Computer Architecture Laboratory
Assignment 2

Write an assembler for the *ToyRISC* ISA.

Input to the Program

1. full path to the assembly program.
2. full path to the object file to be created.

Output of the Program

- the program must create the object file at the specified location.

Broad Outline of the Steps

The mechanism of assembly has already been discussed in class. As an example, consider the assembly of this instruction from assignment 1's statement: `load %x0, $a, %x4` in Table 1.

Table 1: Assembly of `load %x0, $a, %x4`

Field	Value	Binary Representation
Operation	load	10110
`rs1`	`x0`	00000
`rd`	`x4`	00100
`imm`	0	00000000000000000 (recall that 'a' refers to address 0)
Full Instruction		**10110000000010000000000000000000** = **-1341652992** (signed representation)

The task of assembly has been further simplified. We have done the parsing of the assembly program for you (`supporting_files/src/`).

- For all labels, both in data and in code, we have set up the symbol table that maps label to address. To get the address corresponding to a label `str`, simply call `ParsedProgram.symtab.get(str)`.
- Similarly, instructions are also gathered in a simple array. To access the instruction at a given address `addr`, simply call `ParsedProgram.getInstructionAt(addr)`.
- Each instruction is represented by an object of the `Instruction` class (see `generic/Instruction`). You can get the operation type and the operands by calling the appropriate functions mentioned in this class.

- Each operand is represented by an object of the `Operand` class (see `generic/Operand`). You can get (i) the operand type – that is, is it a register operand, an immediate, or a label, and (ii) the value – that is, the register number, the immediate value, or the label string.

You only need to complete the function `src.generic.Simulator.simulate()`.

The expected format of the object file is as follows:

- Header: Here, you simply write the address of the first instruction. This is an integer, and therefore 4 bytes long.
- Data: Here, you write all the static data one after another. Each datum is 4 bytes long.
- Text: Here, you write the encoded instructions one after another. Each instruction is 4 bytes long.

Please see the following example:

```
        .data
a
        10
b
        20
        .text
        load %x0, $a, %x4
        end
```

The given assembly file will contain the integers (and not strings!) 2, 10, 20, −1341652992, −402653184. The header consists of the instruction of the first instruction, in this case 2. The two data follow. The binary equivalents of the `load` and `end` instructions are then placed.

You may read the file that you create using the `xxd` command.

Running, Submitting and Testing

- To run your program, the arguments are `<path-to-assembly-program> <path-to-object-file>`. There are two way of running: (i) through Eclipse, (ii) by exporting a jar file. Run the command `ant` in the folder which contains `src` and `build.xml`. It will compile your code. If there are no errors, run the command `ant make-jar`. A jar file is created at the location `jars/assembler.jar`. Now to run your program, run this command on the terminal: `java -Xmx1g -jar <path-to-jar-file> <path-to-assembly-program> <path-to-object-file>`.
- You may test against the programs you submitted in your previous assignment.
- You are expected to submit a zip archive of the `src` folder. The name of your zip archive should be "<entry-number-1>_<entry-number-2>_assignment2.zip".

- Test if your submission is in the correct format. Run the script `python test_zip.py <path-to-zip-file>` in the same folder that contains `build.xml`, `test_cases`, and `test_zip.py`. As in the previous assignment, the script will test against some test cases.