

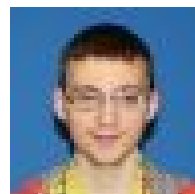
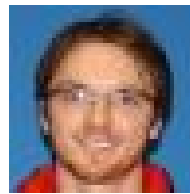
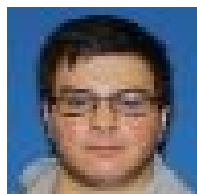
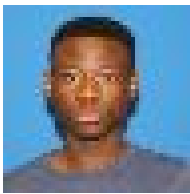
CIS 335 Assignment 6, Group 5

Cover Page

Team Member Names

Isaac Baah, Peter Quigley, Peter Parianos, Ryan Reynolds, and William Yznaga

Photo IDs



Code and Functionality Report

Introduction

Prominent data structures used in the assembler

The data structures most often and most importantly used in the assembler were auxiliary String and int variables, HashTables, ArrayLists generally of instantiated type String, and auxiliary objects both from the Java Standard API and from custom auxiliary classes for this project (e.g. Pass1Table, OpTable, and more).

The following selections of code from Sicasm.java depict *in situ* (actual) usage of these and more data structures, with especially important data structures' comments emphasized **in bold**, and assorted necessary clarification comments [in brackets, like so] (which clarifications appear only in this report, and not in the code's comments).

```
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[s] 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

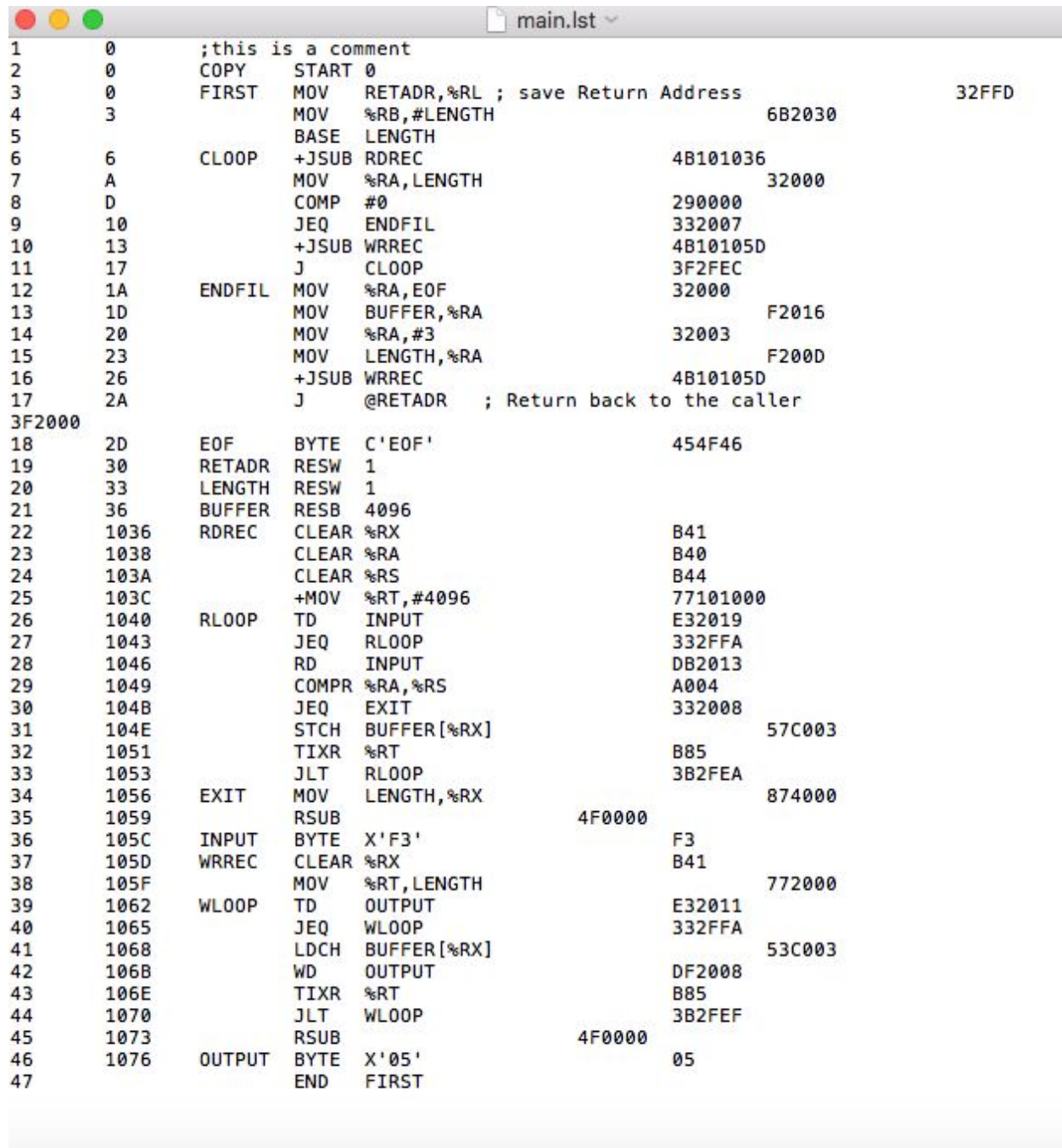
static OpTable OpTab=new OpTable(); //OpTable [object] created to get the op
code for each instruction
```

```
static StringBuilder OBJcode = new StringBuilder(); //String builder [object]  
declared to easily piece together each lines [of] objectcode
```

