CNG 331 – COMPUTER ORGANIZATION TERM PROJECT

ASSEMBLER

Muhammad Khizr Shahid - 2413235

Fatma Erem Aksoy – 2315075

Introduction:

The problem to be solved in this project was to design an assembler which would convert any MIPS assembly program containing some of main MIPS instructions and pseudo-instructions to hexadecimal machine language or object code. Assembler was supposed to support 2 modes; interactive mode and batch-mode. Both the modes were supposed to take commands from the command prompt (cmd). Interactive modes reads an instruction from command line then assembles it to hexadecimal and outputs the result to screen whereas the batch-mode reads a source file with extension **.s**, then assembles to hexadecimal and outputs the result to an object code file with extension **.o** so the problem basically was to create an assembler which provides a solution to all these instructions.

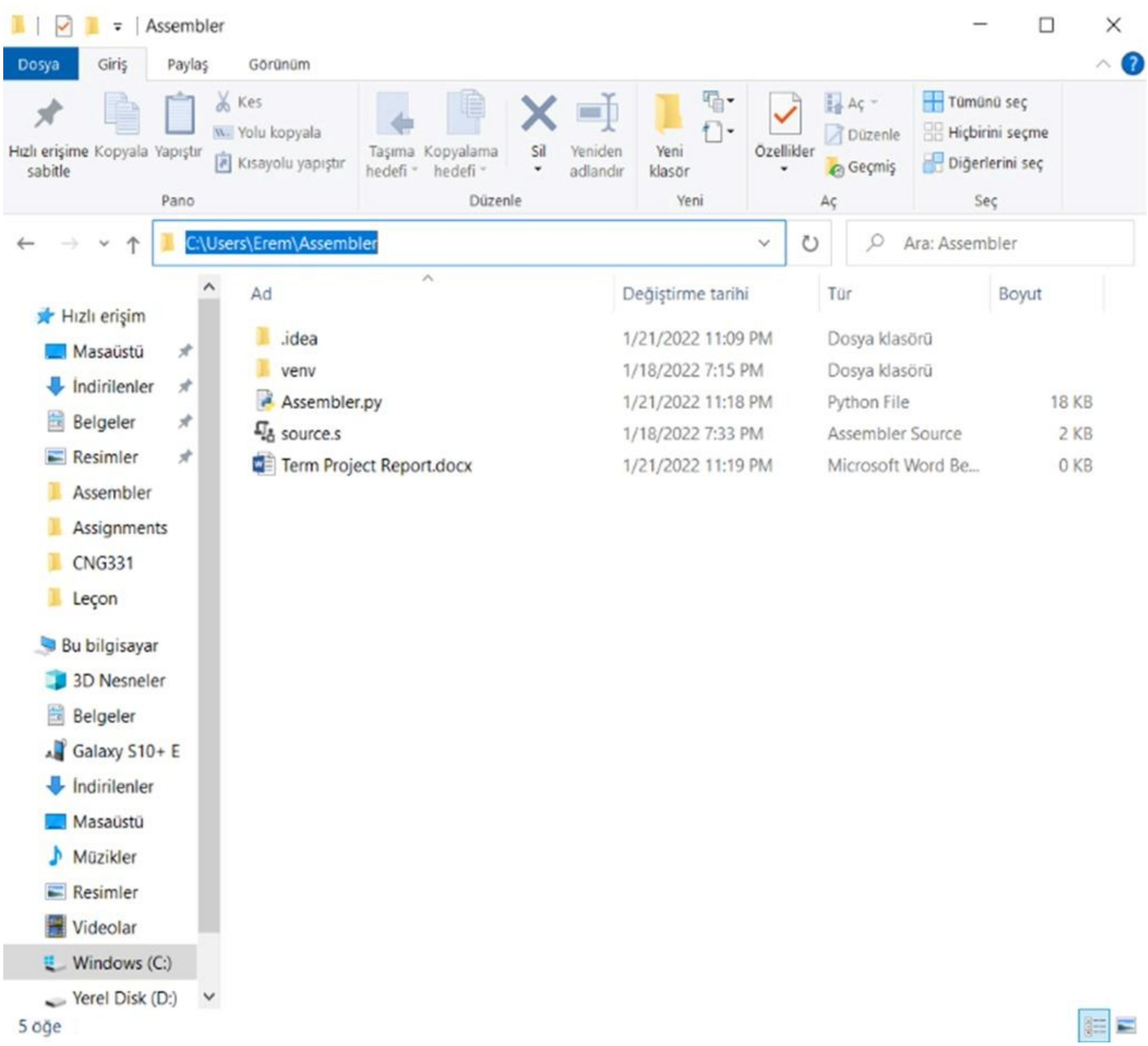
Choices, tools and testing:

