An overview of a simplified five-stage pipeline CPU

To
Professor Mikko Lipasti
CS/ECE 552 Introduction to Computer Architecture

By
Peng Cheng, Yiming Liu, Yi Shen
Department of Electrical and Computer Engineering
University of Wisconsin-Madison
May 2nd, 2018

Overview

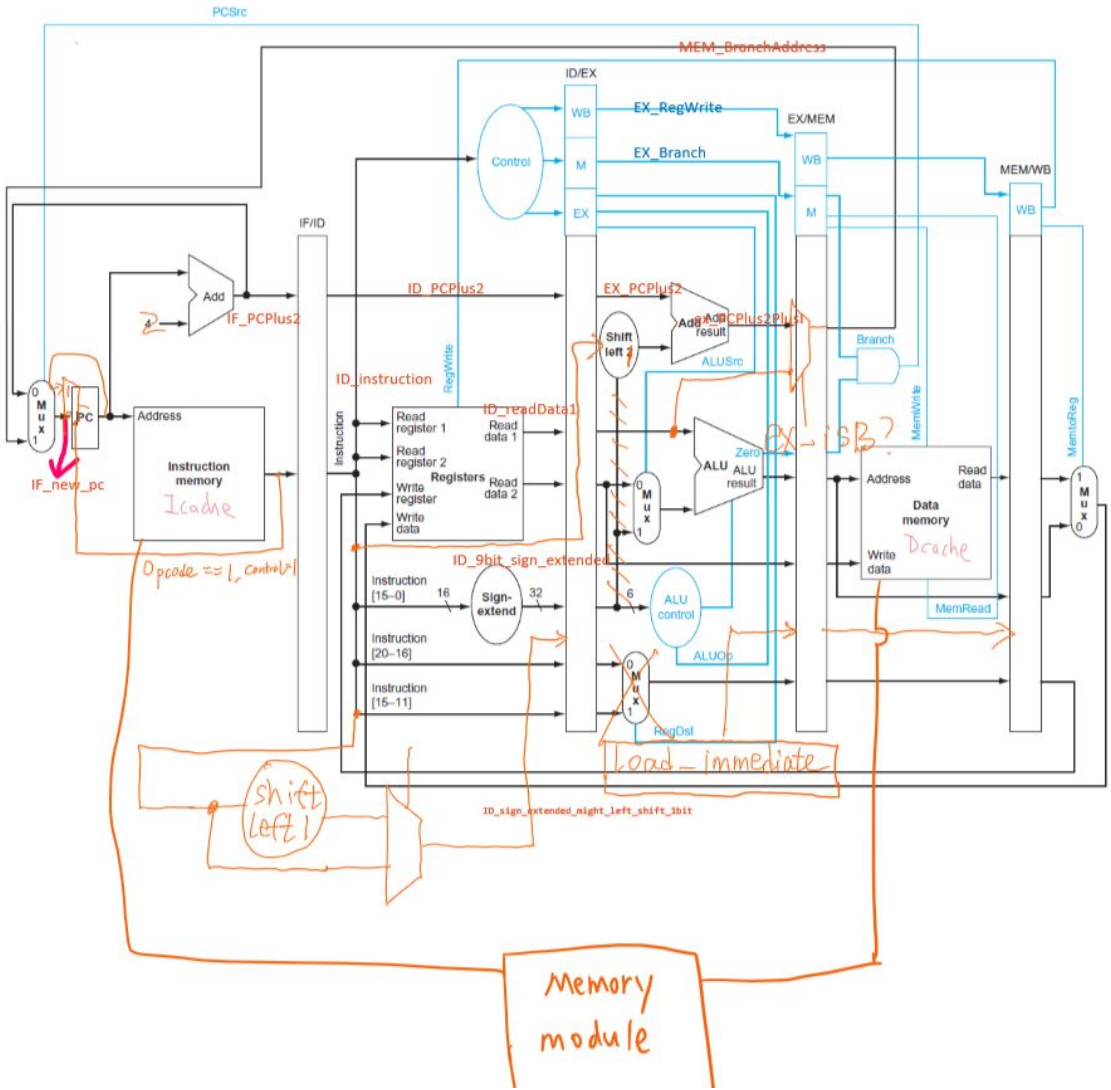
For this entire project, we accomplished the design and testing of a pipelined, cached, 5-stage microprocessor. This process has a bitwidth of 16 bits, cache and memory byte addressable. The processor uses instruction set WISC-S18.

