Documentation

Project 1 : Two Pass Assembler

Course: CSE112 - Computer Organisation

Project members:

1. Rohan Hiranandani - 2019324
2. Vaibhav Gupta - 2019341

How to use

- ❖ The input file should be placed in the same folder/directory as the source code
- ❖ The input file should be of .txt type
- ❖ The output , error and symbol table file will be created in the same directory
- ❖ User should not put any unnecessary white spaces in the assembly code:

  Eg:

  - 'INP 158' -> Valid

    'INP 158 ' -> Invalid

  - 'CLA

    DSP A' ->Valid

    'CLA

    DSP A' ->Invalid

Assumptions

OpCodes and Operands

Opcode	Meaning	Assembly Opcode
0000	Clear accumulator	CLA
0001	Load into accumulator from address	LAC
0010	Store accumulator contents into address	SAC
0011	Add address contents to accumulator contents	ADD
0100	Subtract address contents from accumulator contents	SUB
0101	Branch to address if accumulator contains zero	BRZ
0110	Branch to address if accumulator contains negative value	BRN
0111	Branch to address if accumulator contains positive value	BRP
1000	Read from terminal and put in address	INP
1001	Display value in address on terminal	DSP
1010	Multiply accumulator and address contents	MUL
1011	Divide accumulator contents by address content. Quotient in R1 and remainder in R2	DIV
1100	Stop execution	STP

- The above mentioned OpCodes are the only valid OpCodes.
- There are no macros or procedures to be assembled.
- All operands are declared at the end of instructions, sequentially and in the order of their appearance.

- ‘CLA’ & ‘STP’ take no operands.
- ‘LAC’ , ‘SAC’,’INP’,’DSP’ all take 1 operand as address.
- ‘ADD’,’SUB’,’MUL’,’DIV’ are mathematical operations taking one operand each which can be a literal or address.
- If ‘DIV’ is called then, the quotient goes to R1 and remainder to R2 respectively. If ‘DIV’ is called again then, the values of R1 and R2 will be overwritten.
- ‘LAC’ and ’INP’ can take undefined arguments, other OpCOdes cannot

General

- ‘CLA’ resets the accumulator.
- ‘STP’ instruction tells the assembler when to stop. It is necessary for assembly code to run.
- The assembler starts at the memory address ‘00000000’. ‘START’ and ‘END’ instructions not recognised.
- Program counter starts from 0.
- Label name cannot be a OpCode name.
- Comments can be added with a ***‘;’* or *‘//’*** at the beginning of the comment.
- Variable and label names can be alpha-numeric in nature. However, the same name cannot be used for a variable and label.
- If the size of the literal uses more than 1 memory space, then only the first is written in machine code.
- The maximum memory allowed is 256 words, which can store an 8-bit address.
- If the address exceeds this limit, then the assembler will flag an error.
- Total instruction length is 12-bits : 4-bit OpCode + 8-bit instruction address.

Design

Symbol Table

Contains all the numeric addresses, labels and symbols defined in the code

1. **name:** integer address value , label name or symbol name
2. **isUsed:** if it has been declared in code
3. **isFound:** if it has been called in code
4. **VariableAdd:** store variable address

