```
import time
import numpy as np
import os
import warnings
import string
import re

warnings.filterwarnings("ignore")

class Assembler:
'''
Definition: Assembler object takes a text file that includes MIPS instructions and
turn it into machine code and save it as a file

    Usage: Object = Assembler(source=SOURCE_FILE, target=TARGET_FILE)
    '''

    def __init__(self, **kwargs):

        # defult values of object
        self.checkPrepare = False
        self.programMemorySize = None
        self.programMemoryLocation = 0
        self.targetDirectory = None
        self.sourceDirectory = None
        self.content = None
        self.contentLabels = ["None"]
        self.machineCode = None
        self.machineCodeHex = None
        self.commentSeen = False
        self.previewDetailed = False
        self.previewLine = "all"
        self.previewHex = True
        self.checkSingleLineCommand = False
        self.checkConvertContent = False
        self.executeFormatHex = True
        self.executeFormatLineIndex = False
        self.errorMessage = None
        self.author = "DFA"
        self.name = "Kompaq"
        self.version = "1.0"
        self.description = self.name + " Assembler. Version is " + \
            self.version + ". It was developed by " + self.author

        for key, value in kwargs.items():
            if key == 'source':
                self.sourceDirectory = value
            elif key == 'target':
                self.targetDirectory = value

        self.registerFile = {
            "$zero": "00000",
```

```
            "$at": "00001",
            "$v0": "00010",
            "$v1": "00011",
            "$a0": "00100",
            "$a1": "00101",
            "$a2": "00110",
            "$a3": "00111",
            "$t0": "01000",
            "$t1": "01001",
            "$t2": "01010",
            "$t3": "01011",
            "$t4": "01100",
            "$t5": "01101",
            "$t6": "01110",
            "$t7": "01111",
            "$s0": "10000",
            "$s1": "10001",
            "$s2": "10010",
            "$s3": "10011",
            "$s4": "10100",
            "$s5": "10101",
            "$s6": "10110",
            "$s7": "10111",
            "$t8": "11000",
            "$t9": "11001",
            "$k0": "11010",
            "$k1": "11011",
            "$gp": "11100",
            "$sp": "11101",
            "$fp": "11110",
            "$ra": "11111"
        }

    def checkFiles(self):
        '''
        checkFiles function check if the input and output files are avaible

        return Boolean
        '''

        if self.checkSingleLineCommand:
            return True
        try:
            file = open(self.sourceDirectory)
            file.close()
            return True
        except:
            return False

    def getISA(self, *line):
        '''
        getISA function keeps all of MIPS instruction and returns given line as
machine code
```

```
        return String
        '''
        try:
            instructions = {
                "add": "000000{}{}{}00000100000".format(line[2], line[3],
line[1]),
                "addi": "001000{}{}{}".format(line[2], line[1],
self.convertSignedBinary(line[3], 16)),
                "addiu": "001001{}{}{}".format(line[2], line[1],
self.convertSignedBinary(line[3], 16)),
                "addu": "000000{}{}{}00000100001".format(line[2], line[3],
line[1]),
                "and": "000000{}{}{}00000100100".format(line[2], line[3],
line[1]),
                "andi": "001100{}{}{}".format(line[2], line[1],
self.convertSignedBinary(line[3], 16)),
                "beq": "000100{}{}{}".format(line[1], line[2],
self.convertLabel(line, 'I')),
                "bgez": "000001{}00001{}".format(line[1], self.convertLabel(line,
'I')),
                "bgezal": "000001{}10001{}".format(line[1],
self.convertLabel(line, 'I')),
                "bgtz": "000111{}00000{}".format(line[1], self.convertLabel(line,
'I')),
                "blez": "000110{}00000{}".format(line[1], self.convertLabel(line,
'I')),
                "bltz": "000001{}00000{}".format(line[1], self.convertLabel(line,
'I')),
                "bltzal": "000001{}10000{}".format(line[1],
self.convertLabel(line, 'I')),
                "bne": "000101{}{}{}".format(line[1], line[2],
self.convertLabel(line, 'I')),
                "div": "000000{}{}0000000000011010".format(line[1], line[2]),
                "divu": "000000{}{}0000000000011011".format(line[1], line[2]),
                "j": "000010{}".format(self.convertLabel(line, 'J')),
                "jal": "000011{}".format(self.convertLabel(line, 'J')),
                "jr": "000000{}00000000000000001000".format(line[1]),
                "lb": "100011{}{}{}".format(line[2][1], line[1], line[2][0]),
                "lui": "10001100000{}{}".format(line[1],
self.convertSignedBinary(line[2], 16)),
                "lw": "100011{}{}{}".format(line[2][1], line[1], line[2][0]),
                "mfhi": "0000000000000000{}00000010000".format(line[1]),
                "mflo": "0000000000000000{}00000010010".format(line[1]),
                "mult": "000000{}{}0000000000011000".format(line[1], line[2]),