To program the assembler, we have used Python programming language as it is very user-friendly, diversified, relatively easy to use and provides a wide range of functionalities such as OOP etc which can be used to improve the code and make it more efficient. One other big reason to use Python is that Python is a simple language so we wanted to implement a hardware related project such as assembler using a simple language rather than a high level complex language. We have tested this code on PyCharm as it is the most famous software for Python. In order to keep everything simple and easily accessible, we have provided all the functions, lookup tables and everything related to code in one single file so everything is easily accessible without opening multiple files.

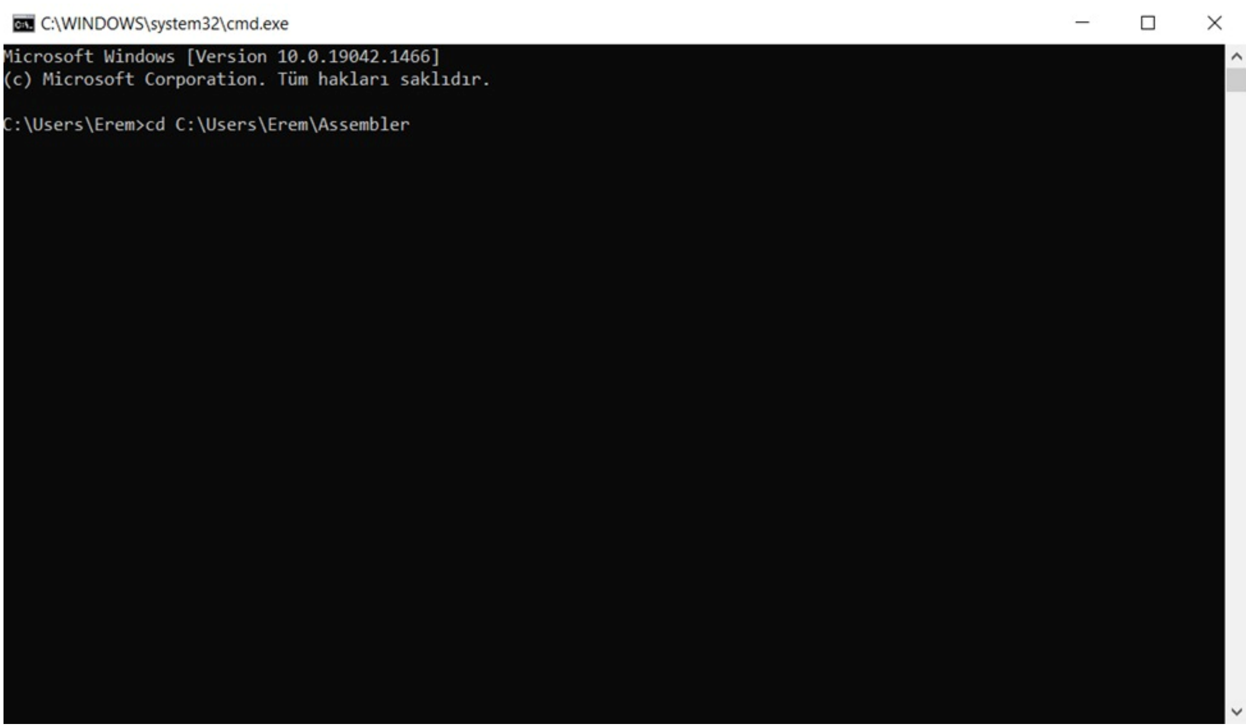
Guide:

Below is a step by step guide to launch and run each mode of the assembler:

