Pipelined MIPS processor with stall on hazard (Assignment-8)

Sarthak Behera - 2018CS10384
Pratik Pranav - 2018CS10368

August 2020

Introduction

In this assignment, we implement the standard MIPS processor on a reduced ISA (Instruction Set Architecture) in C++ demonstrating effects of pipelining. Pipelining has some caveats like hazards which we handle in this assignment through stalls.

Implementation

Pipeline

The Pipeline class encompasses our implementation of the MIPS pipelined processor. This class contains five instances of objects belonging to the class PipeStage each of which represents one stage of our pipelining process. The objects instructionFetch, instructionDecode, executeInstruction, memoryAccess writeBack each are in accordance to one stage of pipeline. They, in turn, contain their own instruction to execute, its address, the registers to read and write, the values in registers to read and write and various control booleans like RegWrite, MemRead, etc.

The Pipeline class also contains field about the processor like clock which stores the number of clock cycles completed, isStalled which conveys if the pipeline is stalled due to a hazard so that instructionFetch waits until the pipeline gets free.

There are various methods to handle each stage of pipeline. The handleIF, handleID, handleEX, handleMEM, handleWB functions handle how the data in each stage is going to be passed on to the next stage. In each clock cycle we call each of these functions to do their respective job. In case of stalls jobs may not be completed but reinvokation in next cycle completes them. Their are also other header files like util.hpp which has utility functions to convert string to decimal, instruction_read.hpp, memory_read.hpp and as the name suggests read data from respective memory locations (files).

Stalling

In this implementation, we stall the pipeline in case of any hazard. Hazards maybe of three types:-

- **Structural Hazard** - Not considered here as instruction and data memory are different.
- **Data Hazard** - We stall the pipeline if we encounter these hazards. These hazards occur when instructionDecode stage has to read from same register that executeInstruction has to write to or when we are trying to read from a register in executeInstruction although writing to it in writeBack stage has not completed.
- **Control Hazard** - These occur when in case we have a branch or jump instruction. Jump instruction has to be identified in the instructionFetch stage itself as it next instructions may not be needed. Branch instructions have to be identified before executeInstruction stage so that we can stall the pipeline suitably.

The checkForStalls method checks for stalls in the pipeline so that we can convey it to each stage appropriately.

Testing

Rigorous testing was done through test cases testing a variety of hazards and stalls. The test cases were specifically chosen to see various functionality of stall works. The tests were verified with Google Test, a test framework for C++.