CSEN 702 Project Report

Ahmed Khaled
49-4211
ahmed.gaberdiab@student.guc.edu.eg

Misk Mohamed
49-19599
misk.abdalla@student.guc.edu.eg

Ali Elserafy
49-10386
ali.elserafy@student.guc.edu.eg

Mohamad Yasser
46-0801
mohamad.mohamady@student.guc.edu.eg

Ahmed Ehab
46-3830
ahmed.tawfikmohamed@student.guc.edu.eg

The '***Tomasulo Algorithm***' is a powerful hardware tool to eliminate RAW hazards while also executing instructions in an out-of-order manner, increasing CPUs' throughput and efficiency. The main goal behind this project is to simulate this algorithm in code while making sure it performs the same way as intended.

Our implementation was all done using pure Java libraries and Object-Oriented Programming approach, helping us tackle the different hardware components separately as different objects who behave differently. We had *'Instruction'*, *'Register'*, '*Buffer*' and '*Reservation Station*' classes as well as the main '*Tomasulo*' class controlling all of those elements cohesively. And as our implementation accepts MIPS assembly instructions as an input, we had an '*Interpreter*' class that reads and scans the instructions from a '*Program.txt*' file present alongside the source directory folder.

We type our program(s) in the following manner:

```
L.D F6 32
L.D F2 44
MUL.D F0 F2 F4
SUB.D F8 F2 F6
DIV.D F10 F0 F6
ADD.D F6 F8 F2
```

(MIPS Program Example)

It follows a space-separated registers and a line-separated instructions style, and as you can tell, yes that's exactly the MIPS program that was introduced in *Lecture 11-12* and was used to demonstrate how the algorithm works, so we decided to use it as our main reference for our testing, implementation and debugging phases. Fortunately, our outputs were the same as the lecture's!

Digging deeper into how the code works, its functionality and how it was developed, we used the main *'Tomasulo'* class as our engine that runs and simulates the program. At first we ask the user to enter the number of reservation stations, load buffers and registers the system should have, and also ask them for the latency of each type of instructions keeping in mind that they already provided the MIPS program in the .txt file. We take all the inputs and initialize our system based on all of those inputs. Then we call the **`simulate()`** method. It's basically a perpetual loop that awaits the user to press any key to simulate a cycle (e.g. space key). During a cycle simulation it calls **`issue(int instructionNumberToBeIssued)`** to issue an instruction, executes it if possible by calling the **`execute()`** method and then writes it back if possible too by calling the **`writeBack()`** method. Let's see how each of those methods functions in order to serve the whole Tomasulo-based system.

- **`public int issue(int instructionNumberToBeIssued):`**

It takes as an input the number of the instruction to be issued in the instructions' queue and basically looks for free buffers according to the instruction type that wants to be issued *(Load/Store, ALU, etc..)* and if there is no empty spaces (a structural hazard) it returns *-1*, and if there is a place it issues that instruction by placing it in that specified buffer/reservation station and marks it as busy holding this instruction. The default return of that function is *0*.

- **`public void execute():`**

It loops through all of the buffers/reservation stations decrementing the remaining execution time if there are any executing non-waiting instructions.

- public void writeBack():

It also loops through all of the buffers/reservation stations searching for any instructions that are ready to write-back after finishing execution in the previous clock cycle, and if it found any, it basically publishes the result on the CDB for any other entities needing it so that they can grab it and start their execution if possible.

Those were the main code structures for our Tomasulo algorithm, but a really important method too is the **`print()`** method that displays how these instructions are executed as well as the content of each reservation station/buffer, the register file and the instructions queue step by step and cycle by cycle as required.

```
Clock Cycle Number = 1

***INSTRUCTIONS QUEUE***
Instruction                Issue        Start       Finish      Write-Back
_____________________________________________________________________________

0.  LOAD   F6   32       |    1     |            |                |

1.  LOAD   F2   44       |          |            |                |

2.  MUL    F0   F2   F4       |        |          |                |

3.  SUB    F8   F2   F6       |        |          |                |

4.  DIV    F10   F0   F6       |        |          |                |

5.  ADD    F6   F8   F2       |        |          |             |

***LOAD/STORE BUFFERS***
Name          Busy       Addr       FU       Time
_____________________________________________

LOAD1    |     yes     |     32    |         |   2

LOAD2    |     no      |              |         |

STORE1    |    no      |            |          |


***RESERVATION STATIONS***
Name         Busy        Op        Vj       Vk       Qj       Qk     Time
_____________________________________________________________________________
ADD1    |    no    |               |         |        |           |          |

ADD2    |    no    |               |         |        |           |          |

MUL1    |    no    |               |         |        |           |          |

MUL2    |    no    |               |         |        |           |          |

***REGISTER FILE***
F1
_______
|            |
|______|

***NOTES***
-> The instruction number: 0 has been issued at Load Buffer= LOAD1
```

(CLI Output Example)

All codes are to be found in the ZIP folder