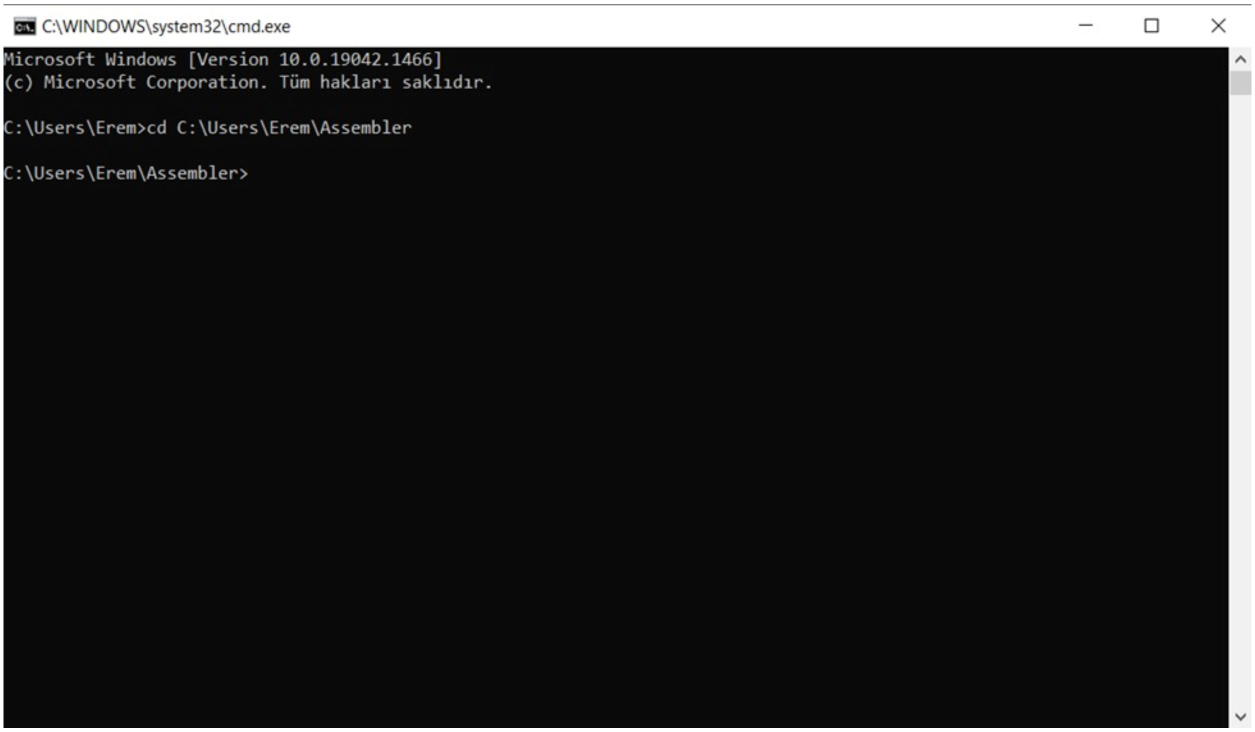
1) Download the folder called Assembler.zip and extract it. You can use any extractor to do that. We have used WinRAR.
   Below is an image indicating what you'll be able to see when you extract it.



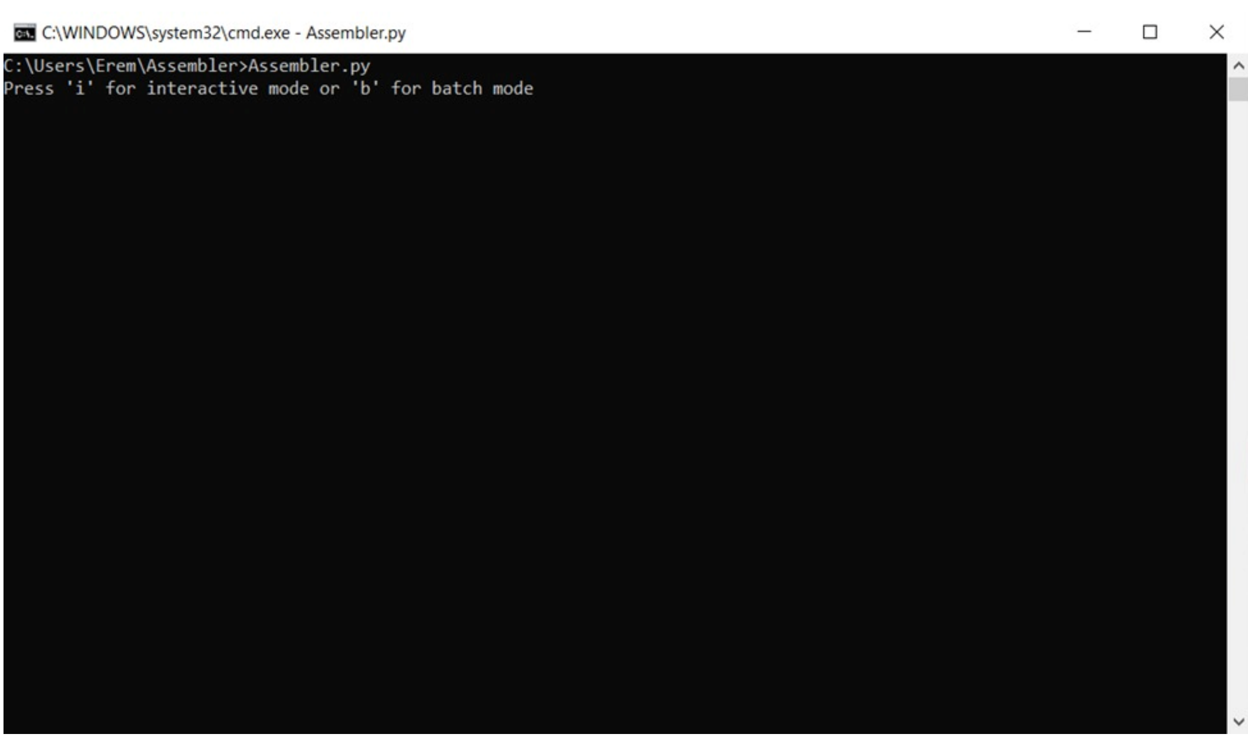
2) Copy the location of the folder which you have extracted. Open command prompt, and paste the location of the folder in the following way.
   cd *space* (location of folder)
   You'll be getting something similar to what we have shown in the image below.