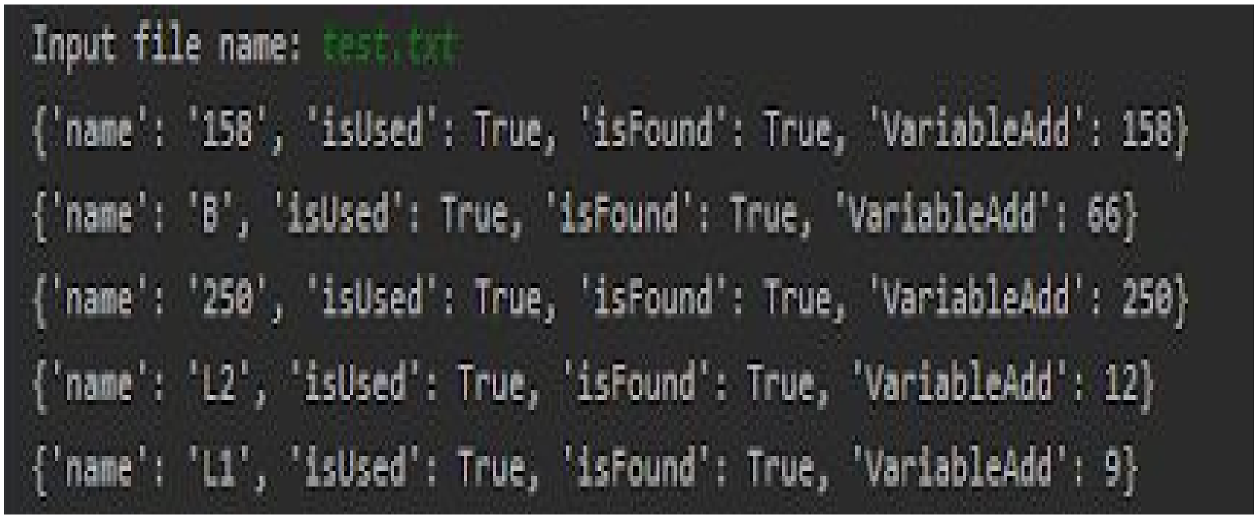
Example:

```
//Code Starts Here
CLA
INP 158
INP B
LAC 250
SUB B
BRN L2
DSP L2
CLA
BRZ L1
L1: DSP B
CLA
BRZ L2
L2: STP
```

name	isUsed	isFound	VariableAdd
‘158’	True	True	158
‘B'	True	True	66

‘250’	True	True	250
‘L2’	True	True	12
‘L1’	True	True	9

(Screen shot from compiler)

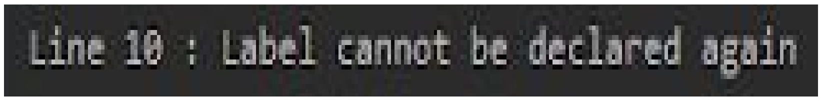


```
Input file name: test.txt
{'name': '158', 'isUsed': True, 'isFound': True, 'VariableAdd': 158}
{'name': 'B', 'isUsed': True, 'isFound': True, 'VariableAdd': 66}
{'name': '250', 'isUsed': True, 'isFound': True, 'VariableAdd': 250}
{'name': 'L2', 'isUsed': True, 'isFound': True, 'VariableAdd': 12}
{'name': 'L1', 'isUsed': True, 'isFound': True, 'VariableAdd': 9}
```

Error Handling

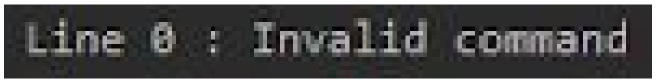
Errors handled in the first pass

- **Multiple declaration of a label**
  This error occurs when a label is defined/declared multiple times.
  Label declaration includes statements such as L1: SUB 220



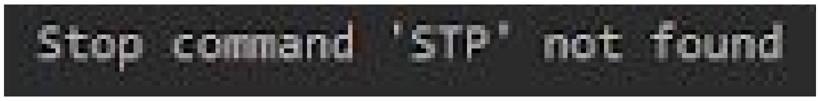
```
Line 10 : Label cannot be declared again
```

- **Invalid OpCode entered**
  This error occurs when an unrecognised OpCode is entered.
  Eg : CLR
        INP 27

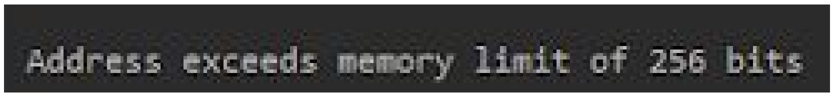


```
Line 0 : Invalid command
```

- **End of program not found**
  This error occurs when 'STP' command is not declared in code so the assembler does not know where the program ends.

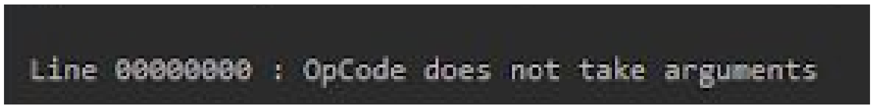

```
Stop command 'STP' not found
```

- **Address exceeds memory limits**
  Since the maximum size of the memory is 8 bit, the user cannot access memory cells greater than 256.
  This error is thrown if the user tries to access addresses greater than 256.
  Eg:
  ```
  CLA
  INP 25800
  ```

