Homework #2: Some Assembly Required

Due Sunday, February 21st, at 11:59 p.m.

In this homework, you will write an interpreter for a small assembly language. When you are done, your solution will be able to parse and execute programs written in the assembly language, modeling the behavior of the programs as if they were executing on a simple computer processor.

Your solution to this homework should maximize code reuse and be extensible so that new assembly language instructions and/or a different computer processor model could be used in the future. To achieve these design goals, you must appropriately define and implement Java interfaces and/or abstract classes. Additionally, you must write unit tests to check the correctness of your solution. Overall, your goals for this homework are to:

- Design software for extensibility and code reuse.
- Understand and apply the concepts of polymorphism, information hiding, and writing contracts, including an appropriate use of Java interfaces and/or abstract classes.
- Design and implement software based on your interpretation of informal specifications.
- Write unit tests and automate builds and tests using JUnit, Gradle and Travis-CI.
- Write tests based on a functional specification, with good testing practices and style.
- Understand the benefits and limitations of code coverage metrics and interpret the results of coverage metrics.

A simple computer and its assembly language

This section, along with Appendix A, describes a simple computer architecture and the minimum features your solution must support.

A computer architecture typically consists of (at least) some way to store in-processor data (called *registers*), off-processor data storage (*memory*) referenced by numeric address, a set of instructions that manipulate the values stored in the registers and memory, and a set of conventions for how the instruction set is used.

For this assignment, you must model a stack-based computer architecture whose processor has at least eight registers: one *program counter* (called `PC`), a *stack pointer* (called `SP`), and six general-purpose data registers (called `R0`, `R1`, `R2`, `R3`, `R4`, and `R5`). The program counter stores the address of the currently-executing instruction; it is incremented after executing each instruction, except for instructions that specifically set the program counter to a different value. Each data register stores a 32-bit signed int (identical to Java's int type) that can be modified or moved to or from memory. Each data memory location stores a signed 32-bit int; in our architecture, a program's instructions are stored in different memory than a program's data. The stack pointer, instruction set, and related conventions provide architecture-level support to treat part of memory as a stack, simplifying in-memory storage of local data and the implementation of subroutines; see Appendix A for details.

When a program starts, the program counter is initially set to 0 (the first instruction of the program), and the data registers and data memory are also initially 0. The stack pointer is initially set to the memory size minus 1, the last valid memory address.

Appendix A describes the minimum assembly language that your solution must support. To help understand the instruction set, we recommend that you see the example programs in your repository at `src/main/resources`.

Implementing your solution

Create a class representing the processor described above. The processor should support registers and memory, and should be able to execute the instructions described in Appendix A. We are intentionally vague about some details of the processor specification and machine architecture. You may make reasonable choices for details that we have not specified.

Write a program that parses a file containing an assembly language program and then uses the processor class to execute the program. Appendix A also describes some minimal requirements for your parser.

Testing your implementation

You must test your solution using JUnit tests. Notably, your testing should not be limited to the inputs we provide, and you are expected to thoroughly test your solution. A good test suite for this problem consists of both end-to-end tests (such as checking your interpreter's behavior on whole programs) and, more importantly, a thorough suite of true unit tests, with (usually) many independent tests per operation.

Although your goal is not to achieve high test coverage, a good test suite will likely achieve nearly 100% line coverage, excluding the test code itself.

Design discussion

In `rationale.md` or `rationale.pdf` in your homework directory, describe and critique the extensibility of your solution. Specifically describe what a programmer must do (what files and classes must be modified, what classes must be created, etc.) to add a new instruction to the assembly language.

Evaluation

Overall this homework is worth 100 points. We will grade your work approximately as follows:

- Correctly applying the concepts of polymorphism and information hiding: 20 points
- Java best practices and compatibility with our informal specification: 40 points
- Unit testing, including compliance with best practices: 30 points
- Documentation, style, and development practices: 10 points

Hints

- Use the `Scanner` class to read our sample input files, and a `BufferedReader` atop `System.in` to read from the console.
- Testing should be a first-class activity in this homework assignment, not an afterthought. You should start writing tests as you complete your solution; do not delay writing unit tests until after your implementation is complete.
- Adding a new instruction to the assembly language should require modifying only one part of your solution. Think carefully about how you can use interfaces to eliminate dependencies within your program; e.g., if your parser depends on just an instruction interface and not on each individual instruction, you probably don't need to modify the parser if you add a new instruction.
- We recommend using one or more Java enums to represent operations in the assembly language. See Effective Java Item 34 and Item 42 for details; understanding the `Operator` example in those items is especially helpful for understanding one extensible way to design your solution.
- We will assess your line coverage with the Jacoco reports that can be generated with `gradle jacocoTestReport`. The reports can be found in `build/reports/coverage`. You might want to use the IDE integration of coverage while you write your tests.

Appendix A: The required assembly language

This appendix describes an assembly language that is designed to be easy to parse and simulate on the described machine. You may assume that all tokens (instruction names, register names, and constant values) are separated by white space. If a token begins with a ";" character, the rest of the line should be treated as a comment and skipped. See `src/main/resources` for example programs.