3) After step 2, you'll have command of that folder and it will be indicated by something similar shown below.



4) Now when you enter ‘Assembler.py’, you’ll basically be able to run the assembler project which will be indicated by a statement show below in the image.

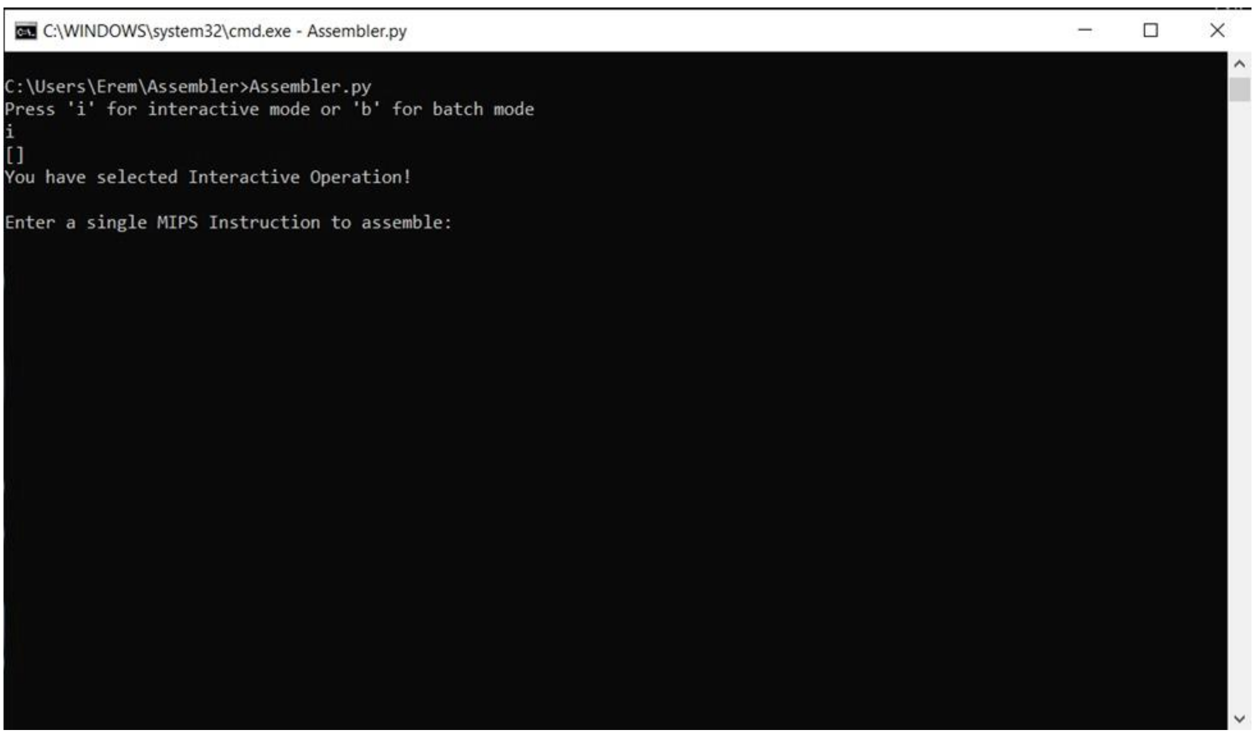


5) From this point onwards, you'll have to choose one of the methods i.e Batch mode or interactive mode to proceed.