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

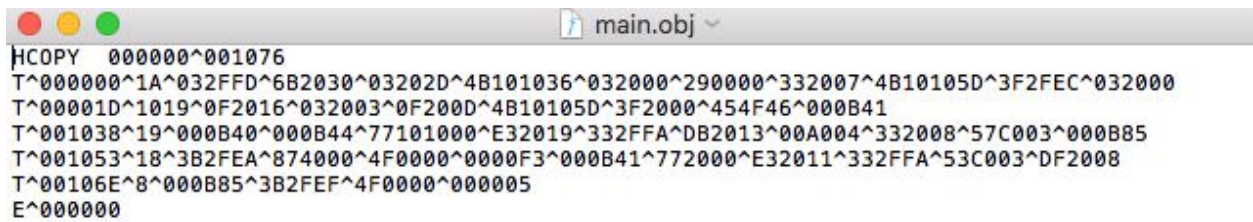
Output

The first and following screenshot depicts the outputted listing file from invocation of the main-containing Java class “main.asm” as input:



```
1      0      ;this is a comment
2      0      COPY      START 0
3      0      FIRST     MOV    RETADR,%RL ; save Return Address          32FFD
4      3      MOV    %RB,#LENGTH          6B2030
5      BASE    LENGTH
6      6      CLOOP     +JSUB  RDREC          4B101036
7      A      MOV    %RA,LENGTH          32000
8      D      COMP    #0          290000
9      10     JEQ    ENDFIL          332007
10     13     +JSUB  WRREC          4B10105D
11     17     J      CLOOP          3F2FEC
12     1A     ENDFIL    MOV    %RA,EOF          32000
13     1D     MOV    BUFFER,%RA          F2016
14     20     MOV    %RA,#3          32003
15     23     MOV    LENGTH,%RA          F200D
16     26     +JSUB  WRREC          4B10105D
17     2A     J      @RETADR ; Return back to the caller
3F2000
18     2D     EOF      BYTE  C'EOF'          454F46
19     30     RETADR   RESW  1
20     33     LENGTH   RESW  1
21     36     BUFFER   RESB  4096
22     1036    RDREC   CLEAR %RX          B41
23     1038    CLEAR %RA          B40
24     103A    CLEAR %RS          B44
25     103C    +MOV %RT,#4096          77101000
26     1040    RL00P   TD      INPUT          E32019
27     1043    JEQ    RL00P          332FFA
28     1046    RD      INPUT          DB2013
29     1049    COMPR %RA,%RS          A004
30     104B    JEQ    EXIT          332008
31     104E    STCH   BUFFER[%RX]          57C003
32     1051    TIXR %RT          B85
33     1053    JLT   RL00P          3B2FEA
34     1056    EXIT    MOV    LENGTH,%RX          874000
35     1059    RSUB          4F0000
36     105C    INPUT   BYTE  X'F3'          F3
37     105D    WREC    CLEAR %RX          B41
38     105F    MOV    %RT,LENGTH          772000
39     1062    WLOOP   TD      OUTPUT          E32011
40     1065    JEQ    WLOOP          332FFA
41     1068    LDCH   BUFFER[%RX]          53C003
42     106B    WD      OUTPUT          DF2008
43     106E    TIXR %RT          B85
44     1070    JLT   WLOOP          3B2FEF
45     1073    RSUB          4F0000
46     1076    OUTPUT  BYTE  X'05'          05
47     END      FIRST
```