                "multu": "000000{}{}0000000000011001".format(line[1], line[2]),
                "noop": "00000000000000000000000000000000",
                "or": "000000{}{}{}00000100101".format(line[2], line[3], line[1]),
                "ori": "0001101{}{}{}".format(line[2], line[1],
self.convertSignedBinary(line[3], 16)),
                "sb": "101000{}{}{}".format(line[2][1], line[1], line[2][0]),
                "sll": "00000000000{}{}{}000000".format(line[2], line[1],
self.convertSignedBinary(line[3], 5)),
                "sllv": "000000{}{}{}00000000100".format(line[3], line[2],
line[1]),
```

```
                "slt": "000000{}{}{}00000101010".format(line[2], line[3],
line[1]),
                "slti": "001010{}{}{}".format(line[2], line[1],
self.convertSignedBinary(line[3], 16)),
                "sltiu": "001011{}{}{}".format(line[2], line[1],
self.convertSignedBinary(line[3], 16)),
                "sltu": "000000{}{}{}00000101011".format(line[2], line[3],
line[1]),
                "sra": "00000000000{}{}{}000010".format(line[2], line[1],
self.convertSignedBinary(line[3], 5)),
                "srlv": "000000{}{}{}00000000110".format(line[2], line[3],
line[1]),
                "sub": "000000{}{}{}00000100010".format(line[2], line[3],
line[1]),
                "subu": "000000{}{}{}00000100011".format(line[2], line[3],
line[1]),
                "sw": "101011{}{}{}".format(line[2][1], line[1], line[2][0]),
                "syscall": "00000000000000000000000000001100",
                "xor": "000000{}{}{}00000100110".format(line[2], line[3],
line[1]),
                "xori": "00111000000{}{}{}".format(line[2], line[1],
self.convertSignedBinary(line[3], 16))
            }
            return instructions[line[0]]
        except:
            pass

    def clearCommas(self, line):
        '''
        clearCommas function clears the commas in the given line

        return Line(List)
        '''
        resultantLine = ["First"]
        for token in line:
            self.commentSeen = True if token == "#" else self.commentSeen
            if not self.commentSeen:
                resultantLine.append(token.replace(',', ''))
        resultantLine.pop(0)
        self.commentSeen = False
        return resultantLine

    def placeVariables(self, line):
        '''
        placeVariables function convert the tokens in the given line into CPU
varialbe
        if it is a CPU variable. Otherwise, turns the original value

        return Line(List)
        '''

        return list(map(lambda element: self.registerFile.get(element) if
self.registerFile.get(element) != None else element, line))
```

```
    def convertSignedBinary(self, number, width):
        '''
        placeVariables function convert the tokens in the given line into CPU
varialbe
        if it is a CPU variable. Otherwise, turns the original value

        return Token(String or None)
        '''

        try:
            return np.binary_repr(int(number), width=width)
        except:
            return False

    def convertOffset(self, token, *line):
        '''
        convertOffset function convert the tokens into array while placing its
memory address
        and decimal offset number to binary number

        return Token(String or List)
        '''

        tempToken = token.replace('(', ' ').replace(')', '')

        if tempToken == token:
            return token
        else:
            tempToken = list(tempToken.split())
            newToken = []
            newToken.insert(0, np.binary_repr(int(tempToken[0]), width=16))
            newToken.insert(1, self.registerFile.get(tempToken[1]))
            return newToken

    def placeOffsets(self, line):
        '''
        placeOffsets function pass tokens of the lines trough convertOffset
function

        return Line(List)
        '''
        return list(map(lambda element: self.convertOffset(element, *line), line))

    def fillInTheBlanks(self, line):
        '''
        fillInTheBlanks function fill the lines with 'None' item to keep
regularity
        '''
        newLine = line.copy()
        size = len(line)
        if size <= 4:
            for _ in range(4 - size):
                newLine.append('None')
        return newLine
```

```
    def takeLabels(self):
        '''
        takeLabels function takes the label in the content and stores it if exits

        Returns None
        '''

        for lineIndex, line in enumerate(self.content):
            if line[0].find(":") != -1:
                self.contentLabels.append(
                    [lineIndex, line[0].replace(":", "")])
                line.pop(0)
                if len(line) < 1:
                    self.content.pop(lineIndex)
            else:
                pass

        if len(self.contentLabels) != 1:
            self.contentLabels.pop(0)

        return None

    def convertLabel(self, line, type):
        '''
        convertLabel function convert labels in the instruction to binary values

        Returns String
        '''

        try:
            lineIndex = int(self.content.index(list(line)))
            labelIndex = None
            calculatedIndex = None

