALU. Four register files were created, each being placed in-between the five stages, to connect wires and make decisions for the next stage's PC and instruction, which required carefully thinking of contingent instruction dependencies and stalls. These register files is how the pipeline was constructed, so that different instructions can be going through the processor core at the same time. This is where we spent most of our time for this phase.

For the final stage we created a 2-way set-associative cache with multi-cycle memory to hold instruction and data memory from the aligned main-memory system. To make things simpler we created a separate module for only the cache finite state machine, which had 25 states to handle cache hits, misses, and evictions based off signals coming in from the memory and cache modules. Once this finite state machine was integrated into the memory system block and hooked up to the cache and main memory, and passing on a working processor, the new memory system block replaced the old single-cycle perfect memory. With instruction and data caches in-place, we finished off our processor with register file bypassing, EX-EX and MEM-EX forwarding to reduce the number of stalls necessary for dependencies. We created Predict-Not-Taken Branch Logic that would continue fetching instructions immediately following branches until the branch decision was made in the Decode stage. If a branch was determined to be taken, the fetched instruction would be flushed, and the instruction indicated by the branch would be taken. If the branch was correctly predicted as not taken, instructions would continue to execute in program order. Each of these optimizations were a small challenge on their own, but proved to be much more complex when implemented synchronously.

Optimizations and Discussions

Very early on we replaced our Ripple-Carry Adder with a Carry-Lookahead Adder. As required, we implemented Execute-Execute and Memory-Execute Forwarding, along with Predict-Not-Taken Branch Logic, which decreased the time we spent stalling. We also placed branch and jump logic into the Decode stage to reduce the number of instructions that would need to be flushed when a branch or jump was taken. Instead of using a direct-mapped cache, we used a 2-way set-

associative cache to decrease cache access time. One optimization that we didn't fully succeed in implementing was filling the pipeline – if a memory access was stalling the pipeline and there were NOPs in earlier stages, we pulled consequent instructions into the pipeline so that execution of valid instructions could continue immediately after memory access exited, therefore decreasing CPI.

We were able to create a fully functioning WISC-SP19 ISA processor, using the solution's 2-way set-associate cache modules. We had hoped to use our own cache in the final project, but there were too many bugs associated with it, and the solution cache had a lower overall CPI than ours did. Given more time, we would have been interested in implementing LRU Cache Replacement Policy for both instruction and data memory as this would make cache hits more frequent, therefore lowering overall CPI. We also wanted to add execute-decode and memory-decode forwarding but ran out of time. This optimization would have greatly improved the CPI for branch instructions.

Design Analysis

A table listing the possible hazards that arise in your pipelined design and the number of stall cycles that each hazard incurs. (Maximum half a page)

Hazards	**Number of stall cycles**
RAW register dependencies, ALU to ALU	0 (Ex-Ex forwarding)
RAW register dependencies, MEMORY to ALU	0 (Mem-Ex forwarding)
RAW register dependencies, MEMORY to MEMORY	1
RAW register dependencies, MEMORY to BRANCH/JUMP	1 (Register file bypassing)
RAW register dependencies, ALU to BRANCH/JUMP	2 (Register file bypassing)
Cache hit	0
Cache miss, eviction	11
Cache miss, no eviction	7

Our two-way set-associative (not the instructors cache used in final design) cache had 25 different states in its FSM. First, the read and write signals were analyzed to determine which was being requested, if either, and an error was thrown if an attempt was made to write and read at the same time. If either a valid read or write signal was passed, we then checked to see if the address being accessed was already in the cache. If there was a hit, data was written or read, and we returned to the idle state – cache hits incurred 2 totals cycles. On a write-miss that wasn't dirty we would load the address from memory, and then write that value back to the cache before returning to idle, a total of 9 cycles. On a write-miss that was dirty, the old value was first written to memory, then the new address extracted from memory and placed into the cache, as before, totaling 13 cycles. On a read-miss that wasn't dirty, the address was found in memory and placed into the cache, where it could be read, in 9 cycles. On a dirty read-miss, the old value was written to memory at the proper location, then the new address was extracted from memory to be read from cache, over 13 cycles. If set 0 was not dirty, new values were first placed there. If set 0 was dirty but set 1 wasn't, then the new value was placed there. If both sets were dirty, then we relied on a signal called victimway to determine which value would be victimized and sent back to main memory. This signal differed depending on whether we were in the Instruction Cache or the Data Cache. In the I Cache, victimway was inverted every time an instruction was fetched, whereas in the D Cache, victimway was inverted only if a successful read or write of the cache occurred.

Conclusions and Final Thoughts

Many beneficial skills have been learned or improved over the course of this project. Time-management was very important. During the first couple of phases we started working on the project as soon as there was information on it, and we tried to meet to work on it multiple times a week in relatively short (2-4 hour) increments. In doing so, we finished our work a few days before it was due and were able to turn it in with confidence. When it came time to start implementing the cache, we took about a week's break from it, which came back to bite us a bit closer to deadlines. We started meeting for longer periods (5-10 hours) and the progress we made became slower as problems became more complex and required more concentration to resolve. We did get everything submitted by respective deadlines, but some deadlines came up quicker than we anticipated. It would have been hard to do this differently, as other classes and personal lives become more complicated

later in the semester, though if we had been able to make it to office hours during the week, perhaps some issues could have been resolved a bit quicker.

We were all glad to have taken ECE 551 prior to ECE 552 so we were already past the basic learning curve of writing in Verilog. However, not being allowed to use System Verilog and having to conform to certain Verilog restraints slowed us down a little bit, though we were able to overcome all obstacles in the end. Creating a processor with a complex modular hierarchy of Verilog files was quite a challenge. For the first phase, we all met and drew up how our processor was going to operate and be inter-connected. This was extremely beneficial to our coding efforts, and we definitely could have saved some headache in later phases if we had devoted time to drawing up schematics for phases 2 and 3 – with so many signals being shared by many different stages, being able to look at a wiring diagram of our code would have made connections more obvious.

Overall, we all learned a great deal from this class. There were many difficulties and challenges throughout the project, but we tried our best to face these issues head on. Although there were many challenges that caused us all headache and stress dreams, this is hands down one of the best projects we have ever worked on. Not only did we learn about the design process of a single core processor, we also learned how to work as a team and face adversity. The rewarding feeling we got when a certain aspect of our design began to work is unparalleled and something we constantly will strive to feel throughout our future career paths. In conclusion, we spent a lot of time struggling with this project but in the end it was worth the effort and taught us so much about the design process and team work.