Your solution must support at least the following instructions:

Data movement instructions. The following instructions move data from one location to another:

- `MOV` *dstReg srcReg*: "Move." Copies the value of register *srcReg* to register *dstReg.*
- `LOAD` *dstReg srcAddrReg*: Copies a value from memory to register *dstReg.* Register *srcAddrReg* contains the memory address of the value to be copied.
- `LOADI` *dstReg val*: "Load immediate." Sets the register *dstReg* value to *val.*
- `STORE` *dstAddrReg srcReg*: Copies the current value from register *srcReg* to the memory address currently stored in register *dstAddrReg.*

Arithmetic instructions. The following instructions perform some arithmetic operation:

- `ADD` *sumReg valReg*: Adds the value stored in register *valReg* to whatever value is currently stored in register *sumReg*, replacing whatever was previously stored in *sumReg*, i.e., *sumReg* += *valReg.*
- `ADDI` *sumReg val*: "Add immediate." Adds the constant *val* to whatever value is currently stored in register *sumReg*, replacing whatever was previously stored in *sumReg.*
- `SUB` *differenceReg valReg*: Subtracts the value in register *valReg* from whatever value is currently stored in register *differenceReg*, i.e., *differenceReg* -= *valReg.*
- `SUBI` *differenceReg val*: "Subtract immediate." Subtracts the constant *val* from whatever value is currently stored in register *differenceReg.*
- `MUL` *productReg valReg*: Multiples whatever value is currently stored in register *productReg* by the value stored in register *valReg.*
- `MULI` *productReg val*: "Multiply immediate." Multiples whatever value is currently stored in register *productReg* by the constant value *val.*

- `DIV` *quotientReg valReg*: Divides whatever value is currently stored in register *quotientReg* by the value stored in register *valReg*, storing the result (the quotient) in *quotientReg*.
- `DIVI` *quotientReg val*: "Divide immediate." Divides whatever value is currently stored in register *quotientReg* by the constant value *val*, storing the result (the quotient) in *quotientReg*.
- `MOD` *remainderReg valReg*: Divides whatever value is currently stored in register *remainderReg* by the value stored in register *valReg*, storing the remainder in *remainderReg*.
- `MODI` *remainderReg val*: "Mod immediate." Divides whatever value is currently stored in register *remainderReg* by the constant value *val*, storing the remainder in *remainderReg*.

I/O instructions. These instructions read (or write) a character from (or to) the console:

- `READ` *reg*: Reads a single character of input from the console and places the character into register *reg*. (This instruction may block until the user types enter.)
- `WRITE` *reg*: Writes a single character to the console from the low-order 16 bits of register *reg*.

Control-flow instructions. The following operations change (or conditionally change) the next instruction to execute, (possibly) resetting the program counter to a specific value rather than just advancing it to the next instruction:

- `JUMP` *addr*: Sets the program counter to the value *addr*.
- `JEQ` *reg addr*: "Jump if equal to zero." If the current value in register *reg* is zero, sets the program counter to *addr*.
- `JNE` *reg addr*: "Jump if not equal to zero." If the current value in register *reg* is nonzero, sets the program counter to *addr*.
- `JGT` *reg addr*: "Jump if greater than zero." If the current value in register *reg* is positive, sets the program counter to *addr*.
- `HALT` Stops executing the program.

Stack instructions. These instructions allow the programmer to treat a portion of the machine's data memory as a stack, simplifying the implementation of subroutines. The stack is implemented with a *stack pointer* stored in register `SP`. This register is initialized to point to the highest address in memory (the memory size minus one). When a value is pushed onto the stack, it is stored in the memory location addressed by the stack pointer, which is then decremented. When a value is popped from the stack, the stack pointer is incremented and the value stored at the memory location in the stack pointer is then loaded into a register. Note that the stack grows "downwards" in memory (from the highest address toward zero).

- `PUSH` *reg*: Pushes a register's value onto the stack. In other words, stores the value from register *reg* into memory at the address currently stored in register `SP` (the stack pointer), and decrements the stack pointer.
- `POP` *reg*: Pops a value from the stack into register *reg*. In other words, increments the stack pointer, and loads the value from the memory address at the (newly-incremented) stack pointer into register *reg*.
- `CALL` *addr*: Calls a subroutine at the given address. In other words, pushes the return address (the current value of the program counter plus one) onto the stack, and then sets the program counter to *addr* rather than just advancing it to the next instruction.
- `RET`: "Return." Returns from the current subroutine. In other words, pops a value off the stack into the program counter, rather than just advancing it to the next instruction.

In addition to being accessed and modified by the instructions above, the stack pointer may be accessed as if it were a general purpose data register. It is common to compute memory addresses relative to the stack pointer in order to access data that was recently pushed onto the stack (such as subroutine arguments).

When writing assembly programs, we recommend that you abide by the following calling convention: The first argument to a subroutine (if it has one) should be placed in register `R5`. Any additional arguments should be pushed onto the stack (before executing the `CALL` instruction). The called subroutine is responsible for saving and restoring any registers it overwrites, with the exception of R5. However, subroutines may choose not to restore certain registers, so long as their documentation makes this clear.

Any program data ("the heap") should be stored at the low end of memory (starting at address zero), so as not to interfere with the stack (which grows from the highest memory address downward).