COMPUTER ORGANIZATION PROJECT 1-DOCUMENTATION

PART I : EXPLANAITION

INTRODUCTION

An assembler is a program that translates assembly language into machine code. The assembler takes a file containing the assembly language code as input and converts it into an object code that contains is understood by the computer.

An assembler can be of two types: One Pass Assembler or Two Pass Assembler. In our project we have written a program for a 12-bit Two Pass Assembler. In this two-pass mechanism the assembler goes through the assembly code and lists out errors (if any) and generates tables. The assembler doesn't perform the second pass if there are any errors in the first pass. However, if there are no errors in the program, it generates the corresponding object code file for the assembly language program in the second pass.

THE TWO PASS ASSEMBLER

As mentioned above, in a two -pass assembler the assembler looks for errors in the first pass and generates tables and translates it into machine code in the second pass. This following section the contains a detailed explanation of our program.

We have made two major functions, one for the first pass and the other for the second pass.

In the first pass function reads the assemblyCode.txt file and the assembly code is iterated over line by line.

In the first pass the following tables are made:

a) The Symbol Table: The symbol table contains the value of all the symbols. A symbol is either a label or a variable. Our table contains the three columns one for symbols, and for its type (label or variable) and the remaining for its corresponding address.
b) The Label Table: The label table contains *only* the labels and has two columns. One for the label name and the other for its address.
c) The Literal Table: The literal table contains all the literals, i.e. all the constants for which our assembler reserves memory. Our table contains two columns, one for its name and the next for its corresponding value.
d) The Opcode Table: The opcode table contains all the valid opcodes. Our table has three columns, one for the opcode, one for the opcode's binary equivalent and the remaining for the operand corresponding to the specified opcode.

As and when we make the tables, we keep noting the errors in the program. Once we have made a note of all the errors along with its corresponding line numbers in the program, we print them out in the terminal. A Boolean flag variable is set in the beginning of the first pass, if any errors are encountered it changes to false. Our second pass runs on the basis of the flag variable. If there are no errors our second pass is called else, the program is terminated after the first pass itself

The following errors have been handled by us in the first pass:

- Invalid Label or variable Name
- Multiple definitions of the same label
- Using a label name as a variable name and vice-versa
- Invalid Opcode Name
- End of program not found
- Incorrect number of operands supplied to an opcode
- Memory Overflow
- Non- initialization of literals or variables
- Use of undefined label in branch statement
- No instruction Provided (empty line)
- Variable Name not defined yet used

In the function which performs the second pass, we translate the assembly program into its appropriate machine language counterpart with the help of the tables made in the first pass. We print the machine language of the assembly language statements along with its address.

ASSUMPTIONS MADE ABOUT ASSEMBLY LANGUAGE

In order to incorporate a fully functioning two-pass assembler, we have made the following assumptions about our assembly language.

- All opcodes have ***only one*** operand. The opcodes CLA and STP are the exceptions to this rule, they have ***no operands*** whatsoever (when there are no operands the translated version shall contain '00000000' ***in place*** of the address of the operand).
- When a label is used a ':' is used ***after*** the label.
- Every opcode and operand are separated by a ***single space***.
- Our assembly language allows ***only single line comments***, all comments start with a ';'
- The opcode DS stands for the ***declaration of a variable*** while DC stands for ***declaration of constants.***
- All variables and literals are to be declared at the ***end of the program*** but ***before the end statement***.
- R1 and R2 are ***reserved keywords*** they are also assumed to be variables and are placed in the ***symbol table***.

- It is necessary for all symbols and constants to be initialized
- The language is ***case sensitive*** (All opcodes are in ***capital letters.***)
- The addresses of the literal values are assigned when they are declared.
- All label and variable and literal names should ***always start with an alphabet*** and contain ***no special character***. The names can be ***alphanumeric*** however the first character should always be an alphabet. Also, it should ***neither be R1 or R2***nor any ***opcode***.
- There can be ***no more than one opcode*** in an assembly instruction.
- In the text file the ***line numbering starts from zero***.

PRE-REQUISITES FOR RUNNING THE PROGRAM

- Python 3.3 must be present in the user's device
- The user must have basic knowledge about assembly language and python
- The name of the file containing the assembly language code must be written in the python code. The file containing the code must be in the same directory or else the full address of the file is required to be mentioned.
- All assumptions regarding the assembly language must be kept in mind

DESCRIPTION OF CODE

We have stored all the given valid opcodes and assembly language statements in the form of a dictionary.

We have a class called 'OpcodeType'. Each object of this class is an entry in the opcode table. Each object gives us the opcode, it's binary equivalent and its operand (if any).

In our second class 'SymbolType', each object is an entry in the symbol table. Each object gives us the symbol, it's type and its address.