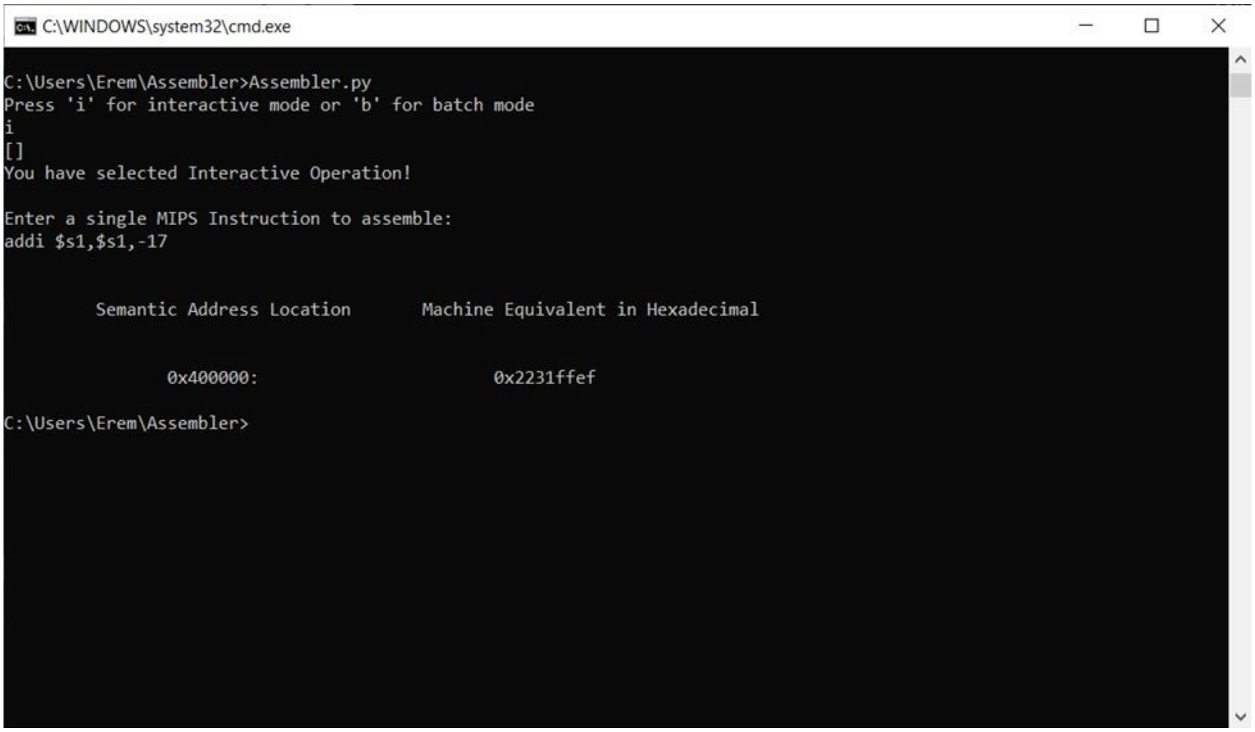
Interactive mode:

In this section, we provide the functionality of interactive mode with an example of simple assembly instruction.

6) In order to access Interactive mode, user needs to enter "i". Pictorial representation of this is shown below.



