Computer Hardware Simulation (Java)

Project Overview

The project simulates the basic modeling of a computer's hardware/software architecture. The goal being to model a computer with the capability to create instructions, store them, decode, and process accordingly, storing the results. The classes used to model the design are a *bit* class, *longword* class, *RippleAdder* class, *Multiplier* class, *ALU* class, *Memory* class, *Assembler* class, and a *Computer* class.

The *bit* class is the first class and the starting point, used to represent an ordinary bit. A *bit* can hold the 0 and 1 values and be used to compute the logic-gate operations between each other. With the *bit* we create the *longword* class, which alongside the *bit* class, make up the primary means to create and store instructions. They are also the datatypes that make up the means to carry out arithmetic operations.

The *RippleAdder* and *Multiplier* classes are used to easily carry out arithmetic operations (addition, subtraction, multiplication), using *longword*'s and *bit*'s for the calculations. The *ALU* class is our arithmetic logic unit, made from the *RippleAdder* and *Multiplier* classes, and covers all arithmetic and logic-gate operations. It uses *bit* op-codes to specify the operations.

The *Memory* class is created from the *bit* class and serves as our *Computer*'s memory for storing our instructions. It can be configured to any size or used as a default size.

The *Assembler* class is used to more conveniently create instructions as *longword*'s from proper English-written commands for the *Computer*. It takes the English instructions and assembles them into a *longword* instruction with the proper operation op-codes, operation values, and storage locations.

The *Computer* is our final class and the main structure of the project. It is constructed from all of the smaller classes, containing a *Memory*, *ALU*, *Assembler*, and a variety of *bit*'s and *longword*'s used for key functions. The *Computer* also has an extra set of memory (not a *Memory* object), being an array of sixteen *longword* registers.

The *Computer* class's architecture is designed to function in a manner where:

- If the *Computer* is set to run (not be halted), then until it is halted, it will continuously execute a loop where it will **fetch**, **decode**, **execute**, and **store** the instructions and their values from the *Memory*. The instructions are executed in a singular sequential manner by treating the *Memory* as a stack and using a stack-pointer and program-counter (represented as *longword*'s) to designate where we are in the stack and instruction queue.

- For every loop, there will be a current-instruction, The current-instruction will go through the **fetch**, **decode**, **execute**, and **store** phases where it will be, as the names suggest: fetched, decoded for what type of instruction it is and values it is to use, executed accordingly, and stored accordingly.

Note: The decode, execute, and store phases should not be treated in a strict literal sense, as some instructions are designed to be handled where the decoding/execution/storage might take place within a different phase.

- The *Memory* is used to hold the entirety of the *Computer*'s instructions to be handled, whereas the sixteen *longword* registers are used to hold the data necessary for the computations as the instructions are executed.

- The designed instructions cover a small yet fundamental group of functions that would be necessary for a computer with the ability to handle computations and unique designed functions.

  The *Computer*'s instructions include the abilities to:
  - Move values into registers
  - Compute and store the results of arithmetic and logical operations
  - Halt the *Computer*'s run cycle of fetch, decode, execute, store
  - Print the contents of the registers and *Memory* to the console
  - Make comparisons between values to evaluate qualities of <, <=, ==, >=, >, !=
  - Branch control flow according to premises (if-statements)
  - Jump forwards and backwards around the *Memory* stack to execute instructions

- Finally, the *Computer*'s instruction handling properties, for handling the instructions in a sequential, queue-like, stack-like manner is managed and preserved by a combination of both the **fetch** phase of the *Computer*'s loop, and a group of 5 types of instructions: **jump**, **push, pop, call, and return**. The fetch phase retrieves the instructions for the *Computer* sequentially by incrementing the program-counter pointer to take the next ahead instruction each loop of the cycle. The jump/push/pop/call/return instructions allow the *Computer* the ability to move around the *Memory* stack on command and handle instructions in a non-sequential manner. This gives the *Computer* the arbitrary ability it needs to incorporate and handle unique functions and loops that may be designed by a user. In simpler terms, this means that it gives the *Computer* the means to move around the stack and repeatedly jump forwards or backwards to necessary instructions as may be needed by a user's uniquely designed functions.

Through all of this, we have designed the beginning foundation for a functional computer that could be added to and potentially scaled to handle a wide variety functionality.

Testing

- Brief tests were designed and conducted with each step of the continuous development and integration of the classes that made up the architecture of the project. The *bit*, *longword*, *RippleAdder*, *Multipler*, *ALU*, *Memory*, and *Computer* classes all contain respective test files that briefly went over their functionalities to ensure the proper workings.
- The comments in the test files describe the processes being done to test the components.

- The tests were usually done with random-generated values, so that the tests could be repeatedly ran with new values to ensure consistent precision.
- Each test file also includes the tests of the previous class files. The final *Computer* test file, *cpu_test3* includes the tests of all previous class tests.