We also have a Boolean variable 'flag' which is initiated as True. The second pass is called on the basis of this variable.

Functions in code

- ***search_operand_in_symbolTable***: This function takes a symbol as a parameter and returns True if the symbol is in the symbol table otherwise returns false. This function helps avoid duplicate entries.
- ***give_address_from_symbolTable***: This function takes a symbol as a parameter and returns its corresponding address. If there is no address for that particular symbol it returns '00000000' this is in case of 'CLA' or 'STP' when the operand is null.
- ***check_valid_name***: This function checks whether or not the symbol or literal name is valid. It returns 'False' is invalid.
- ***line_length_1***: Handles all those lines whose length after splitting is 1. It first checks whether the operand is in the reference table. If not, it raises an error. It then checks that the operand is 'CLA' or 'STP'. If not, it raises an error.

- ***line_length_2***: Handles all those lines whose length after splitting is 2.
  It first checks whether the line has a label. If it does then it handles that the label name is not repeated. Otherwise it adds the label in the FoundLabel table. If the both the line to which the control is to be transferred and the line where label is defined have been encountered then that label is added to the symbol table.
  Then It the checks whether the line has a 'DS' or 'DC'. If it does then it accordingly adds a variable to the symbol table and a literal to the literal table after checking that their name is valid and not repeated. It also raises an error if literal or variable are not initialised.
  Then it checks that the opcode is valid and is not 'CLA' and 'STP'.
  If none of the above happens then the line must have a valid opcode with a variable or a literal or label as its operand. Hence it adds them to Opcode Table. In case branch statements it checks Found Labels table. If the label is not in that table then it adds it to the ToBeFound Label table
- ***line_length_3***: Handles all those lines whose length after splitting is 3. It first checks if a label is present in the line. If not, it raises an error. The it checks that the label name is unique and valid. Finally, it adds label to Found Labels table and the symbol table. It also adds the rest of the line to Opcode Table.
- ***firstPass***: This is one of the most important functions of our code. It takes the help of the above-mentioned functions and throws errors and builds all the required tables. It checks that for the memory overflow error and end of programs not found error. It also checks that none of the line is blank or empty. The global variable 'flag' comes into play in this function. Whenever an error is in encountered 'flag' turns false and thus the second pass is never called. At the end of transversal of all the lines it checks for any found labels which were not defined or any found variables or literals which were not defined.
- ***secondPass***: This function is called if an only if 'flag' is True (i.e. no errors are encountered in the first pass). This function does the translation, i.e. it takes help of the tables generated with the help of firstPass and creates a file containing the machine code of the assembly language provided to it.
- ***print_opcodeTable***: Prints the Opcode Table.
- ***print_symbolTable***: Prints the Symbol Table.
- ***print_literalTable***: Prints the Literal Table.
- ***print_labelTable***: Prints the Label Table.

Variables in The Code

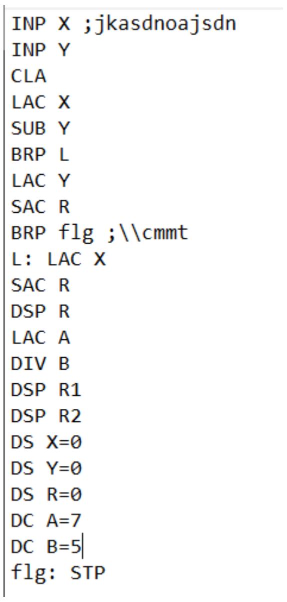
- refrenceTable: This is a dictionary where all the given valid opcodes and their corresponding binary equivalent are stored.
- foundLabel: It is basically the label table which is stored in the form of a dictionary.
- toBeFound: It is a list which stores all those labels which are yet to be found.
- toBefoundVar:It is a list which stores all those variables whose addresses are yet to be generated.

- aliteral: stores the address of corresponding literals in the form of a dictionary.
- flag: a global Boolean variable on the basis of which secondPass is called.
- assemblyCode = opens the file containing assembly code in ‘read only’ mode.
- listOfLines = Each item of the list is a line of the assembly code file in string format.
- opcodeTable: stores Opcode Table in form of a list.
- symbolTable: stores Symbol Table in form of a list.
- literalTable: stores Literal Table in the form of a dictionary.

PART II: WORKING

In order to show the working of our code, we have taken the following sample inputs.