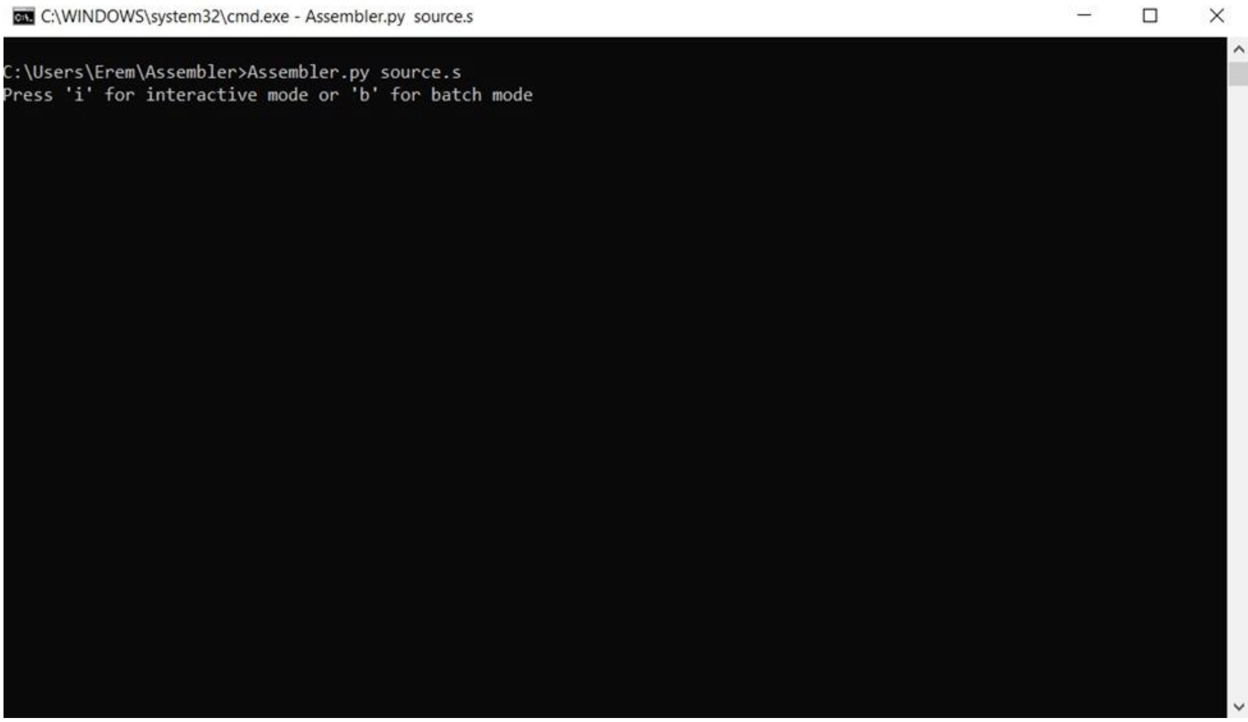
7) User is then required to enter an assembly instruction. We indicate the functionality of this mode with the instruction "addi $s1, Ss1, -17". Output of this instruction can be seen below.



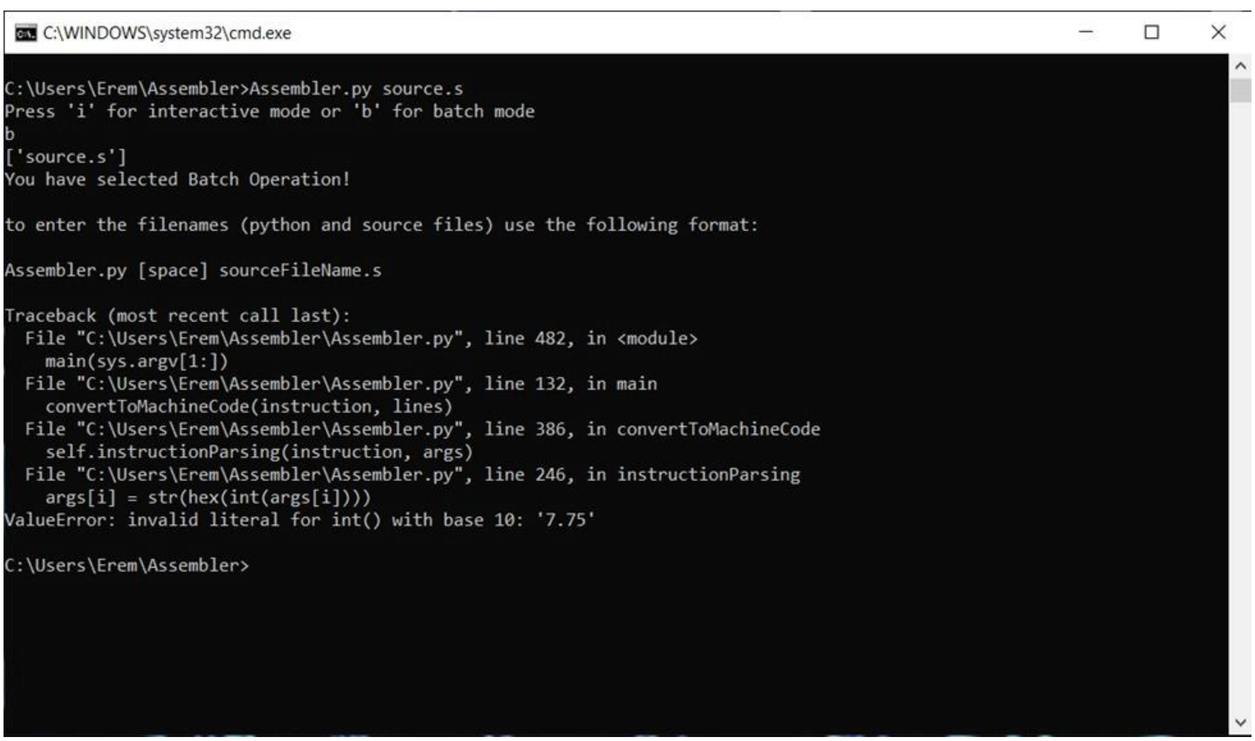
Batch mode:

In this section, we provide the functionality of batch mode.

8) We have added a feature that along with the name of the project user can directly add the name of the input file “source.s” as file in order to load both things simultaneously as shown below.



9) The user is now supposed to enter “b” if they want to access the batchmode which can be shown in the image below.



Unfortunately, due to some errors we weren't able to make this mode work 100% but if everything had gone fine, we would have been able to get and output.o file in the same folder as the Assembler.py.

Content of Assembler.py file:

```
'''  Team Members: Fatma Erem Aksoy - 2315075
                Muhammad Khizr Shahid - 2413235  '''

#We import system libraries.
import sys
import re

# Instructions and their equivalent 6-bits opcode in the format of: Funct/Shamt/Immediate (in Hexadecimal)

# instruction format length
# 6 : R instructions
# 4 : I instructions
# 2 : Jump instructions

instructionDefinitions = {
    'nor': ['0x00', 'rs', 'rt', 'rd', 'shamt', '0x27'],
    'or': ['0x00', 'rs', 'rt', 'rd', 'shamt', '0x25'],
    'ori': ['0x0D', 'rs', 'rt', 'imm'],
    'beq': ['0x04', 'rs', 'rt', 'imm'],
    'bne': ['0x05', 'rs', 'rt', 'imm'],
    'j': ['0x02', 'add'],
    'jal': ['0x03', 'add'],
    'jr': ['0x00', 'rs', 'rt', 'rd', 'shamt', '0x08'],
    'add': ['0x00', 'rs', 'rt', 'rd', 'shamt', '0x20'],
    'addi': ['0x08', 'rs', 'rt', 'imm'],
    'addiu': ['0x09', 'rs', 'rt', 'imm'],
    'addu': ['0x00', 'rs', 'rt', 'rd', 'shamt', '0x21'],
    'sw': ['0x2B', 'rs', 'rt', 'imm'],
    'lw': ['0x23', 'rs', 'rt', 'imm'],
```

```
    'and': ['0x00', 'rs', 'rt', 'rd', 'shamt', '0x24'],
    'andi': ['0x0C', 'rs', 'rt', 'imm'],
    'sb': ['0x28', 'rs', 'rt', 'imm'],
    'sc': ['0x38', 'rs', 'rt', 'imm'],
    'sh': ['0x29', 'rs', 'rt', 'imm'],
    'slt': ['0x00', 'rs', 'rt', 'rd', 'shamt', '0x2A'],
    'sll':['0x00','rs','rt','rd','shamt','0x00'],
    'slti': ['0x0A', 'rs', 'rt', 'imm'],
    'sltiu': ['0x0B', 'rs', 'rt', 'imm'],
    'sltu': ['0x00', 'rs', 'rt', 'rd', 'shamt', '0x2B'],
    'sub': ['0x00', 'rs', 'rt', 'rd', 'shamt', '0x22'],
    'subu': ['0x00', 'rs', 'rt', 'rd', 'shamt', '0x23'],
    'lbu': ['0x24', 'rs', 'rt', 'imm'],
    'lhu': ['0x25', 'rs', 'rt', 'imm'],
    'lui': ['0x0F', 'rs', 'rt', 'imm'],
}

# Names of the registers and their 5-bits address (in Decimal):
# Definitions of Registers:
registerDefinitions = {
    '$zero': 0,
    '$a0': 4,
    '$a1': 5,
    '$a2': 6,
    '$a3': 7,
    '$at': 1,
    '$s0': 16,
    '$s1': 17,
```

```
    '$s2': 18,
    '$s3': 19,
    '$s4': 20,
    '$s5': 21,
    '$s6': 22,
    '$s7': 23,
    '$t0': 8,
    '$t1': 9,
    '$t2': 10,
    '$t3': 11,
    '$t4': 12,
    '$t5': 13,
    '$t6': 14,
    '$t7': 15,
    '$t8': 24,
    '$t9': 25,
    '$k0': 26,
    '$k1': 27,
    '$gp': 28,
    '$sp': 29,
    '$fp': 30,
    '$ra': 31,
    '$v0': 2,
    '$v1': 3,
}
```

```python
# Names and the formats of the pseudo instructions:
pseudoInstructionDefinitions = {
    'la':['lui', 'ori'],
    'move': ['addu'],
    'ble': ['slt', 'bne'],
    'bgt': ['slt', 'bne'],
    'bge': ['slt', 'beq'],
    'sgt':['slt'],
}

# The input 'i' or 'b' will be taken from the user in terms of their choice of the operation
operation = str(input("Press 'i' for interactive mode or 'b' for batch mode \n"))


# Main Function:
def main(argv):
    # Interactive Operation
    if operation == 'i':
        print("You have selected Interactive Operation!\n")

        # As the interactive mode is selected by user, we'll take only one instruction from the user
        # Memory size needed to store the machine code will also be calculated

        MIPS = input("Enter a single MIPS Instruction to assemble: \n")

        instruction = assemblyAnalyzer(4194304, instructionDefinitions, registerDefinitions, pseudoInstructionDefinitions)