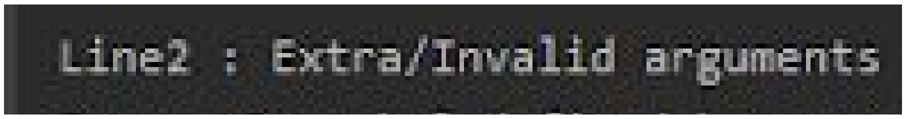

```
Address exceeds memory limit of 256 bits
```

Errors handled in the second pass

- **OpCode with invalid arguments**
  This error occurs when an OpCode with invalid arguments in entered into the code
  Eg:
  ```
  CLA 11
  ```
  Or
  ```
  L1: STP 12
  ```


```
Line 00000000 : OpCode does not take arguments
```

- **Exceeding instruction length**
  This error occurs when the 12 bit instruction length is exceeded.
  Eg:
  ```
  CLA
  BRP L1
  L1: INP P 12
  ```

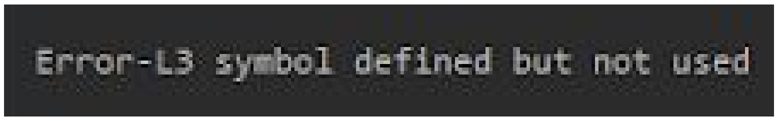

```
Line2 : Extra/Invalid arguments
```

- **Symbol table errors:**

- **Symbol used but not defined**
- **Symbol defined but not used**

Eg:

```
CLA
DSP A
L3: INP 150
STP
```



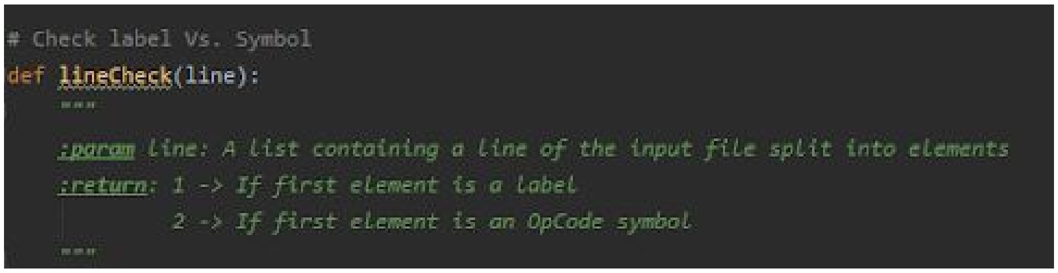
```
Error-L3 symbol defined but not used
```

Methods and Working

First Pass

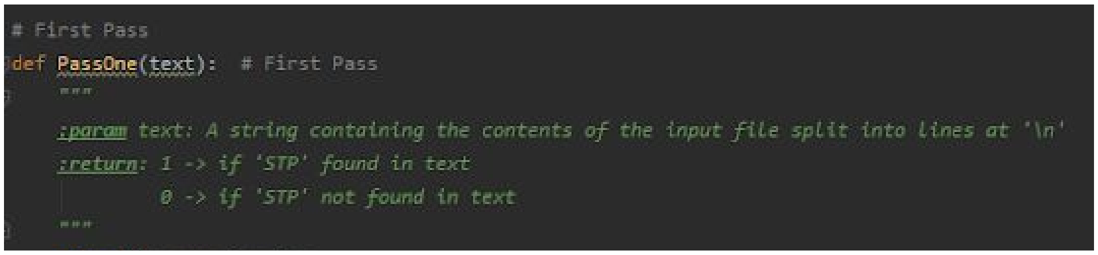
After input file is opened. The text is split into lines.
The lines are split into elements.

Check if the line contains a label or a symbol



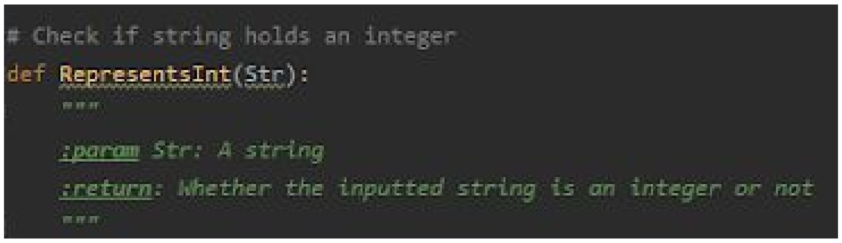
```
# Check label Vs. Symbol
def lineCheck(line):
    """
    :param line: A list containing a line of the input file split into elements
    :return: 1 -> If first element is a label
             2 -> If first element is an OpCode symbol
    """
```