<u>Sample Input: 1</u>



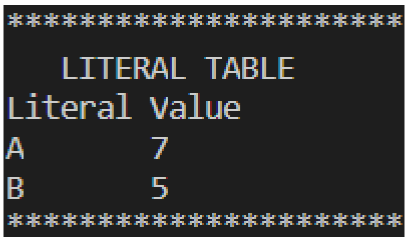
```
INP X ;jkasdnoajsdn
INP Y
CLA
LAC X
SUB Y
BRP L
LAC Y
SAC R
BRP flg ;\\cmmt
L: LAC X
SAC R
DSP R
LAC A
DIV B
DSP R1
DSP R2
DS X=0
DS Y=0
DS R=0
DC A=7
DC B=5
flg: STP
```

The following program is a code which is written in assembly language, it compares two integers and gives their maximum. This program also divides 7 by 5 and displays the quotient as well as the remainder.

This code shows how well our assembler handles forward referencing, variable declaration etc.

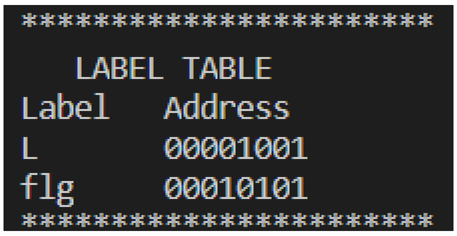
Sample Output:1

The following are the screenshots of the tables that have been generated for our sample input file in the first pass of our assembler's algorithm.



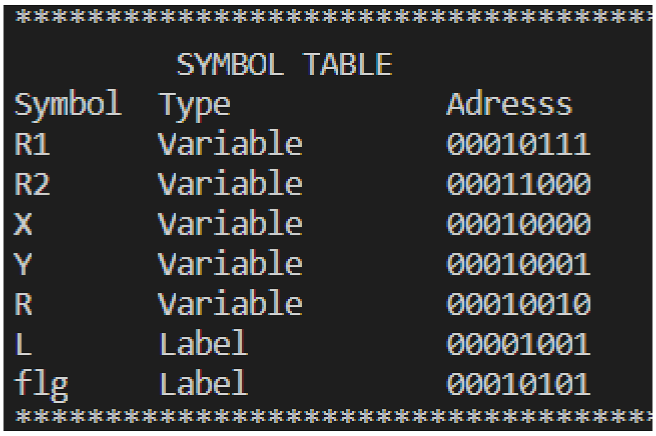
```
***********************
   LITERAL TABLE
Literal Value
A       7
B       5
***********************
```

As it can be seen, in the first pass our program has created the literal table and has correctly identified the literals.



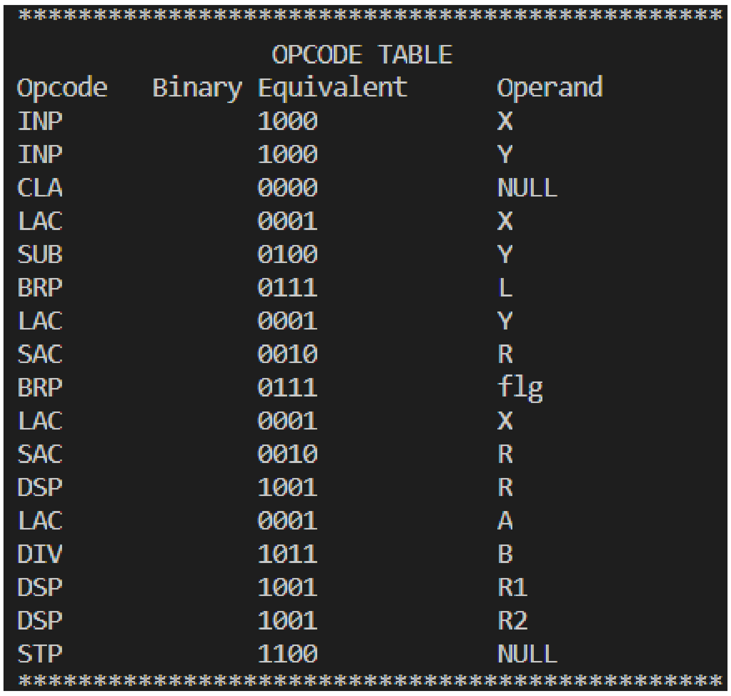
```
***********************
   LABEL TABLE
Label   Address
L       00001001
flg     00010101
***********************
```

As shown, the label table has also been, the three labels in the code have been identified and their addresses have been generated as well.



```
************************************
         SYMBOL TABLE
Symbol  Type            Adresss
R1      Variable        00010111
R2      Variable        00011000
X       Variable        00010000
Y       Variable        00010001
R       Variable        00010010
L       Label           00001001
flg     Label           00010101
************************************
```

The symbol table has also been generated, it shows the names of the symbols and states their types and has also recorded the addresses generated.



