CS/ECE 552 Spring 2020 Final Project

Team 17
Yichen Li, Simeng Li

Design Overview

Our project was about building up a simplified microprocessor, which is 5-stage pipelines, and includes instruction cache and data cache. In our group, we worked together in most times. We usually worked together in implementations. Yichen contributed the most of debug processes, and Simeng did more in drawing diagrams and schematics.

There are five stages in the pipeline, including fetch, decode, execute, memory, and writeback. For the pipeline registers, we put them before the output of the stage so that the outputs of the stages have already gone through the pipeline registers.

In the fetch stage, we mostly fetch the instructions according to the current PC and pass the PC and instruction information to the next stage. In the decode stage, we read the register files, decode the instructions, output control signals, and modify register data. We also resolve the branch in the decode stage. In the execute stage, we compute the result of the instructions using the signals and data passed from the decode stage and output the final data. In memory stage we read or write the memory data we want from the data memory and pass signals needed for writeback stage. In the writeback stage we use the select signals passed in to write the writeback data to correct positions.

The hazard detection unit is outside of all 5 pipeline stages. If it detects hazards, it will send hazards and forwarding signals and help to forward the correct data to the destination stage (or it will stall when necessary).

We have 2 caches in total: the instruction cache in the fetch stage and the data cache in the memory stage. When either of the caches is working and not asserting Done signal, all the pipeline stages will stall to wait. We have mainly 3 modules in cache: memory, control unit, and cache. We use a two-way associative cache for the cache units and use a set of logic units to control the data flow of them. We also designed a state machine in cache control to control the whole system.

Optimizations and Discussions

For the optimizations, we applied four methods including resolve branch in decode stage, resolve direct jumps in fetch stage, mem-to-mem forwarding and using dynamic branch prediction.

For resolving branch in decode stage, we make the decision in decode stage according to the signals we decoded from the instruction.

For direct jumps, we resolve them in the fetch stage, which is a byproduct of branch prediction.

For mem-to-mem forwarding, we add the forwarding path from memory stage to the input of the pipeline registers in execute stage, which is useful when we have a store instruction after load instruction, where the load is writing the value of the load instruction.

For dynamic branch prediction, we intended to increase the rate of branch predictions to increase CPI and improve performance. To achieve that goal, we store the previous action at that position of the branch, and use the record to predict the branch decision (if the previous action

was to take the branch, then we will take it again this time). Compared to the original design, dynamic branch prediction can reduce the number of prediction failures when we meet a for-loop that need to execute many times.

Design Analysis

Hazard table:

hazard	number of cycles to stall
RAW (read, 2 cycles after load)	0
RAW (read after load)	1
RAW (other kinds)	0
WAR	0
WAW	0

For the data hazards, we are implementing forwarding to improve performance. With the forwarding all the stalls due to hazards can be eliminated except the RAW on the instruction directly following a load instruction (the pipelines may still stall due to the cache).

For the cache, we eliminated the compare-read stage to simplify our design. There are 5 stages in total in our design. When read or write signal is asserted, if it is dirty and not hit, then we are going to memory writeback stage to write current data in cache to memory and fetch new content from memory. If it is (both not hit and not dirty) or not valid, we are going to cache writeback stage to fetch the correct content from memory. When finally write data back from memory to cache, we directly used the writeback data to replace the original data we read, which helped to make the state machine more efficient.

we are using two-cycle hit (comp -> COMP_WR -> COMP). Cache miss with eviction (COMP -> MEM_WB -> CACAE_WB -> FINISH -> COMP) needs 12 cycles, and cache miss without eviction (COMP -> CACHE_WB -> FINISH -> COMP) in our state machine needs 8 cycles.

Conclusions and Final Thoughts

In this project, we tried to implement a simplified microprocessor and were able to understand how the microprocessor works in more detail. It can help us to understand the details of microprocessors more deeply. During the implementation, we encountered many problems we might not see without implementing the microprocessor by hand. For example, it is hard to think of the conditions that logic loops occur if we do not implement it by hand, as many details such as the positions of pipeline registers. The connections between different modules can play an important role.

We have also learned how to build up a huge project from small pieces and add new functionalities to increase its performance. While we were doing the project, it is usually much harder than we thought and unexpected new problems usually come out.

If we could do this project again, we would make a more detailed plan about the structure of the whole design and how they should connect to each other. We know it is difficult but if we could make a detailed and more suitable design for the whole schematic, we can make the performance much better by not having the problems of old units “restricting” the space of optimization when approaching the end.