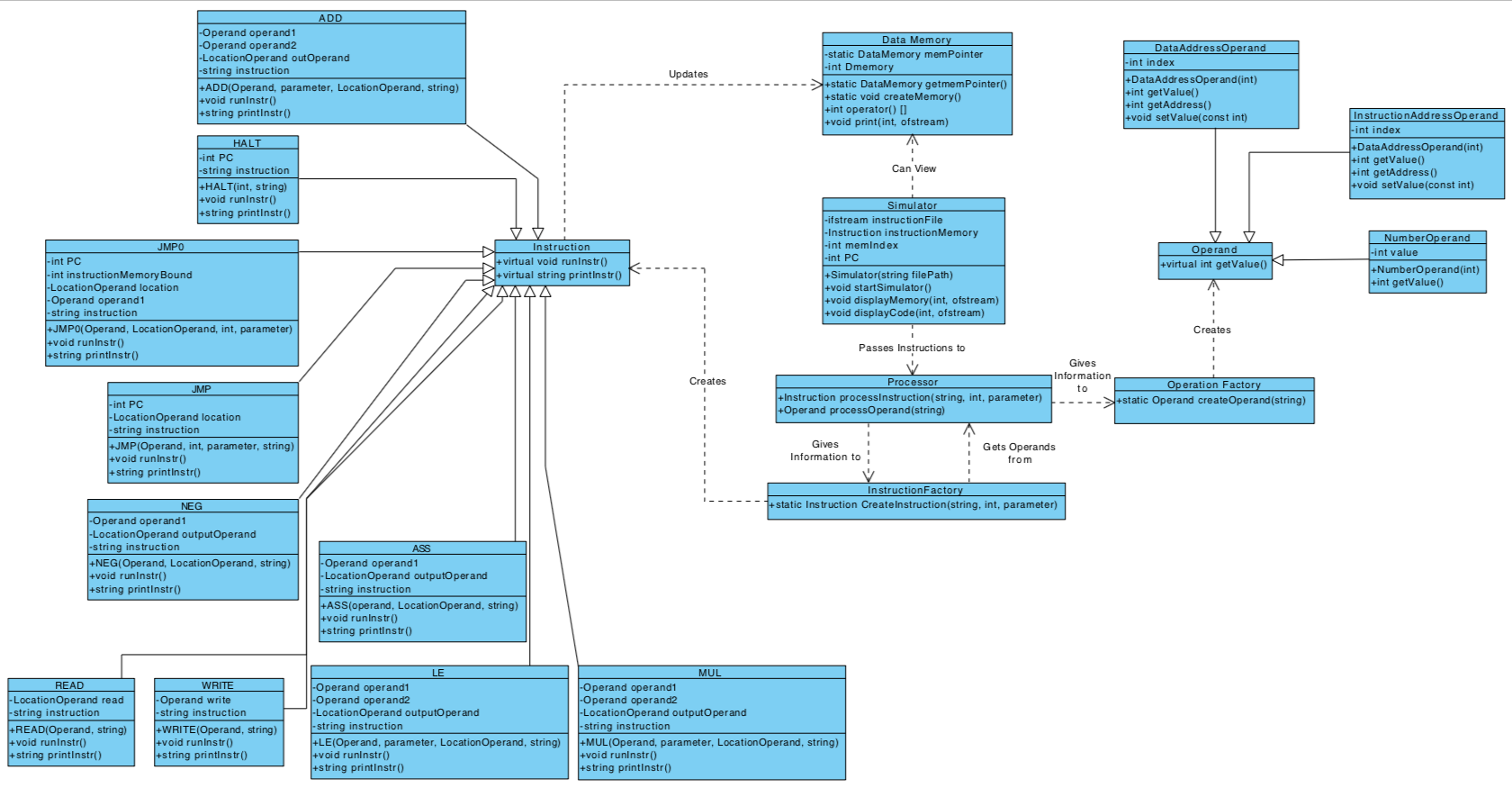
Object Oriented Programming

Assignment 2 Report

Malak Sadek

900140107

**Class Diagram:**

**Classes Explanation**:

**Operand.h**

*Operand* class is an abstract class that is defined as a parent for *DataAddressOperand*, *InstructionAddressOperand* and *NumberOperand* so that the user may enter values directly using as numbers without worrying about which class to use. It defines the abstract function *getValue*, which all children then override.

**OperandFactory.hpp**

**OperandFactory. cpp**

The *OperandFactory* class is responsible for figuring out whether an operand is an *InstructionAddressOperand* (instruction memory address), a *DataAddressOperand* (data memory address) or a *NumberOperand* (a normal number) based on the type passed to it from the *Processor*. It then creates a new instance of the needed type of operand and gives it to the *InstructionFactory* to make an instruction.

**InstructionAddressOperand.hpp**

**InstructionAddressOperand.cpp**

*InstructionAddressOperands* inherit from operand class and define their own *getValue*, since they represent the values stored in instruction memory locations, they also have a *setValue* function and a *getAddress* function.

**DataAddressOperand.hpp**

**DataAddressOperand.cpp**

*DataAddressOperands* inherit from *Operand* class and define their own *getValue*, since they represent the values stored in data memory locations, they also have a *setValue* function and a *getAddress* function.

**NumberOperand.hpp**

**NumberOperand. cpp**

*NumberOperands* inherit from *Operand* class and define their own *getValue*, since they represent normal numbers, they do not have any other functions.

**DataMemory.hpp**

**DataMemory. cpp**

The *DataMemory* class represents the data memory that can store 1024 ints

**Instruction.h**

This abstract class was created as a parent for all the classes representing instructions, it defines abstract functions for *runInstr* and *printInstr* and the children all override them.