            if type == 'I':

                for index, label in self.contentLabels:
                    if label == line[3]:
                        labelIndex = index
                calculatedIndex = labelIndex - lineIndex - 1
                return self.convertSignedBinary(str(calculatedIndex), 16)

            elif type == 'J':

                for index, label in self.contentLabels:
                    if label == line[1]:
                        labelIndex = index
                calculatedProgramMemoryLocation = self.programMemoryLocation[2:]
                calculatedProgramMemoryLocation = int(
                    calculatedProgramMemoryLocation, 16)

                calculatedIndex = calculatedProgramMemoryLocation + 4 * labelIndex
                calculatedIndex = np.base_repr(calculatedIndex, base=2)
```

```
                calculatedIndex = np.base_repr(
                    int(calculatedIndex, 2), base=2, padding=26-
len(calculatedIndex))
                calculatedIndex = calculatedIndex[4:-2]

                return calculatedIndex
            else:
                return '0'
        except:
            return False

    def convertPseudoInstruction(self, line):
        '''
        convertPseudoInstruction function convert the pseudo instruction to
possible MIPS instruction if exits

        Returns Line(List)
        '''

        tempLine = ['None']
        if line[0] == 'move':
            tempLine.append('add')
            tempLine.append(line[1])
            tempLine.append(line[2])
            tempLine.append('00000')
            tempLine.pop(0)
        else:
            tempLine = line

        return tempLine

    def convertLineToBinary(self, line):
        '''
        convertLineToBinary function pass lines through getISA function

        Returns String
        '''

        return self.getISA(*line)

    def convertLineToHex(self, line):
        '''
        convertLineToHex function pass lines through getISA function

        Returns String
        '''

        try:
            hexValue = np.base_repr(int(self.getISA(*line), 2), base=16)
            hexValue = np.base_repr(
                int(self.getISA(*line), 2), base=16, padding=(8-len(hexValue)))
            return hexValue
        except:
            return 'errorAtHexConversion'
```

```
    def convertContent(self):
        '''
        convertContent function converts all content, saves it

        Returns Boolean
        '''
        if self.checkPrepare:

            if not self.checkConvertContent:

                self.machineCode = ["FirstElement"]
                self.machineCodeHex = ["FirstElement"]
                for line in self.content:
                    self.machineCode.append(self.convertLineToBinary(line))
                    self.machineCodeHex.append(self.convertLineToHex(line))
                self.machineCode.pop(0)
                self.machineCodeHex.pop(0)
            else:
                pass

            self.checkConvertContent = True
            return True
        else:
            return False

    def prepare(self):
        '''
        Definition: Prepare function transforms the given file into meaningful
data
        saving it into array while passing it through inherit functions \n
        Usage: Object.prepare()

        Returns Boolean
        '''
        if self.checkFiles():

            # reading the source file
            if not self.checkSingleLineCommand:
                fileContent = ['contentarray']
                with open(self.sourceDirectory, 'r') as file:

                    lines = filter(None, (line.rstrip() for line in file))

                    jInstruction = False
                    jalInstruction = False

                    for line in lines:
                        line = ''.join(re.findall(
                            r"^[a-zA-Z0-9,#:$\(\)\+\-\s]+", line))

                        line = line.lower()

                        if line[:2] == 'j ':
```

```
                            jInstruction = True
                            line = line[2:]
                        elif line[:4] == 'jal ':
                            jalInstruction = True
                            line = line[4:]
                        else:
                            pass

                        line = line.replace(' ', '').replace(
                            ':', ': ').replace(',', ', ').replace('#', ' #
').replace('$', ' $').replace('( $', '($')

                        if jInstruction:
                            line = 'j ' + line
                        elif jalInstruction:
                            line = 'jal ' + line
                        else:
                            pass

                        jInstruction = False
                        jalInstruction = False

                        fileContent.append(line.split())

                    fileContent.pop(0)
                self.content = fileContent

            # taking program memory location from the file if exist
            if self.content[0][0].find('0x') != -1:
                self.programMemoryLocation = self.content[0][0]
                self.content.pop(0)

