CS552 Final Project

WISCSP13

Sanjay Rajmohan   Shyamal Anadkat

1.0 Design Overview

We designed and implemented a pipelined processor with multi-cycle memory with two-way set associative caches for instructions and data. The final demo built up through milestones throughout the semester.

The processor is pipelined into distinct stages which consist of Fetch, Execute, Decode, Memory, and Writeback. The fetch stage gets the instruction directed by the current PC from the instruction memory or the cache. The instruction is then passed to the next stage, which is decode. The decode stage is responsible for decoding the instruction based on the opcode and has a control unit which sets various control signals based on the instruction. The decode module also includes the register-bypass module. The execute stage, as it may sound, has an ALU for executing logical and arithmetic instructions based on the control signals. It is also responsible for address calculations for load/store instructions and branch instructions. Next, the memory stage is responsible for reading/writing to the data memory or the 2-way data set associative data-cache. Data that is passed as address can be forwarded to the execute stage if the next instruction is dependent on the instruction currently in the memory stage. The last stage is the writeback stage which will write back either the computed data or the data read from memory to the register file. Just like in the memory stage, the data to write in the writeback stage can be forwarded to the execute stage if there is dependence. We also implemented a branch prediction mechanism that predicts that a branch is not taken and only flushes the previous stages by passing nops if a branch is taken.

The memory system for the cache was designed in incremental stages. The first stage involved designing it for a direct-mapped cache, where we designed an FSM that checks if the address was a hit or miss, then either just returns the data from the cache or retrieves the data from memory, writes to cache and then returns it. The second stage involved implementing a 2-way set associative cache with LRU policy where the victim to evict was chosen based on flipping the way bit every time a line was accessed. This was similar to the first stage with the addition of another cache module.

2. 0 Optimizations and Discussions

– Brief discussion of optimization implemented (Maximum 0.5 pages)

Our optimized version of the final demo included the branch prediction, forwarding, as well as integration of the caches. This resulted in enhanced memory access time and improved slack time. We were also able to implement branch prediction and forwarding in our final demo. To test our branch prediction and forwarding, we wrote small tests that had a bunch of branch instructions that would not get executed and instructions dependent on the previous instructions. Our optimized design performed far better than our previous demo design. For addition operations, we use carry-lookahead adders instead of ripple carry adders, which reduces the time of execution but at the cost of more required gates to compute carry bits.

– Discussion about failures, if any (Required for partial credit):

All the tests except the exception tests passed, although we wish that we could have implemented them as well.

3. 0 Design Analysis

Hazard Type Or Cache Miss/Hit	**Stall cycles**
Control Hazard	2 cycles – branch prediction. If the processor detects that a branch or a jump is taken, it will send nops through the fetch and decode stages and send the computed PC to the fetch stage from the memory stage, and the instruction in the new PC will be executed in the next cycle.
Data Hazard	1 cycle – data dependency. When the processor is currently executing a load in the execute stage and the instruction in the decode instruction is dependent on the destination register of the load, it will stall for 1 cycle by sending a nop through the execute stage and then the written value is forwarded from the writeback after value is read from memory.
Hits for Instruction Cache	2-3 cycles. Our implementation is such that it will start at idle state if the current instruction is either a branch or the data cache is stalling to read. In this case, the number of cycles is 3. For other instructions, it will be 2 cycles as it starts at the compare read state and goes to done on a hit.
Hits for Data Memory Cache	2 cycles. In case of data cache, it does not have to do stalls for branches, so it will be 2 cycles for a hit.
Misses for Instruction Cache	Without Eviction: 11 (For Load), 12 cycles (For Store) With Eviction: 15 cycles (For Load), 16 cycles (For Store) Without eviction, the memory system will do a compare read, and when it's a miss, will do a memory read and store that in the cache. With an eviction, the memory system must write the values first and then do the memory read, which takes more cycles.
Misses for Data Cache	Without Eviction: 11 cycles

	With Eviction: 15 cycles This is very much like the instruction cache and hence has the same number of cycles, although it only does reads and not writes.

4. 0 Conclusions and Final Thoughts

- A conclusion outlining what you learned by doing this project and what you would have done differently. (Maximum half a page)

  The project was a great exposure to the world of computer architecture and processor design. By climbing different milestones throughout the project, we were able better able to understand how each stage in the processor works – single cycle unpipelined, pipelined as well as padded with multi-cycle memory and several optimizations such as branch prediction, forwarding, and 2 way set associative caches which would reduce the average memory access time. Integrating everything in the final demo was harder than we thought, and we should have started to work on it earlier. We were not too organized in writing the modules which made it harder to find and understand what might be causing issues, and would change that and not spend too much time figuring out what we wrote rather than figuring out the actual bugs.