Michael Ing

ECE 2162 Computer Architecture

Professor Jun Yang

Due 13 December 2018

Tomasulo’s Algorithm Final

**Section 1: Execution Guide**

I chose to write this project in Java because of my familiarity with it, using Notepad++ and the command line because of their ease of use and accessibility. I wrote my code on a Windows machine using java jdk 11.0.1. The executable file is called TomasuloRunner.java, and can be compiled and run using the following commands from the terminal:

javac TomasuloRunner.java

java TomasuloRunner

Input is taken from a file (currently called “input.txt” but this can be changed on line 34 of TomasuloRunner.java), which is formatted similarly to the sample input file. The first line, showing the columns of RS, execution cycles, memory cycles, and functional units is ignored. The second line has the details for the integer adder, the third and fourth lines have the details for the floating-point adder and multiplier, and the fifth line has the details for the load/store unit.

Following that is a blank line, and then the seventh and eighth lines define the size of the ROB and the size of the CDB. Then there is one line that initializes register values in the ARF, and another line that initializes values in the memory.

After a final blank line comes the set of instructions which are read directly into the instruction buffer. The instructions are formatted exactly as specified in the project description and are only limited by the size of the ROB. An example of an input file is included at input.txt

All output is printed to a file (currently called output.txt) and is organized as specified in the project description. For debugging purposes, there is also a method in TomasuloRunner.java called printResults() that takes most of the data structures as well as a print array, which denotes the data structures to be printed and allows for easier organization by the programmer.

**Section 2: Test Benchmarks**

I created eight tests to check the usability of my code, each labeled “inputXXX.txt”, where XXX describes what the file specifically examines. These include inputNoDep, which is a base case without any data dependencies, inputTrueDep, which has a basic RAW data dependency, and inputFalseDep, which tests both WAR and WAW false dependencies. I also created inputLoadStoreForward, which tests the forwarding of data from stores to loads, inputStructHaz, which tests structural hazards in functional units, inputBrPred, which tests branch prediction and the BTB, and inputLoop, which loops through the base case of inputNoDep. My final test case was inputAddi, which simply tests the functionality of the ADDI instruction. Each of these files has its own associated output file entitled “outputXXX.txt” corresponding to the XXX above, and all input and output files are located in their respective folders. An example of the code generating output is shown with input.txt (which is a copy of inputNoDep.txt) generating output.txt. These are both located in the main project folder.

After extensive testing and troubleshooting, all the test cases function as desired, and the output generated in output.txt matches the appropriate output in the corresponding output file. The simplest test, inputAddi, simply runs three Addi commands and stores the proper register values. The inputNoDep test properly demonstrates the floating point and integer reservation stations as well as a simple load command, all without any dependencies. The inputTrueDep test delays the correct amount of cycles such that it executes one cycle after the result is written back, while the inputFalseDep file uses the ROB and the commit stage to handle false dependencies without having to delay cycles. The inputStructHaz and inputLoadStoreForward tests are rather self-explanatory, as the former demonstrates instructions delaying if reservation stations are full (structural hazards), and the latter displays store instructions forwarding values to load instructions so that memory doesn’t have to be accessed. The final two tests are inputBrPred and inputLoop. InputBrPred shows off both types of branch prediction (mis predicting a taken branch and mis predicting an ignored branch) functioning properly where incorrect instructions are fetched and squashed, and inputLoop loops over the inputNoDep test using branching and proper branch prediction. I believe this list of tests is mostly comprehensive and demonstrates my working system efficiently ordering and reordering instructions.

**Section 3: Project Breakdown/Teammate Evaluation**

I worked on this project by myself, and thus wrote all files and test benchmarks myself. I also worked through all debugging and developed all of the architecture.