            self.takeLabels()

            # preparing content for execution
            self.content = list(
                map(lambda line: self.clearCommas(line), self.content))
            self.content = list(
                map(lambda line: self.placeVariables(line), self.content))
            self.content = list(
                map(lambda line: self.placeOffsets(line), self.content))
            self.content = list(
                map(lambda line: self.fillInTheBlanks(line), self.content))
            self.content = list(
                map(lambda line: self.convertPseudoInstruction(line),
self.content))

            self.checkPrepare = True
            return True
        else:
            self.checkPrepare = False
            return False

    def preview(self, **kwargs):
```

```
        '''
        Definition: Preview function monitors required steps of process on
terminal \n
        Usage: Object.preview(
            line = ["all", lineNumber], detailed = [True, False])
        '''
        if(self.checkPrepare):

            self.convertContent()

            for key, value in kwargs.items():
                if key == "detailed":
                    self.previewDetailed = value
                if key == "line":
                    self.previewLine = value
                if key == "hex":
                    self.previewHex = value

            print("Program Memory Location: ", self.programMemoryLocation)

            if self.errorMessage:
                print("Error Message: ", self.errorMessage)

            if self.previewDetailed:
                print("------------- Detailed View")
                for lineIndex, line in enumerate(self.content):
                    print(lineIndex, line)

            if self.previewHex:
                previewContent = self.machineCodeHex
            else:
                previewContent = self.machineCode

            print("------------- Assembled Code")
            if self.previewLine == "all":
                for lineIndex, line in enumerate(previewContent):
                    print(lineIndex, line)
            elif isinstance(self.previewLine, int):
                print(previewContent[self.previewLine])
            else:
                pass

            return True
        else:
            return False

    def execute(self):
        '''
        Definition: Execute function execute all needed process and save the
machine code file
        into given target file \n
        Usage: Object.execute()
        '''
        if(self.checkPrepare):
```

```
            self.convertContent()

            with open(self.targetDirectory, "w+") as file:
                executeContent = self.machineCodeHex if self.executeFormatHex else
self.machineCode
                for lineIndex, line in enumerate(executeContent):
                    if self.executeFormatLineIndex:
                        file.write(str(lineIndex) + " " + line + "\n")
                    else:
                        file.write(line + "\n")

            return True
        else:
            return False

def main():

    BASE_DIR = os.path.dirname(os.path.abspath(__file__))

    assembler = Assembler()
    print(assembler.description)

    while True:

        command = input(">> ")
        singleLineCommand = []
        singleLineCommand.insert(0, list(command.split()))
        command = list(command.split())

        if command[0] == "debugmode":
            assembler.sourceDirectory = BASE_DIR + '/' + "code.src"
            assembler.targetDirectory = BASE_DIR + '/' + "result.obj"
            assembler.prepare()
            print(assembler.contentLabels, assembler.errorMessage)
            assembler.execute()
            assembler.preview(detailed=True)
            exit()

        elif command[0] == "source":
            assembler.sourceDirectory = BASE_DIR + '/' + command[1]

        elif command[0] == "target":
            assembler.targetDirectory = BASE_DIR + '/' + command[1]

        elif command[0] == "execute":
            if assembler.execute():
                print("Executing process is done!")
            else:
                print('First, you need to use "prepare" command!')

        elif command[0] == "prepare":
            if assembler.prepare():
                print("Preparing process is done!")
```

```
            else:
                print("There is a problem with files!")

        elif command[0] == "preview":
            if assembler.preview(detailed=False):
                pass
            else:
                print('First, you need to use "prepare" command!')

        elif command[0] == "clear":
            os.system('cls' if os.name == 'nt' else 'clear')

        elif command[0] == "exit":
            exit()

        else:
            temp = Assembler()
            temp.checkSingleLineCommand = True
            temp.content = singleLineCommand
            temp.prepare()

            tempMachineCode = temp.convertLineToHex(temp.content[0])

            if tempMachineCode:
                print(tempMachineCode)
            else:
                print("Invalid command!")

            del temp

if **name** == '**main**':
    main()
```