In our design phase 1, we designed a basic single cycle MIPS like processor, it contained no support for data forwarding, stall, and memory module is single cycle.

For our design phase 2, we added pipeline into our phase 1 design, supported data forwarding and global stall signal. The process was able to run at a higher frequency, even though we didn't test it.

For our last phase, cache system was added. We no longer have 2 seperate memory module, one for instruction and one for data. They are each replaced with a cache module, and both connecting to the 4-cycle memory module.

The following is our final design diagram.



Special Features

In phase 2, a single cycle processor is upgraded to a five-stage processor, which also supports data forwarding and register bypassing. A typical scenario in a pipelined processor that demands data forwarding are known as register read after write (RAW) and register write after write (WAW). To figure out whether a register is modified and then being read or written before being written back, which is accomplished in the fifth stage in our design, we first need to check if an instruction involves register modification. In other words, if an instruction does not modify any register, it is unnecessary to do a hazard detection on it. Hence, based on the provided instruction set architecture (ISA) by professor Mikko Lipasti, we decode each instruction at IF/ID stage to save registers that will be modified in later stages to IF/ID pipeline stages. Three signals IF_RsExists, IF_RtExists and IF_RdExists are defined correspondingly in IF stage and also passed to later stages as input signal to hazard detection logic. Furthermore, we implement stall detection unit between IF stage and ID stage so that only IF stage will be stalled if data forwarding cannot be achieved. Besides, since branch unit is designed under assumption that branch will always always be taken, we flush all instructions that enter the processor after the branch instruction and all related control signals by writing zero to specific pipeline registers.

In phase 3, instructions and data is read from cache instead of memory when there is a hit. However, a single 2 bytes data or instruction fetching from memory costs four clock cycles and 16 bytes in total is put in a cache block after each misses. Considering eight two-byte data is read from memory, we decide to do a pipelined reading from memory to reduce cache miss penalty. Instead of sending a new memory address to memory after a valid data is returned, which means sending a address every four clock cycle, we send an address in each cycle for eight times. In this case, memory is able to manage sending valid data back to cache and taking addresses from cache concurrently. Consequently, delay on memory access reduces from 32 cycles to 12 cycles. Moreover, there might be a conflict between instruction cache and data cache if they compete for memory access simultaneously. An arbitration module is created to allow only one cache takes over the memory in a clock cycle and the other one is stalled accordingly. Data cache is rendered a higher priority in our design. If one of those two caches is denoted as busy fetching data, the other one cannot read from memory until the fetching is completed. Lastly, the whole processor is stalled if a cache miss occurs.

Completeness

Our design in phase1, phase2 and phase3 meets all the specified requirements. For phase3, we successfully passed all the provided four tests. During the demo, we successfully showed professor Lipasti and TA that our design functions as expected.

Testing

1. Analyzing the waveform and tracing the related signals are our main testing and debugging strategies. If we noticed that a certain signal was not generated correctly, we would check a set of signals that determines the erroneous one and analyze those more basic pieces first.
2. We used unit testing in phase 3 for the instruction cache and data cache; specifically, since we were given the memory unit in phase 2, we replaced part of the cache with the given one and see if other parts worked as expected and continued to do so until find the error.
3. We ran into a problem where a bunch of signals turned out to be high-Z. To solve it, we targeted the first signal that went to high-Z. Also, we sometimes gave those signals constants to see if other signals that depend upon it worked as expected.
4. For the fourth test in phase 3, each team member worked out the result of each instruction by hand and compared the results in registers with the contents from our design to help us find the bug.