The next screenshot depicts the resultant object code file as generated by our code:

A screenshot of a terminal window with a title bar containing three colored circles (red, yellow, green) and a file icon labeled 'main.obj'. The terminal displays several lines of assembly code in uppercase letters and hexadecimal values, separated by spaces and carets (^).

```
HCOPY 000000^001076
T^000000^1A^032FFD^6B2030^03202D^4B101036^032000^290000^332007^4B10105D^3F2FEC^032000
T^00001D^1019^0F2016^032003^0F200D^4B10105D^3F2000^454F46^000B41
T^001038^19^000B40^000B44^77101000^E32019^332FFA^DB2013^00A004^332008^57C003^000B85
T^001053^18^3B2FEA^874000^4F0000^0000F3^000B41^772000^E32011^332FFA^53C003^DF2008
T^00106E^8^000B85^3B2FEF^4F0000^000005
E^000000
```

The last screenshot depicts directory file-listings before and after invoking the main-method-containing Java class on the “main.asm” test file.

Before invoking, the directory listing shows Java class files and main.asm, whereas after invoking, another directory listing shows the presence of the Listing file (“main.lst”) and the Object Code file (“main.obj”).

```
Windows PowerShell
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-assign-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

Beginning Operations

Beginning of main method, leading to file generation and array building

Lines 34 through 54

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, respectively.

To get the eventual filenames 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. In this way, the listing file and the object file both retain the same pre-extension filename as the passed-in assembly file.

Those tasks finished, 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 (that is, when the current line is equal to null). Each line is stored in a string variable called mainInLine and then added to an ArrayList called list, while a copy of the literal string data of each line is added to a string variable called str1; between appending of lines, a separator of space (" ", sans quotes) is also appended to str1. Finally, the current resulting str1 is in turn added to an ArrayList of String arrays, the top-level ArrayList itself being called mainAsString.

Pass 1

Part 1

Lines 55 through 120

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 semicolons, 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.

Part 2

Lines 121 through 160

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]`. Once 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 `nextloc`, which are initialized 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 `nextloc` 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.

Pass 2

Lines 162 through 262

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 necessary. 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.

Listing file generation

Lines 264 through 279

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.

Object file generation

Lines 281 through end of code

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 assigned a value from *addressTable* based on index *i*. *Last* is then assigned the difference of *first* and the previous value of *last*. This writes the first and final lines of the object code. Every subsequent line is retrieved from the textual 2-dimensional array.

To finish the object code generation, 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 the 2-D array is populated by these loops, it is formatted. The formatting 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* is 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

These denote which areas team members put majority effort toward:

Not exhaustive of all areas to which members contributed

Ryan Reynolds

Initial coding for Pass 1, Pass 2; continued coding for Listing file and auxiliary subclasses

Peter Parianos

Coding for Object Code file generation; also debugging, initial command line functioning and testing of, and initial report writing

Peter Quigley

Continued coding for Pass 2; also debugging, and initial report writing

William Yznaga

Some coding for Pass 1 and subclasses; also all setup of initial Github and all initial teaching of usage of git and GitHub; extensive report proofreading

Isaac Baah

Some coding for subclasses, continued coding for Object Code file generation; debugging