CSCE 3301 – Computer Architecture
Fall 2023

Project 2: femTomas
Tomasulo Algorithm Simulation

Project Overview: The goal of this project is to implement an architectural simulator capable of assessing the performance of a simplified out-of-order 16-bit RISC processor that uses **Tomasulo's algorithm without speculation**. This document details the instruction set to be supported, the inputs to the simulator, and the expected outputs

Implementation language: Any general-purpose programming language. The resulting application can either be a console application or a graphical user interface (GUI) application (desktop, web-based, or mobile app) as a bonus feature.

Instruction set architecture (ISA): The simulator assumes a simplified RISC ISA inspired by the ISA of the Ridiculously Simple Computer (RiSC-16) proposed by Bruce Jacob. As implied by its name, the word size of this computer is 16-bit. The processor has 8 general-purpose registers R0 to R7 (16-bit each). The register R0 always contains the value 0 and cannot be changed. Memory is **word addressable** and uses a 16-bit address (as such the memory capacity is 128 KB). The instruction format itself is not very important to the simulation and therefore is not described here. However, the simulator should support the following set of instructions (16-bit each):

1. **Load/store**
   - Load: Loads a word from memory into `rA`. Memory address is formed by adding `offset` to contents of `rB`, where `offset` is a 6-bit signed constant (ranging from -32 to 31).
     - `LOAD rA, offset(rB)`
   - Store: Stores value from `rA` into memory. Memory address is computed as in the case of the load word instruction
     - `STORE rA, offset(rB)`
2. **Conditional branch**
   - Branch if not equal: branches to the address `PC+1+offset` if `rA!=rB`. Note that the PC is incremented by one when the branch is not taken.
     - `BNE rA, rB, offset`
3. **Call/Return**
   - Call: Stores the value of `PC+1` in `R1` and branches (unconditionally) to the address specified by the `label`. The label is also encoded as a 6-bit signed constant.
     - `CALL label`
   - Return: branches (unconditionally) to the address stored in `R1`
     - `RET`
4. **Arithmetic and Logic**
   - Add: Adds the value of `rB` and `rC` storing the result in `rA`
     - `ADD rA, rB, rC`
   - Add immediate: Adds the value of `rB` to `imm` storing the result in `rA` where `imm` is a 6-bit signed constant.
     - `ADDI rA, rB, imm`
   - Nand: Performs a bitwise Nand operation between the values of `rB` and `rC` storing the result in `regA`
     - `NAND rA, rB, rC`
   - Divide: Divides the value of `rB` by `rC` storing the result in `rA`.
     - `DIV rA, rB, rC`

Simulator inputs:

1. The assembly program to be simulated. The user should also specify its starting address. Instructions can be entered one by one or even selected from a list.
2. Data required by the program to be initially loaded in the memory. For each data item both its value and its memory address should be specified. All values are assumed to be 16-bit values.

Simulation: The simulator should simulate the program's execution on a **single-issue** processor (multiple-issue simulation is a bonus feature). It should follow the non-speculative version of Tomasulo's algorithm. The simulator should only take the **3 backend stages** into account: issue (1 cycle), execute (N cycles depending on the functional unit involved as detailed in the table below), and write (1 cycle).

You can assume that all instructions have already been fetched and decoded and are waiting to be issued. The following is also assumed about the functional units available:

	Number of reservation stations available	**Number of cycles needed**
`LOAD` unit	2	1 (compute address) + 2 (read from memory)
`STORE` unit	2	1 (compute address) + 2 (writing to memory)
`BNE` unit	1	1 (compute target and compare operands)
`CALL/RET` unit	1	1 (compute target and return address)
`ADD/ADDI` unit	3	2
`NAND` unit	1	1
`DIV` unit	1	10

For conditional branches, the processor uses an always not taken predictor. Recall that in the non-speculative version of Tomasulo, instructions can only be issued (but not executed) based on a prediction.