Creates the symbol table and checks for the presence of 'STP'



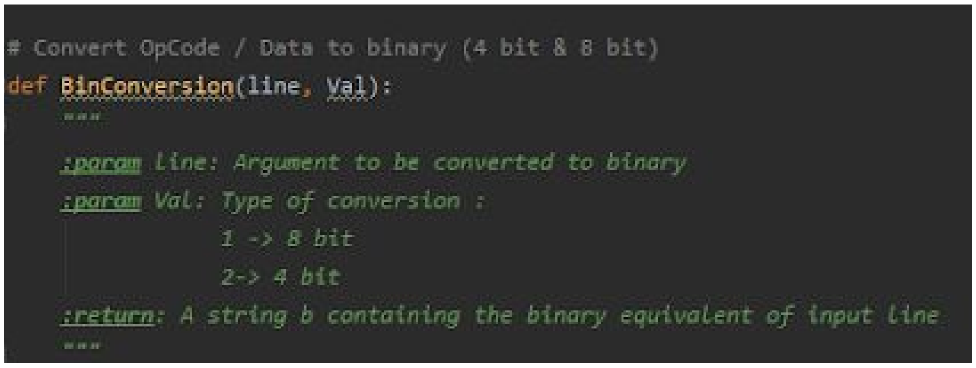
```
# First Pass
def PassOne(text):  # First Pass
    """
    :param text: A string containing the contents of the input file split into lines at '\n'
    :return: 1 -> if 'STP' found in text
             0 -> if 'STP' not found in text
    """
```

Second Pass



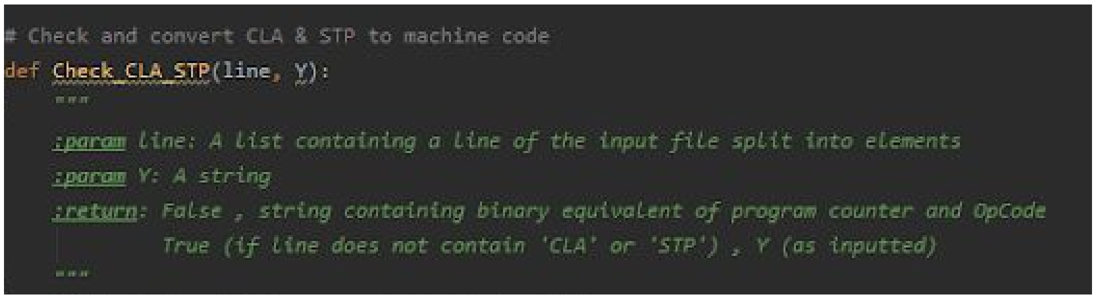
```
# Check if string holds an integer
def RepresentsInt(Str):
    """

    :param Str: A string
    :return: Whether the inputted string is an integer or not
    """
```



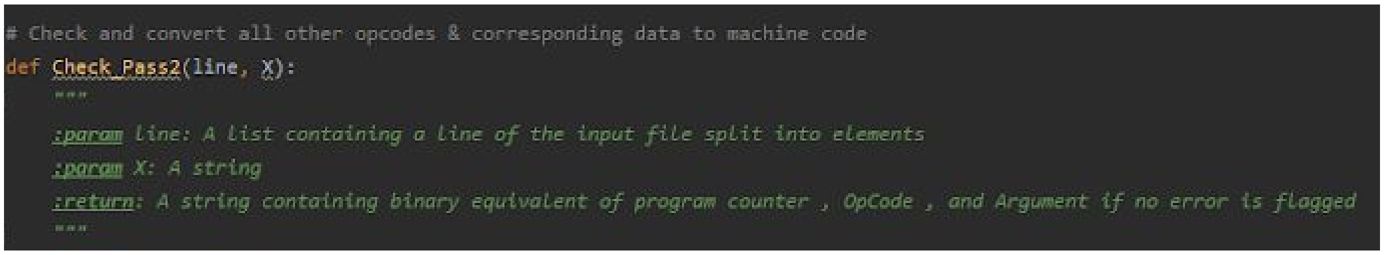
```
# Convert OpCode / Data to binary (4 bit & 8 bit)
def BinConversion(line, Val):
    """

    :param line: Argument to be converted to binary
    :param Val: Type of conversion :
                1 -> 8 bit
                2-> 4 bit
    :return: A string b containing the binary equivalent of input line
    """
```

