CODE DESCRIPTION: GROUP 6

<u>Introduction: prominent data structures used in the assembler:</u>
//copied from variable declaration in Sicmasm.java

static String mnemonic; // mnemonic string

static int op; // variable to store op code
static int first2; //Integer to store first two digits of Object Code

private static File main,obj,lst; //files to read in and out too

static int address=0000;static int nxtloc=0000; //starting address and PC address

hash table for storing label and corresponding address
static Pass1Table symTab=new Pass1Table();
static Pass1Table addressTable=new Pass1Table(); //

static OpTable ObjectCodeTable=new OpTable();

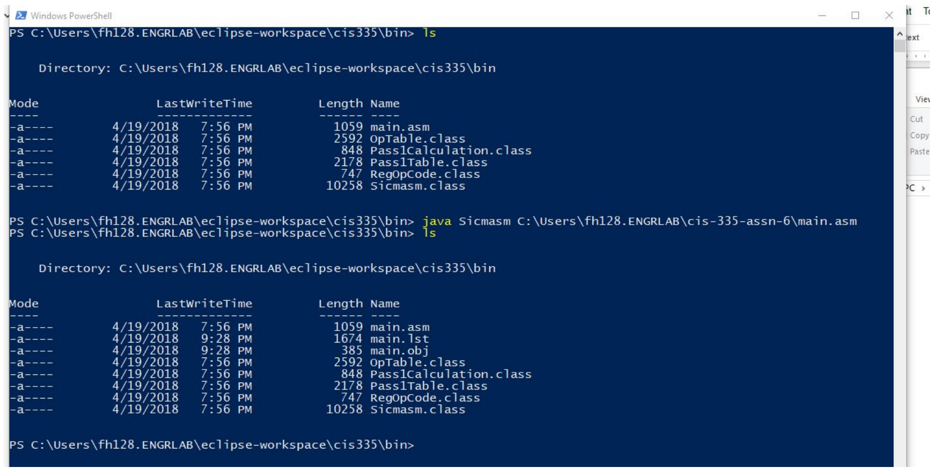
static ArrayList<String[]> mainAsString = new ArrayList<String[]>(); //array list for storing each string in the main.asm in a two D Array list

static OpTable OpTab=new OpTable(); //Optable created to get the op code for each instruction

static StringBuilder OBJcode = new StringBuilder(); //String builder declared to easily piece together each lines objectcode

static RegOpCode register=new RegOpCode(); //used to get the opcode for the special cases where the program uses a register

<u>Output</u>



```
PS C:\Users\fh128.ENGRLAB\eclipse-workspace\cis335\bin> ls


    Directory: C:\Users\fh128.ENGRLAB\eclipse-workspace\cis335\bin


Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        4/19/2018   7:56 PM           1059 main.asm
-a----        4/19/2018   7:56 PM           2592 OpTable.class
-a----        4/19/2018   7:56 PM            848 Pass1Calculation.class
-a----        4/19/2018   7:56 PM           2178 Pass1Table.class
-a----        4/19/2018   7:56 PM            747 RegOpCode.class
-a----        4/19/2018   7:56 PM          10258 Sicmasm.class


PS C:\Users\fh128.ENGRLAB\eclipse-workspace\cis335\bin> java Sicmasm C:\Users\fh128.ENGRLAB\cis-335-assn-6\main.asm
PS C:\Users\fh128.ENGRLAB\eclipse-workspace\cis335\bin> ls


    Directory: C:\Users\fh128.ENGRLAB\eclipse-workspace\cis335\bin


Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        4/19/2018   7:56 PM           1059 main.asm
-a----        4/19/2018   9:28 PM           1674 main.lst
-a----        4/19/2018   9:28 PM            385 main.obj
-a----        4/19/2018   7:56 PM           2592 OpTable.class
-a----        4/19/2018   7:56 PM            848 Pass1Calculation.class
-a----        4/19/2018   7:56 PM           2178 Pass1Table.class
-a----        4/19/2018   7:56 PM            747 RegOpCode.class
-a----        4/19/2018   7:56 PM          10258 Sicmasm.class


PS C:\Users\fh128.ENGRLAB\eclipse-workspace\cis335\bin>
```

Overview

Lines 34-54: Beginning of main method/file generation/array building

The assembler's main method begins by first checking the input arguments for a user-entered path to the assembly file. If two arguments are detected, the path entered generates a file called main by the implementation of File(). The files for the object file and the listing file are then stored in variables called obj and lst. To get the names for these new files, a substring is taken from the original assembly file, going from the beginning of the file's name up to the last index of '.'; .obj and .lst are then added onto the file names. Now, the original main.asm file is ready to be read and copied. It is read line by line with a BufferedReader in a while loop which terminates once the end of file is reached (when the current line is equal to null.) Each line is stored in a string variable called mainInLine and then added to an Array List called list, while a copy of each line split by a space is added to a string variable called str1, which is also added to an Array List of string arrays called mainAsString.

