ICSI404 – Assignment 8 – an assembler

This assignment builds on the previous assignment.

In the last assignment, testing was done by creating strings of bits (0101010 style) and pre-populating memory. This is a very tedious way to write programs. Nearly as soon as we had computers, their designers realized that programming using bits was too error prone and difficult. Kathleen Booth, in 1947, invented the first assembler.

An assembler is a program that takes text as input and outputs bits that correspond to that text. As you can imagine, writing

move R1 -1

is much easier than writing 0001 0001 1111 1111

To create an assembler, we will create a new class, Assembler. It will have one public method: public static string[] assemble (string[]). We will pass into that method a series of strings, like “move R1 -1” and it will return the bit patterns that are represented by that command. A simple approach is to use the split method on string to break an input line up into tokens (remember this from ICSI311?).

A more complete, robust implementation would be to build a lexical analyzer. This would loop over each character. When a space is found, it would decide whether this is a number, a register (R1, for example) or a keyword (like “move”).

You could use a case statement to decide what to do based on the first token (the assembly command). Note that there will be lots of cases of repeated functionality – you must create helper methods. One example of this is converting a register name (like R1) to a bit pattern (0001). While we are working on low level code, we should not forgo good code quality.

Like lexical analysis, there is a more complete, robust solution available – a recursive descent parser. These were covered in ICSI311, but Wikipedia has some good explanation: https://en.wikipedia.org/wiki/Recursive_descent_parser

You must create a new test class (assembler_test) that sends assembly language into the assembler and confirms that the bit patterns created are correct.

You must submit buildable .java files for credit.

Rubric	Poor	OK	Good	Great
Comments (SO2, PI2)	None/Excessive (0)	"What" not "Why", few (5)	Some "what" comments or missing some (7)	Anything not obvious has reasoning (10)
Variable/Function naming (SO2, PI3)	Single letters everywhere (0)	Lots of abbreviations (5)	Full words most of the time (8)	Full words, descriptive (10)
Unit Tests (SO 2, PI4)	No unit tests are supplied or they none pass. (0)	Few unit tests are included and/or few pass. (7)	All instructions covered by unit tests and all pass. (13)	Multiple unit tests per instruction and all pass. (20)
Assembler (SO 1&2, PI 1) Software meets the specification)	The software doesn't meet the specification (doesn't compile, doesn't assemble a significant number of instructions). (0)	The software meets some of the specification (e.g. only some instructions are properly assembled). (7)	The software meets all of the specification (typical instructions are correctly assembled). (13)	The software handles significant corner cases (bounds cases are correctly assembled) and gives errors when the input is out of bounds. (20)
Lexing: (SO1, PI2) Software uses lexical analysis to separate incoming text	software uses position or other mechanism to split text (0)	software uses ad hoc mechanism for splitting strings (10)	software uses language appropriate mechanisms to split text (20)	software uses a dedicated lexical analyzer to separate incoming text (20) + BONUS 10 points toward course
Parsing: (SO1, PI3) Software parses incoming text	Software doesn't parse; makes overly broad assumptions about format (0)	Software uses custom code for every instruction (10)	Software uses minimal code for each instruction format with good sharing (20)	Software uses a parser (recursive descent or generated) (20) + BONUS 10 points toward course

BONUS POINTS – these 10/20 points are added to the sum of your assignments before dividing to get an average.