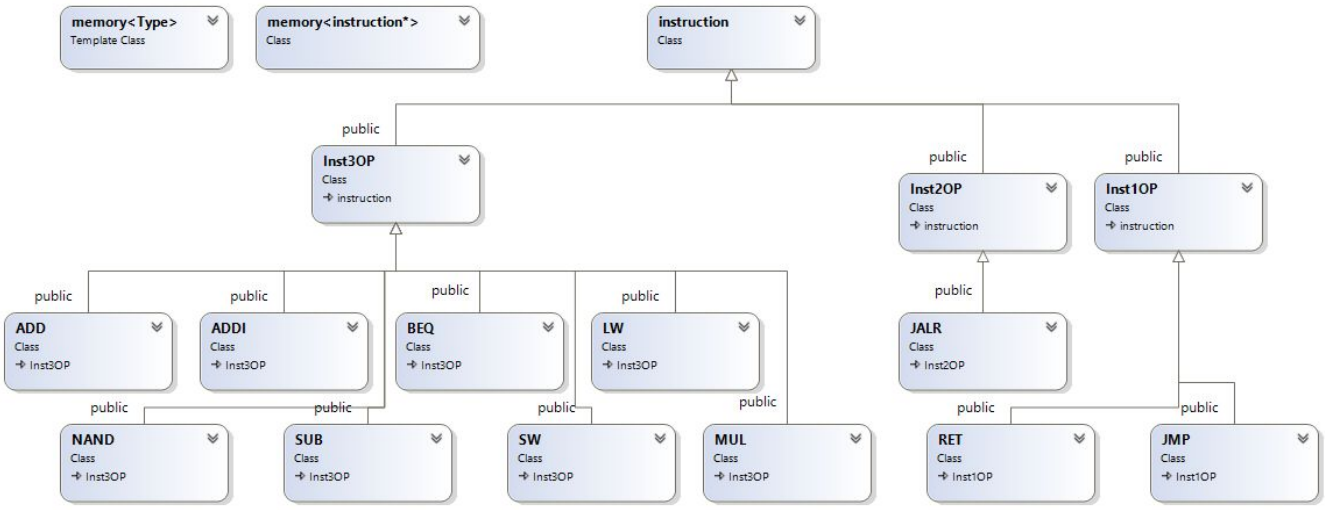
Project 2: Tomasulo's Algorithm Simulation

Final Milestone 1



Laila N ElKoussy
900160812
Yahia Khaled Farid
900161331
Yousef Mohab Koura
900160083

Table of Contents

Introduction

The Central Processing Unit (CPU) is the heart of any electronic device. They vary in types, functions, and design, but their efficiency mainly lies in how many instructions they can execute in a certain given time. The most popular kind of design for a CPU is a pipelined CPU. Unfortunately, some instructions, like floating point instructions, can lag the whole pipeline, and therefore hinder the performance. There is some approach that allows us to limit the effect of that; which is having a superscalar out-of-order CPU. In this project, we will be simulating simplified superscalar out-of-order 16-bit RISC processor that uses Tomasulo's algorithm with speculation.

Our Simulator

In this milestone, our simulator does not implement Tomasulo's algorithm. It only takes the input and parses the content in the corresponding containers. Our simulator follows a Harvard architecture, which means it has separate 128 KB memories for the data and the instructions.

Interface

Our interface first prompts the user to enter the starting address in hexadecimal. After this, it asks the user to choose between two inputting modes.

The first mode is for the user to enter instruction by instruction manually. The second is to read directly from a text file. After taking the input, the simulator echoes it so that the user can make sure that the instructions entered are correct. If taking input from a file, the file needs to specify the .text and .data sections. It is allowed to have multiple .text and .data sections, but it is imperative that the indicators be there.

