Documentation

This following code implements a Two-Pass Assembler. The code is written in Python 3 and would require the user to have Python 3 or above set up in their system. It uses a given set of Opcodes and produces an output file with the generated machine language.

How to Run the Code:

1. Open the directory in which the code is saved.
2. Save the input as the file ‘input.assembly’.
3. Run the file ‘Assembler.py’
4. The output will be generated in ‘output.txt’
5. If there are errors it will show in the terminal.

Errors which are handled:

1. **Duplicate Label Error**:

   If a duplicate label has been given then an error will be thrown.

2. **Many Operands Error**:

   If an opcode is provided with many operands it handles the error thrown.

3. **Fewer Operands Error**:

   If an opcode is provided with lesser operands it handles the error thrown.

4. **Label not defined**:

   If Label is not defined then, the code throws an error.

5. **Opcode not Found**:

   If the Opcode of a given instruction is not found, then the code throws an error.

6. **Literal Error**:

   If find literal where label or variable is needed, then the code throws an error.

7. **Start Error**:

   If multiple start statements are found, the code throws an error.

8. **Variable Value**:

   If the variable value is used but not defined, the code throws an error.

9. **Endpoint not Found**:

   If the STP command is not found then, the code throws an error.

Some Basic Code Requirements:

1. The Label has to be defined by a colon (:) and the colon should not be separated by a space.
2. Comments are handled if the comment line starts with a hash (#) symbol. Inline comments are handled.
3. Literals are defined with a '=' before the value. They accept only numbers.
4. If START has not been declared explicitly then the code starts the address at 0, by default.
5. The components must be separated properly by a single space otherwise, opcode won't be generated.
6. Variables and Literals are assigned memory after the virtual address space for instruction is created.
7. The opcodes that perform branching function (BRZ, BRN, BRP) accept 'LABELS' as an argument and opcodes namely (INP, DSP, ADD, MUL, SUB, DIV) can accept variables or memory addresses (in decimals)
8. The memory inputs in the assembly code are accepted in decimal format i.e. Base 10
9. The assumption is made that 4-bit is the opcode and 12-bit is the memory size, which make the word 16-bit long.

Intermediate Outputs:

1. **Symbol Table**

   All the symbols, including labels and variables are stored in the symbol table. This is done in the first pass. This is a Hashmap in the code.
   Format: Name | Type | Offset | Value |

2. **Opcode Table**

The Opcode generated in the first pass is stored in the Opcode table. This is a 2D Array in the code.
Format: Ass. Code | OpCode | Offset | Argument | Opcode ID
Note:

- In cases where the argument is not specified '-1' is used
- Offset is the value after the START address

3. **Literal Table**

The literals are stored in this table and can be used in the first pass. This is stored as a hashmap.

SAMPLE CODE:

Input.txt

```
START 50
CLA
INP A
INP B
LAC 157
SUB C
BRN L1
DSP 157
CLA
BRZ L2
L1: DSP
158
CLA
BRZ L2
L2: STP
C DW 200
END
```

Ouput.txt

0000000000000000
1000000001000000
1000000001000001
0001000010011101
0100000001000010
0110000000111100
1001000000011110
0000000000000000
0101000000111111
1001000010011110
0000000000000000
0101000000111111
1100000000000000

Opcode Table:

Ass. Code	OpCode	Offset	Argument	OpcodeID
CLA	0000	0	-1	1
INP	1000	1	A	9
INP	1000	2	B	9
LAC	0001	3	157	2
SUB	0100	4	C	5
BRN	0110	5	L1	7
DSP	1001	6	30	10
CLA	0000	7	-1	1
BRZ	0101	9	L2	6
DSP	1001	10	158	10
CLA	0000	11	-1	1
BRZ	0101	12	L2	6
STP	1100	13	-1	13

Symbol Table:

Name	Type	Offset	Value
A	var	14	-
B	var	15	-
C	var	16	200
L1	symbol	10	-
L2	symbol	13	-

Literal Table:

Name	Offset