While simulating the execution, the program should record the number of instructions completed, the number of branches encountered, the number of cycles spanned, and the number of branch mispredictions.

Simulator output: At the end, the simulator should display the following performance metrics:

1. A table listing the clock cycle time at which each instruction was: issued, started execution, finished execution, and was written
2. The total execution time expressed as the number of cycles spanned
3. The IPC
4. The branch misprediction percentage

Simplifying assumptions:

1. Fetching and decoding take 0 cycles and the instruction queue is already filled with all the instructions to be simulated.
2. No floating-point instructions, registers, or functional units
3. No input/output instructions are supported
4. No interrupts or exceptions are to be handled
5. For each program being executed, assume that the program and its data are fully loaded in the main memory
6. There is a one-to-one mapping between reservation stations and functional units. i.e., Each reservation station has a functional unit dedicated to it
7. No cache or virtual memory

***Bonus* features:**

1. Building the application as a GUI application (desktop, web-based, or mobile app)
2. Building the application as an educational GUI application. In this case the user should not only be allowed to simulate entire programs at once and get performance metrics but also to step through the program cycle-by-cycle while monitoring the internals of the processor (the status of all reservation stations, the register status table, the register file, and all memory locations holding relevant data values). This feature is also helpful for debugging purposes and will be **counted as two features** as far as grading is concerned (since it includes and extends the previous bonus feature).
3. Support a variable hardware organization. This bonus too will **be counted as two features** as far as grading is concerned. If implemented the user should specify the number of reservation stations for each class of instructions. Additionally, the user will specify the number of cycles needed by each functional unit type.

4. Support a multiple-issue hardware organization. This bonus too will be **counted as two features** as far as grading is concerned. If implemented the user should specify the pipeline width.
5. Implementing and integrating a parser to allow the user to provide the input program using proper assembly language including labels for jump targets and function calls. The assembly program can be entered directly in the application or in a text file read by the application.
6. Implement **two** dynamic branch prediction algorithms and compare their effects on the performance.

Important Plagiarism notice: You have to write your own code from scratch. Submissions based on others code will receive a grade of **zero** in the entire project (even if you understand every code line and even if the code is heavily re-factored/modified). Examples of such sources include (but is not limited to) code coming from the following sources: other teams, previous course offerings, open-source software, tutors, Internet, etc.

Deliverables and Deadline: There are no milestones for this project. You are required to submit the full project at once on **Monday December 11, 2023 (11:59 pm)**. Please note that this is the last day of classes, and no extensions will be granted beyond that date.

Grading Criteria

- As already announced, both projects (project1 and project2) collectively are worth 40% of the course marks. This project will count for 15% of the course marks while project 1 will count for 25%
- Bonuses: Each bonus feature will count for 5% with a maximum of 2 bonuses worth 10%. Please do not be tempted to implement more than 2 bonus features, as this will cost you too much time.
- Deductions:
    - -5% for not following the required submission directory structure.
    - -5% for late submissions (1 day only).
    - -100% for plagiarized submissions.

General Guidelines

- Work in the same teams as for Project1.
- Every group member must log all of her/his activities in a journal (a text file). A journal entry may look like the following:
  `Dec 5, 9:00PM: implemented a function to detect instructions dependencies.`
- Final project must be submitted by the group leader as a single zip file including the following:
    - A readme.txt file that contains student names and IDs as well as release notes (issues, assumptions, what works, what does not work, etc.,)
    - Journals folder: has the group journals
    - Source Code folder: Simulator code in the general-purpose language of your choice
    - Test folder: The assembly programs you used to test your program (and associated data if any). You should at least provide 3 programs. The programs must cover all instructions supported and one of them at least must have a loop.
    - A report file including:
        - A brief description of your implementation including any bonus features included
        - A user guide including a full simulation example step-by-step with snapshots.
        - The results obtained from the simulation of each assembly program provided.
        - A brief discussion of the obtained results.
        - Optionally, you can include a section about your experience working on this project. This section will NOT affect your grade in any way.