CS552 Final Project

Memory Leak
Sparsh Agarwal, Joanne Lee

Design Overview

The project is a comprehensive design and implementation of a microprocessor, created in Verilog. The microprocessor is pipelined, uses a multi-cycle memory with caches, and implements branch prediction as well as data bypassing. We started the project by implementing a very straightforward single-cycle microprocessor that involved a simple perfect memory and no pipelining, memory caching or any kinds of optimizations to decrease CPI needed for each instruction executed. Then, a pipelined design of the microprocessor was introduced to allow multiple instructions to be processed at the same time, greatly improving the speed of the microprocessor. After that, we incorporated two-way set-associative memory caches into our microprocessor's memory modules. We designed and implemented a memory controller that served as an interface utilizing the cache and main memory system. Finally, we added in optimizations that include data bypassing from the register file as well as between pipelined stages, and handled branches by predicting them to be not taken. The result was a working and very speedy microprocessor that only used stalling for memory accesses.

The microprocessor is broken down into five pipeline stages: Instruction Fetch, Instruction Decode/Register File Read, Execute, Memory Access, and Write Back. In the Instruction Fetch stage, we have the instruction memory system and a control unit module for the program counter that controls when and how new instructions are read from the memory, such as during a branch or jump instruction, or during a memory stall. In the Instruction Decode/Register File Read pipeline stage, the two important modules are the register file system and the operation control unit that decodes the instruction fetched to determine what control signals should be asserted. This stage also includes simple ALU modules used to determine whether branches and jumps are taken and to calculate new addresses in addition to a module to sign-extend immediates. In the Execute stage lives the main ALU for computations using values from the register as well as a ALU control unit that dictates what operations the ALU should execute. Next is the Memory Access stage where we have the data memory system to store and load data from memory, and in the Write Back stage, any data needing to be stored to the register file is written.

The memory controller system consists of a cache module and the main memory, and was implemented using a complex state machine. During a memory read, the memory controller will attempt to find the required data in the cache. If it fails to do so, it will write back any dirty data if needed and choose a way to evict, then proceed to fetch a line of data from the main memory that contains the requested data, write this to the cache and retry the memory read, now succeeding. During a memory write, the memory controller will look for the requested address in the cache, and if found, will write the new value and set the dirty bit. If not found, it will write back any dirty data, evict a line in the cache and replace it, then write the new data into the cache and set the dirty bit.

Optimizations and Discussions

The optimizations we implemented were data bypassing from the register file, data bypassing between pipelined stages, and the handling of branches by predicting them to be not taken. We used the bypassing register file we designed in one of the homework modules to allow data being written to the register file to be read during the same cycle, thus eliminating the need for unnecessary stalling. Data bypassing between pipeline stages was involved in almost every stage, such as from the Execute stage to Instruction Decode, Memory Access to Execute. The extensive bypassing efforts resulted in our microprocessor not utilizing any stalls except for memory stalls originating from the memory system. By predicting branches to be not taken, we designed the microprocessor to assume that any instruction involving branching will not take the branch and proceed as normal, but when a branch is determined to be taken, instructions that are not supposed to be executed is flushed from the pipeline.

Our final microprocessor successfully passed all tests and functions as required by the specifications (the ultimate test we only got to pass about an hour after the 5pm deadline), except for the extra credit tests involving SIIC and RTI instructions and exception handling. Given more time, we would have liked to complete the project to its fullest by implementing the SIIC and RTI instructions and handling exceptions.

Design Analysis

No hazards!	0

The cache and memory controller system, designed using a state machine, starts by immediately accessing the cache to determine a cache hit or a cache miss. On a cache hit, only one cycle is taken as the data can be read or written to immediately if found in the cache. During a cache miss requiring an eviction, 12 cycles are needed. One cycle is used for the initial read or write, four cycles for reading the dirty data from the cache and writing back to the memory, six cycles for reading the requested line of data from memory and writing to the cache, and one more to retry the request. On a cache miss without requiring an eviction, eight cycles are taken. One cycle for the initial read or write, six cycles to read the line from the memory and write to the cache, and one final cycle to retry the request.

Conclusions and Final Thoughts

It comes as no surprise that the project taught us, in detail and in an extremely hands-on approach, how every small module and part plays its part in such a complex computer system, the microprocessor. The project walked us through the gradual evolution of a simple single-cycle microprocessor to one utilizing pipelining and further optimizations. It highlighted the importance of efficient memory access, which tends to be extremely slow, using the memory hierarchy to exploit faster but smaller cache systems. The project also showed us how having a functional microprocessor was not enough and that great efforts had to be made, and continues to be made, to improve and optimize it for even better speeds and processing performance.

Regarding what could have been done differently, we believe we did the very best we could and are satisfied with what we have designed. Given a choice, we would not change the decisions we had made in the design.