Lines 55-120: Pass 1

Pass 1 in this code is executed primarily by translating commands from ASM to SIC using if statements, isolating the mnemonics, calculating the addresses and then storing the addresses and symbols in hash maps. It begins with a for loop within the while loop which reads the current split line stored in str1 substring by substring. A series of if statements within the for loop is used to translate certain mnemonics and characters into the proper SIC format. First, an if statement tests for comments by checking for semi-colons, and then eliminating the semi-colons and everything after. The next if statement checks to see if the current substring is "MOV" or "+MOV". If true, more if statements within the if statement test to see what the next substring is to determine what the operand is. If the first character of the next substring is %, another set of if statements check the third character for the letters 'L', 'A' and 'X'. In each case, the current substring is switched to STL, STA or STX respectively, and the next substring (containing the operand) is erased. Similarly, a separate if statement checks for instances of +MOV to be translated into +STT. Finally, if none of these conditions are met, there is a series of if statements which checks for instances of LDA, LDX, LDT and LDB by checking the next substring for the characters 'B', 'A', etc.

Lines 121-160: Pass 1 cont.

Once the instructions are translated accordingly in each line, the mnemonic is stored in a variable called mnemonic. If the length of the current line in str1 is three substrings, an if statement determines the location of the mnemonic to be at str1[1]. Otherwise, mnemonic is assigned str[0]. One the mnemonic is isolated, an object of type Pass1Calculation is created from the Pass1Calculation class with the mnemonic as its data field. The following lines of code then use the int variables address and nxtloc, which are intialized as 0 at the beginning of the program. In each line, address is assigned to nextloc, and nextloc is incremented by the value returned by the Pass1Calculation method getInstructionSize, which returns what the location counter should be incremented by with each mnemonic (this is determined by a set of if statements within the method which analyze the mnemonic data field.) Once the address of the current line is determined and nxtloc is set for the next iteration, the address is converted to hexadecimal and stored in a hash map with the line number (stored in count) called addressTable. This hash map is located in the class Pass1table, which essentially holds all of the addresses for each line number to be used for later. Finally, the while loop from the beginning concludes each iteration (each line) by detecting if there is symbol (str1.length == 3), and then passing the symbol and its hex address to symTab, which is also a hash map defined in the Pass1table class. Once the end of the file is reached, the while loop is terminated, and Pass 1 is complete.

Lines 162-262: Pass 2

Pass 2 begins by using the array list of strings mainAsString to once again isolate the mnemonic line by line as a string Mnemonic, as well as the destination as a string using get functions. Once these two strings are obtained, the object code starts to formulate. First, the two leftmost digits are obtained from the hash map OpTab and stored in an integer called first2. OpTab is a hash map stored in the OpTable class which contains the op codes for all of the mnemonics to be found in a given program. Using if statements as well as OpTable's OPcode(Mnemonic) function, first2 is assigned a value based on the value returned by the hash map, with the addition of the ni flags if neccessary. Next, the flags are set for each case by assigning a value to int flag using if statements again. The if statements detect if extended mode is being used (charAt(0) == '+'), as well as "LENGTH", "BUFFER" and "RSUB." This establishes the third digit of the object code. Finally, the remainder of the object code is calculated and saved into displacement using the destination variable as a key for the symTab hash map to return a value based on the addresses of the symbols. Once the opcode, the flags and the displacement are all found in the given line, they are combined using a string builder called OBJcode and then converted to uppercase and saved in a string variable called objcode. OBJcode is then cleared for the next iteration, and objcode is added to an OpTable() constructor called ObjectCodeTable. As the loop goes on, each line's object code will be added to ObjectCodeTable, which is used in generating the object code and listing files later.

Lines 264-279: Listing File

The listing file is created using the getAddress method of addressTable, the getObjectCode method of ObjectCodeTable and the contents of list, which was the original array created holding each line of the source code. The values obtained from these data structures together form all of the contents of the listing file. In order to write onto the listing file, a PrintWriter named lstout is implemented and directed at lst, the blank listing file generated at the beginning of the main method. The PrintWriter goes line by line in the listing file using a for loop, first printing the address found in addressTable, then a tab, then the source code found in list and finally the object code found in ObjectCodeTable. At the end of the for loop, the listing file is fully written, and the PrintWriter is closed.

Lines 281-end: Object File

For the object code file, three string arrays are created. For one of them, it is a 2-d array. This contains all text records. First the header record is created. This gets the name, starting address, and length of the program. Text records are subsequently added thereafter. All text records are added in the same 2-D array named text. This part of the program relies heavily on the hashMap *addressTable.* It retrieves the address for every iteration within the for loop, where i is incremented until i = list.size() . This refers to the arraylist declared at the beginning. It first checks to see if this equals END, then writes the END record accordingly. Afterwards, a variable called first is assigned a parsed integer from *addressTable* at *i* with a radix of 16 (this is the base of the integer). So this address is stored as a hex number, then parsed to an int. Last is assined a value from *addressTable* bsaed on index i. Last is then assigned the difference of first and the previous value of last. This writes the first and last lines of the object code. Every subsequent line is retrieved from the text 2d array. There is a check boolean that checks to see if text is equal to null at one of the lines. If this is true, i is decremented. This indicates that there is an extra line. After text 2d array is populated by these loops, it is formatted. The formation starts with the head array. We need to format this to a minimum of 6 bits. Next, the text format is created in similar fashion, as well as the end format. A printwriter s used with passed arguments of the *obj* file created at the beginning of the program. This part of the program mainly uses the *printwriter write()* method.

Credits:

Ryan: Pass 1, Pass 2, Listing file, Subclasses
Peter P: Object code file, debugging, command line function, report
Peter Q: Pass 2, debugging, report
William: Pass 1, Subclasses, Github
Isaac: Subclasses, Object code file, debugging