Instruction Set Architecture and Format

- Load word: Loads value from memory into regA. Memory address is formed by adding imm with contents of regB, where imm is a 7-bit signed immediate value (ranging from -64 to 63).
  `LW regA,regB,imm`
- Store word: Stores value from regA into memory. Memory address is computed as in the case of the load word instruction.
  `SW regA,regB,imm2.`
- Jump: branches to the address PC+1+imm
  `JMP imm`
- Branch if equal: branches to the address PC+1+imm if regA=regB
  `BEQ regA,regB,imm`
- Jump and link register: Stores the value of PC+1 in regA and branches (unconditionally) to the address in regB.
  `JALR regA,regB`
- Return: branches (unconditionally) to the address stored in regA
  `RET regA`
- Add: Adds the value of regB and regC storing the result in regA
  `ADD regA,regB,regC`
- Subtract: Subtracts the value of regC from regB storing the result in regA
  `SUB regA,regB,regC`
- Add immediate: Adds the value of regB to imm storing the result in regA
  `ADDI regA,regB,imm`
- Nand: Performans a bitwise NAND operation between the values of regB and regC storing the result in regA
  `NAND regA,regB,regC`
- Multiply: Multiplies the value of regB and regC storing the result in regA
  `MUL regA,regB,regC`

Code

Our code so far only takes input from the user and places it in the instruction memory and data memory. But, we have a whole skeleton ready for the second milestone. Since we have multiple classes and an inheritance tree, we have appended our class diagram (appendix 1).

Our main function is the one in charge of the user interface. The simulator is in its own class (`SIM.h`). It's the one containing all of the memories, the program counter, and it is the one mediating between the instructions and handling the flow of the program. Our memories are word-addressed so they're indexed using 64KB, not 128KB. Our memory class is a template class (`Memory.h`), we use it for both the instruction and data memory.

In order to be able to have one holder for all types of instructions, we have created an instruction class (`Instruction.h`) that serves as a base-class for our different types of instructions. Then, we have three classes inheriting from it, Inst1OP, Inst2OP, Inst3OP, each with a header file using the same names. These three classes are designed to separate between the number of operands in the instructions. After this, we have a dedicated class for each instruction, each inheriting from its corresponding operand class.

Note: The basic design for this project is influenced by an old design for a Simple Integer Machine done by the team-members. Some functions, such as the ones in charge of input-handling, are modified versions of functions from the old design made to fit the requirements of this project.

Appendix

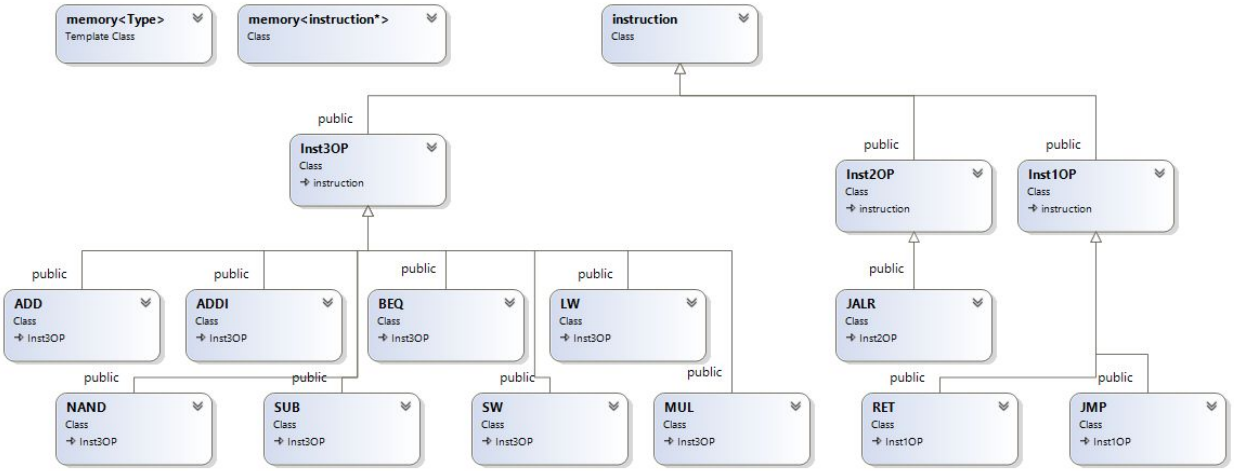


Figure 1: Class Diagram