**Project part 0: Write an assembler that converts each instruction into its 16 bit representation.**

**Design notes**

**Opcode to Binary Map**: We make hash table for opcodes of instructions to binary code. This good for quick change during make assembly.

**Reading Instruction**: We read instructions from file. We think to read from right to left. We look for space in line to know where instruction and operands split.

**Binary to Octal Method**: We have method BinaryToOctal to change binary to octal from smallest digit. But we see in code, maybe not perfect for all situation.

**Format and Convert**: We have function formatAndConvertInstruction. It take opcode and other parts, make them into binary string with padding zeros if need, then change to octal.

**Handling Different Instructions**: We put instructions into categories. Each category like LoadStore, Transfer, they have own way to handle. We write different methods for each one.

**Address in Parsing**: When we start parse, we check if first instruction is "LOC". If not, we set address to 6. We reserve first five addresses.

**Writing to Files**: We have two methods for write results to files. One is writeListingFile for list of instructions with original lines. Other is writeLoadFile for load file.

Following are the elaborated description of every function used in the Assembler project.

1. readFile method used to take a text file as input and read each line and add a line as an instruction to an ArrayList named **Instructions** until a null line is found. If any error occurred during this process, programs throws an IOexception error.
   1. Why arraylist? Easy for manipulation - add, get, remove. Dynamic sizing, Indexed Access
2. The function loadDict() is used to load a HashMap named Ins2Bi which stores all the instructions and their respective opcodes in a key value pair. Both key and value are strings. The key is the instruction and value is its opcode.

Eg: Ins2Bi.put("STR","000010"); → STR is key whose value is 000010 in the HashMap

1. Then we have a method named parse() which parses instructions and stores it into addresses. We have reserved the first 5 addresses for special cases as described in the Project Description. These addresses are:

**Reserved Locations**:

Memory Address                Usage

0                              Reserved for the Trap instruction for Part III.

1                              Reserved for a machine fault (see below).

2                              Store PC for Trap

3                              Not Used

4                              Store PC for Machine Fault

5                              Not Used

1. The first thing we do in the parse() method is to check if the first instruction in the input file is “LOC”. If yes, we initialize the address = 6 (as, first 5 address are reserved Locations)

1. Then we iterate through every instruction in the ArrayList **instructions**.

For each instruction, we split the instruction whenever a whitespace is found in the instruction using the split function in JAVA to convert the instruction which is a string type to an array. For example, if instruction is "ADD R1, R2", calling split("\\s+") on this string will result in the array ["ADD", "R1,", "R2"], because the whitespace character between words "ADD", "R1,", and "R2" is used as the delimiter.

1. For each instruction, we check if the instruction is a LOC. We do this by checking the first array element we created using split. I.e. tempArray[0]. If yes, we set the address as given by the instruction.We get the array using the second element in the tempArray i.e tempArray[1]. For example, “LOC 9” instruction sets the address to 9
2. Else, If the instruction is “Data”,
   1. We check if the second element i.e. tempArray[1] in the instruction is END. If yes, we set the second element of the array to “01024” instead of END.(END has value “01024”)
   2. For all the instructions, we add them  to an ArrayList named **parsedInstru** with type String**.** As the instruction is Data, it will have two parts, the first will be the address in octal, the second will be the value to be stored in that address.
   3. We first format the address as: **String.format("%06o", address)**. This formats the address variable as a string with a width of at least 6 characters, padded with leading zeros (%06), and represented in octal format (%o). So, it ensures that the address is represented as a string of 6 characters in octal format. This is the first part of the instruction.
   4. Secondly, we we format the tempArray[1] which gives the value to be stored in the address in the same way. String.format("%06o", Integer.parseInt(tmpArr[1])): This part parses the second token of the instruction (tmpArr[1]) as an integer using Integer.parseInt(). Then, it formats this integer as a string with a width of at least 6 characters, padded with leading zeros (%06), and represented in octal format (%o). This ensures that the data value is represented as a string of 6 characters in octal format.
3. Else, if the instruction is not data or loc, we determine which category is the instruction. The categories are:

* Miscellaneous instructions
* Load and Store instructions
* Transfer instructions
* Arithmetic and Logical instructions
* Register Operation instructions
* Shift and Rotate instructions
* I/O instructions
* Floating Point and Vector instructions

Each category has its own logic to handle instructions according to the      project description.

1. Then have a method writeListingFile that is used for outputting the ListingFile as a text file.

1. Finally, we have a method writeLoadFile used for outputting the LoadFile as a text file.