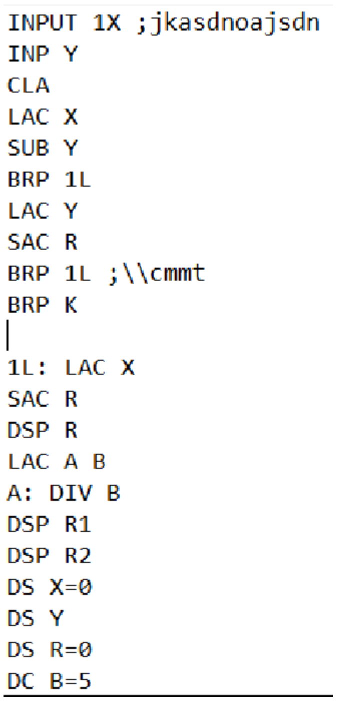
```
**************************************************
                  OPCODE TABLE
Opcode    Binary Equivalent       Operand
INP              1000             X
INP              1000             Y
CLA              0000             NULL
LAC              0001             X
SUB              0100             Y
BRP              0111             L
LAC              0001             Y
SAC              0010             R
BRP              0111             flg
LAC              0001             X
SAC              0010             R
DSP              1001             R
LAC              0001             A
DIV              1011             B
DSP              1001             R1
DSP              1001             R2
STP              1100             NULL
**************************************************
```

Finally, the opcode table has also been generated, as it can be seen. It contains the opcodes and its corresponding binary equivalent and also stores all operands corresponding to the specific opcode.

As no errors were encountered in the first pass the second pass function was called and the required object code file was generated as can be seen below.

```
1000 00010000
1000 00010001
0000 00000000
0001 00010000
0100 00010001
0111 00001001
0001 00010001
0010 00010010
0111 00010101
0001 00010000
0010 00010010
1001 00010010
0001 00010011
1011 00010100
1001 00010111
1001 00011000
1100 00000000
```

Sample Input: 2



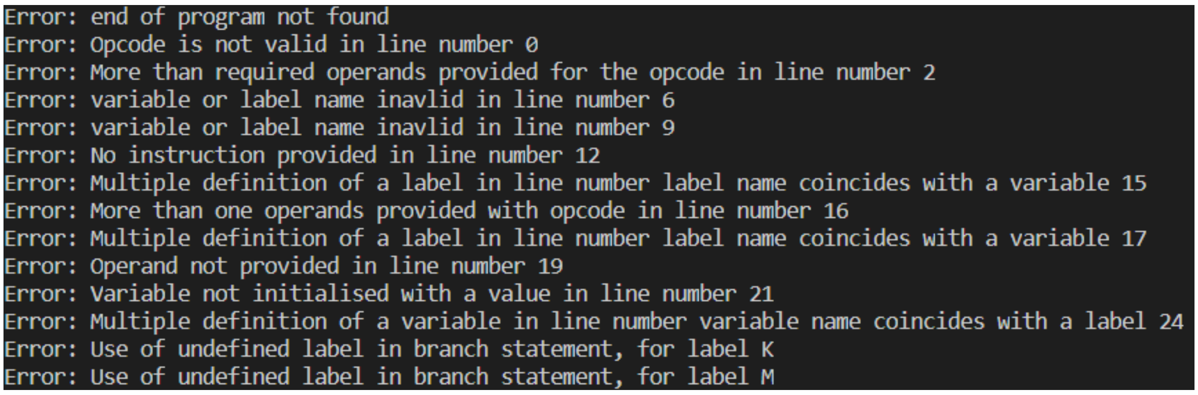
```
INPUT 1X ;jkasdnoajsdn
INP Y
CLA
LAC X
SUB Y
BRP 1L
LAC Y
SAC R
BRP 1L ;\\cmmt
BRP K

1L: LAC X
SAC R
DSP R
LAC A B
A: DIV B
DSP R1
DSP R2
DS X=0
DS Y
DS R=0
DC B=5
```

The following input is an example of an incorrect assembly code this input shall show how well our program handles all sorts of errors.

Sample Output: 2

The following output lists out all the errors in the given input file and as it can be seen it mentions the corresponding line numbers of the instructions with the errors. As it can be seen, the program terminates after the first pass and no object code is generated for an error filled input file.



```
Error: end of program not found
Error: Opcode is not valid in line number 0
Error: More than required operands provided for the opcode in line number 2
Error: variable or label name inavlid in line number 6
Error: variable or label name inavlid in line number 9
Error: No instruction provided in line number 12
Error: Multiple definition of a label in line number label name coincides with a variable 15
Error: More than one operands provided with opcode in line number 16
Error: Multiple definition of a label in line number label name coincides with a variable 17
Error: Operand not provided in line number 19
Error: Variable not initialised with a value in line number 21
Error: Multiple definition of a variable in line number variable name coincides with a label 24
Error: Use of undefined label in branch statement, for label K
Error: Use of undefined label in branch statement, for label M
```