**InstructionFactory.hpp**

**InstructionFactory. cpp**

The instruction factory is responsible for receiving an instruction opcode from the *Processor* class and figuring out which kind of instruction it is and making an instance of that class.

**Processor.hpp**

**Processor.cpp**

The *Processor* sets the instruction's opcode and extracts all the operands, based on the instruction string. It then passes the information to the *InstructionFactory* to create the appropriate instruction. Additionally, it also identifies whether an operand is a *NumberOperand,* a *DataAddressOperand* or a *InstructionAddressOperand*. The regex class is used to determine the type of operand since addresses for the data memory will start with the character '$' and addresses for the instruction memory will start with the character ‘#’. And then passes it onto the *OperatorFactory* to create the appropriate operator.

**Simulator.hpp**

**Simulator. cpp**

This class represents the simulator itself. It creates the instruction memory as a 1024 element array of instructions and opens an instruction file and begins giving the instructions to the *Processor* to determine their class and then runs them.

**ADD.hpp**

**ADD. cpp**

The *ADD* class represents the add instruction. It inherits from *Instruction* and defines its own *runInstr* and *printInstr* functions taking two *NumberOperands* or *DataAddressOperands* for inputs and a *DataAddressOperand* output.

**ASS.hpp**

**ASS. cpp**

The *ASS* class represents the assign instruction. It inherits from *Instruction* and defines its own *runInstr* and *printInstr* functions taking two *NumberOperands* or *DataAddressOperands* for inputs and a *DataAddressOperand* output.

**HALT.hpp**

**HALT. cpp**

The *HALT* class represents the halt instruction. It inherits from *Instruction* and defines its own *runInstr* and *printInstr* functions. It simply sets the program counter to the last data memory location (1024) so that the program stops.

**JMP.hpp**

**JMP. cpp**

The *JMP* class represents the jump instruction. It inherits from *Instruction* and defines its own *runInstr* and *printInstr* functions taking an *InstructionAddressOperand*, and setting the program counter to that address.

**JMP0.hpp**

**JMP0.cpp**

The *JMP0* class represents the jump if = 0 instruction. It inherits from *Instruction* and defines its own *runInstr* and *printInstr* functions taking a *DataAddressOperand* or *NumberOperand* and a *InstructionAddressOperand* and if the first operand = 0 then it sets the PC to the second operand.

**LE.hpp**

**LE. cpp**

The *LE* class represents the less than or equal instruction. It inherits from *Instruction* and defines its own *runInstr* and *printInstr* functions functions taking two *NumberOperands* or *DataAddressOperands* and a *DataAddressOperand* output. If the first operand <= the second operand then it sets the output operand to 1 (else 0).

**MUL.hpp**

**MUL. cpp**

The *MUL* class represents the multiply instruction. It inherits from *Instruction* and defines its own *runInstr* and *printInstr* functions taking two *NumberOperands* or *DataAddressOperands* and a *DataAddressOperand* output.

**NEG.hpp**

**NEG. cpp**

The *NEG* class represents the add instruction. It inherits from *Instruction* and defines its own *runInstr* and *printInstr* functions taking a *NumberOperand* or *DataAddressOperand* as an input and a *DataAddressOperand* output that it sets to the negative of the first operand.

**READ.hpp**

**READ. cpp**

The *READ* class represents the read instruction. It inherits from *Instruction* and defines its own *runInstr* and *printInstr* functions taking a *DataAddressOperand* and input from the user and placing it in that data memory address.

**WRITE.hpp**

**WRITE.cpp**

The *WRITE* class represents the write instruction. It inherits from *Instruction* and defines its own *runInstr* and *printInstr* functions taking a *DataAddressLocation* and outputting its data value.

**Main.cpp**

This tests the simulator. It creates an instance and passes it the paths to 9 different test files in a for loop, outputting the result of each test file to the console.

------------------------------------------------------------------------------------

**Validation Performed:**

*\*All validation is performed in the Simulator.cpp and Processor.cpp files*

*\*\*If an error is detected, the simulator does not continue with the current instruction set, it starts the next one*

1. If a data memory index is outside the range of the data memory.
2. If an instruction memory index is outside the range of the instruction memory.
3. If jump or jump0 do not receive an instruction memory address.
4. If jump or jump0 receive an instruction memory address larger than the total number of instructions.
5. If ADD, ASS, LE, MUL, NEG, READ, or WRITE receive an output operand that is not a data memory address.
6. If the instruction has for example “4$” instead of “$4” or “4#” “#4” or a character instead of a number, for example “ADD 3, a, $5”
7. If the input of READ is not a number
8. If the instruction is not one of the required instructions.
9. If the instruction file could not be opened.

10.If there is a syntax or execution error in an instruction

------------------------------------------------------------------------------------

**How to use code:**

* Data memory addresses use ‘$’, instruction memory addresses use ‘#’, there is a space between the instruction and operands but no space between operand and comma,
* **Eg. ADD $1, 3, $10**
* Running main.cpp creates an instance of the simulator which runs 9 tests provided by text files and prints them in the console.
* To run the code simply place all the header files and test files in your project and run main.cpp.
* Exceptions will be generated if there is anything wrong with the test files or with the instructions inside them except for infinite loops.
* **Please change the directory for the Instruction text files to your directory in main.cpp on line 16.**
* **Please note that InstructionSet7.txt (test #7) has wrong instructions on purpose to test that exception handling works.**
* **If you want to use your own test cases, remove the for loop in main.cpp on line 14 and add the directory of the text file to string s.**

------------------------------------------------------------------------------------------------

Thank you!