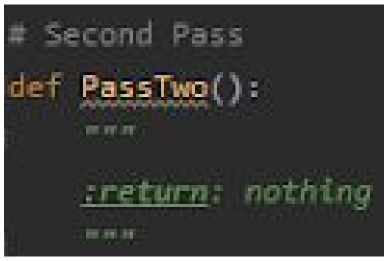


```
# Check and convert CLA & STP to machine code
def Check_CLA_STP(line, Y):
    """

    :param line: A list containing a line of the input file split into elements
    :param Y: A string
    :return: False , string containing binary equivalent of program counter and OpCode
             True (if line does not contain 'CLA' or 'STP') , Y (as inputted)
    """
```



```
# Check and convert all other opcodes & corresponding data to machine code
def Check_Pass2(line, X):
    """
    :param line: A list containing a line of the input file split into elements
    :param X: A string
    :return: A string containing binary equivalent of program counter , OpCode , and Argument if no error is flagged
    """
```



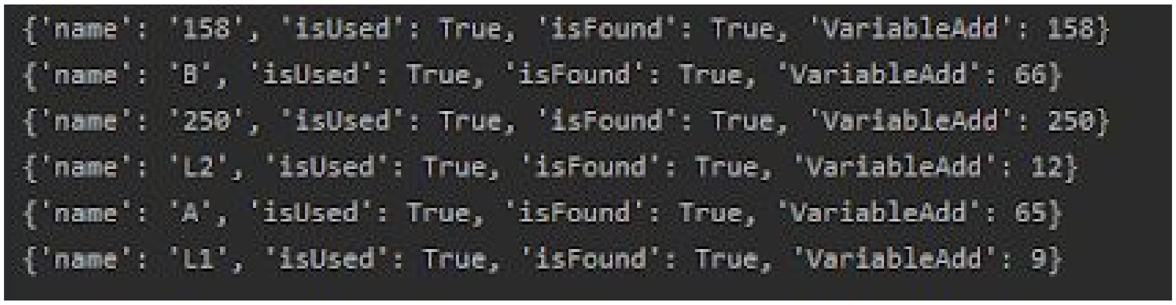
```
# Second Pass
def PassTwo():
    """
    :return: nothing
    """
```

Sample input and output

Assembly code

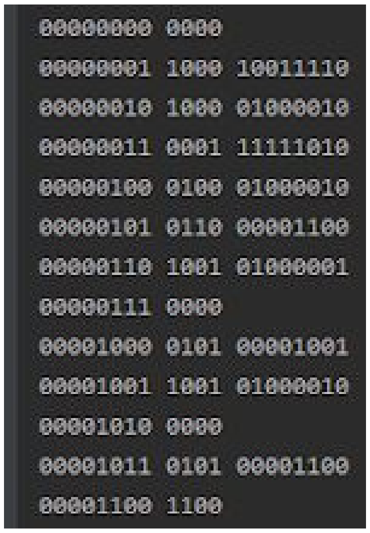
```
//Code Starts Here
CLA
INP 158
INP B
LAC 250
SUB B
BRN L2
DSP A
CLA
BRZ L1
L1: DSP B
CLA
BRZ L2
L2: STP
```

Symbol table



```
{'name': '158', 'isUsed': True, 'isFound': True, 'VariableAdd': 158}
{'name': 'B', 'isUsed': True, 'isFound': True, 'VariableAdd': 66}
{'name': '250', 'isUsed': True, 'isFound': True, 'VariableAdd': 250}
{'name': 'L2', 'isUsed': True, 'isFound': True, 'VariableAdd': 12}
{'name': 'A', 'isUsed': True, 'isFound': True, 'VariableAdd': 65}
{'name': 'L1', 'isUsed': True, 'isFound': True, 'VariableAdd': 9}
```

Machine code



```
00000000 0000
00000001 1000 10011110
00000010 1000 01000010
00000011 0001 11111010
00000100 0100 01000010
00000101 0110 00001100
00000110 1001 01000001
00000111 0000
00001000 0101 00001001
00001001 1001 01000010
00001010 0000
00001011 0101 00001100
00001100 1100
```