CS/ECE 552 Spring 2020 Final Project

Team-12

Qinjun Jiang, Ruokai Yin

Design Overview

Brief Overview:

This is a semester-long project targeted on completing the functional design of a 5-stages pipelined microprocessor called the WISC-SP20. All components of our design are written in Verilog.

The feature of WISC-SP20 includes caching, data forwarding, and exception handling, etc. All the instructions supported by WISC-SP20 are listed in the link below: http://pages.cs.wisc.edu/~sinclair/courses/cs552/spring2020/handouts/project/cs552-spring2020-projectISA.pdf

There are five pipeline stages for WISC-SP20: fetch, decode, execute, memory, and write-back. Fetch stage is responsible for choosing the right PC into the instruction-memory and fetching the desired instructions out from the memory. Decode-stage is responsible for decoding the instruction and sending the corresponding control signals into the later stages. Decode-stage also sends the proper register data into the later stages as well as stores the data from the write-back stage into the proper register. Execute stage handles all the arithmetic computations for WISC-SP20 and delivers the proper results into the later pipeline stages. Memory-stage contains the data-memory and is responsible for handling memory-related read and write requests. Write-back-stage makes decisions of which data to be written back to the decode stage.

WISC-SP20 implements two identical caches for instruction-memory and data-memory. In general, our cache is a two-way set-associative cache with the write-back scheme. The detailed performance of our cache design will be discussed in the Analysis part below.

For this project, the work that has been done includes but not limited to, designing of schematic diagrams, implementing the design into functional Verilog codes, and making optimizations on CPI of the design. Ruokai pays more attention to Verilog code implementation while Qinjun spends more time in schematic designing. Overall, Ruokai and Qinjun both contribute dedicatedly and evenly to the project.

Optimizations and Discussions

Several optimizations have been done. They are mainly the optimizations targeted on improving the CPI performance of WISC-SP20.

We noticed that the bottleneck of our processor's CPI lies in the Cache stall cycles. Before being optimized, our cache needs 3 cycles for a hit, 9 cycles for a miss(without writing back), and 13 cycles for a miss with write back. We reduce the number of cycles for a hit down to one, thus reduce 2 cycles for any cache operations.

Another optimization is also targeted on the stall cycles. We noticed that in our previous designs, our fetch stage can dive into an instruction-memory stalling while there is a branch decision that is being decoded out of the decode stage. In the previous design, our processor will wait for the instruction-memory for a minimum of 7 cycles stall then flush out the instruction that has just been fetched out due to the branch decision in the decode stage. The change is minor: just disable the instruction memory when there is a branch decision in the decode stage. However, the gain is decent, we successfully reduce 6 redundant stall cycles for this situation.

Other optimizations include two additional data-forwarding paths: MEM-MEM forwarding and EX-D forwarding and the LRU cache replacement policy. These optimizations are done in the extra-credit version of the design.

Design Analysis

Pipeline Hazards Analysis:

After implementing EX-EX, MEM-EX, EX-D, MEM-MEM forwarding, there are still three types of data hazards that require stalling.

Reason	**#Stall cycles**	**Notes**
data dependency on previous LD	1	1) MEM-EX forwarding can be applied after 1 cycle stalling 2) Not for Rd in ST because of MEM-MEM forwarding
data dependency of Branch, JR, and JALR on previous instruction	1	EX-D forwarding can be applied after 1 cycle stalling (except for LD)
data dependency of Branch, JR, and JALR on LD in such pattern: LD, one other instr, Branch/JR/JALR	1	There is no need to do forwarding after 1 cycle stalling as we have register bypassing.

control hazard of making branch decision in the Decode stage	1	As we implemented Predict-Not-Taken for branch prediction, we need one cycle stalling and flushing when making an incorrect prediction.

Cache Design Analysis:

As mentioned in the optimization part, we have done a decent amount of optimizations for our cache design. After the optimization, our cache controller now has the following cycle delay.

For a cache-hit, it takes a single cycle to finish. That means our processor will not stall for a data-cache or instruction-cache hit.

For a cache-miss with the eviction of a line, it takes 11 cycles to finish. This is also the critical path in our cache-controller state diagram. Our cache needs to traverse all 11 states in our state machine to write back the victim line(if both the dirty and valid bits are set) and fetch out the new line. One cycle is used to determine that there is a miss, followed by four cycles of write back. Then six cycles are needed to fetch the desired line out of the memory.

For a cache-miss without any eviction, it takes 7 cycles to finish. As described above, one cycle is used to detect the miss, and six following cycles are used to fetch out the desired line.

Conclusions and Final Thoughts

By doing this project, our understandings of pipelined processors have grown much deeper compared to before. We gain more experience on how to combine different functional units not only spatially but also temporally. The experience of writing large amounts of Verilog codes is also valuable. We also become quite familiar with Github, which is a very good skill for computer science students to have.

One thing that we would have done differently is to think more and plan more about how our design should deal with control hazards before we start writing the code. The debug of multiple overlapping stalling signals requires quite a bit of design sanity.