**EECE 421 Project Report**

Implemented simulator: Tomasulo’s algorithm

Programming Language: C++

Implemented by: Dany Solh

**Introduction:**

In this project, Tomasulo’s algorithm was implemented using the C++ programming language and was divided into multiple components controlled by a central process. The project consists of two major folders: the ***include***folder which contains the header files (.h) that are responsible for declaring each component with its attributes and methods, and the ***src*** folder which contains the source files (.cpp) that are responsible for defining the methods of each component and include the central process that controls all the components. The **Processor.cpp** controls all the components of the simulator but is launched from the **main\_code.cpp** file. The ***build*** folder contains sample codes to test the simulator and the executable program ***my\_program.exe***, compiled and built.

**Running the simulator:**

The application should be already compiled and ready to run by executing ***my\_program.exe*** in the command prompt or terminal. Make sure to be in the executable’s directory or the project’s directory. If in the first, simply type in the terminal: **./my\_program.exe**. If in the second, type in the terminal: **./build/my\_program.exe**.

Once the application starts running, you will be prompted to enter the path of a **.txt** file that should contain the code to run in the simulator. To avoid any issue, it is best to enter the full path of the file but if you enter a valid path, the simulator should be able to find the file and start execution; otherwise, you will be prompted to enter the path again.

Once the simulator completed the execution of all instructions, the output metrics will be printed out. These include:

* The total number of clock cycles,
* The final register file (if you want to check the operations are done correctly),
* The average number of instructions per clock cycle (IPC),
* The average reservation station occupancy as a percentage of total cycles executed,
* The load/store buffer utilization as percentage of total cycles executed,
* The number and percentage of stall cycles due to structural hazards.

You will also be given the choice of running another code or leaving the simulator. To leave the simulator, enter **QUIT** when prompted.

If you need to recompile the application, you can either:

* Enter the following command in the appropriate terminal: **g++ -Iinclude -o build/my\_program src/main\_code.cpp src/Issue\_Unit.cpp src/Register\_File.cpp src/Register\_Status.cpp src/Functional\_Unit.cpp src/Memory\_Unit.cpp src/Registration\_Entry.cpp src/Registration\_Station.cpp src/Processor.cpp**
* Run the provided Makefile by entering the command **make** in the appropriate terminal. Make sure to delete the already existent executable file in the ***build*** folder and the content of the ***obj*** folder for the command to run successfully.

**Notes and points to consider:**

1. As there is only one set of registers in this implementation, the register file stores only integer operands.
2. The value of each register is initialized to its index. For example, R0 has an initial value of 0, R1 has an initial value of 1, etc.
3. For simplicity, the memory unit was designed as word-addressable and can store 1024 words (in other words, has a size of 4KB).
4. The memory was implemented as an array of 1024 elements, each representing a word. Hence, when computing an address in a LOAD or STORE operation, it will be the index of a word in memory obtained by the sum of the positive integer immediate and the value in the register specified in the instruction.
5. All the words in the memory are initialized to 0.
6. Make sure not to include empty lines in the code passed to the simulator to avoid any issue during instruction decoding.
7. By mistake, I defined reservation stations as **registration** stations, but their operation is not affected and is as expected.

**Operation of simulator:**

The simulator follows a four-step process:

1. It checks for any reservation stations whose operands are ready and have not yet been passed on to a functional unit. If such stations are found and the associated functional unit is available, the operands and other control signals are passed on to the functional unit.

For load and store buffer, an extra verification is needed for the addresses. We need to make sure that a load instruction that targets the same address as a previous store instruction doesn’t execute before the store, and vice versa, to ensure consistent memory updates. This is mainly done by having an address age variable that is constantly updated and used to control the order of memory operations.

1. If any functional unit completes execution, we write back its result to any reservation station that depends on it and to the appropriate register in the register file (determined by the register status unit). Only one functional unit can write on the Common Data Bus during a clock cycle.
2. If a functional unit is busy and hasn’t completed execution yet, we update its status: a counter that is incremented and compared to the number of clock cycles needed to complete the executed operation.
3. If we haven’t issued all instructions yet, we will decode the instruction we are currently at and attempt to assign it to an appropriate reservation station. If no reservation station is available, we will have to stall the fetching and decoding of instructions and will try again at the next clock cycle to find a reservation station for the current instruction.

However, if all instructions were issued, we will verify if all of them were executed by verifying that all reservation stations and load/store buffer are empty and exit the simulation if that is the case.

These four steps will be repeated until the code is completely executed. The diagnostics of the program’s execution will be displayed directly after it.