        instruction.buildMemory([MIPS])
        convertToMachineCode(instruction, [MIPS])
```

```python
    # Batch Operation
    elif operation == 'b':
        File = argv
        print("You have selected Batch Operation!\n")
        print("to enter the filenames (python and source files) use the following format:\n")
        print("Assembler.py [space] sourceFileName.s\n")

        # As the batch operation is selected, we load the list of all MIPS instructions from the source file
        # Memory required for storing the machine code is found
        # To find the corresponding machine instruction, we parse each MIPS instruction one by one with the help of below functions

        for filename in File:
            codeLines = open(filename)
            lines = codeLines.readlines()
            instruction = assemblyAnalyzer(64, instructionDefinitions, registerDefinitions, pseudoInstructionDefinitions)
            instruction.buildMemory(lines)
            convertToMachineCode(instruction, lines)

    # Checks if the user enters any value other than i or b
    elif operation != 'b' and operation != 'i':
        print("Invalid operation entered, please enter either 'b' or 'i' !")
        exit()

# Parsing Part
class Command(object):
    def __init__(self, commandstr):
        return
```

```python
class assemblyAnalyzer(object):
    # Current location in memory
    currentLoc = 0

    # Default memory location
    defaultMemLoc = 0

    # List of labels and their respective locations
    labelLoc = {}

    # Memory locations and their values
    memLoc = {}

    # Parsed Instruction Table Initialization
    instructionDefinitions = {}

    # Parsed Register Table Initialization
    registerDefinitions = {}

    # Parsed Pseudo instruction Table Initialization
    pseudoInstructionDefinitions = {}

    # Result List Initialization
    updatedList = []
    def __init__(self, defaultMemLoc, instructionDefinitions, registerDefinitions, pseudoInstructionDefinitions):
        # Memory initializations
        self.defaultMemLoc = defaultMemLoc
        self.instructionDefinitions = instructionDefinitions
        self.registerDefinitions = registerDefinitions
```

```python
        self.pseudoInstructionDefinitions = pseudoInstructionDefinitions

    def buildMemory(self, lines):
        # Rearrange the instructions to calculate required memory size
        self.currentLoc = self.defaultMemLoc
        for line in lines:
            # Dealing with unnecessary parts in the string
            if '#' in line:     #dealing with comments
                line = line[0:line.find('#')]
            line = line.strip()
            if not len(line):   #dealing with spaces
                continue

            # Checking the memory locations
            checkWordSize(self)

            # Reading the labels
            if ':' in line:
                label = line[0:line.find(':')]
                self.labelLoc[label] = str(self.currentLoc)
                line = line[line.find(':') + 1:].strip()

            checkWordSize(self)

            # Parsing instructions by splitting them with the comma
            instruction = line[0:line.find(' ')]
            args = line[line.find(' ') + 1:].replace(' ', '').split(',')
            if not instruction:
                continue
```

```python
            count = 0
        self.currentLoc += findInstructionSize(self, instruction, args)

    def instructionParsing(self, instruction, arguments):
        machineCode = self.instructionDefinitions[instruction]
        counter = 0  #counts the number of the arguments
        offset = 'not_valid'
        args = arguments[:]
        for arg in args:
            if '(' in arg:
                # parsing 'not valid'
                offset = hex(int(arg[0:arg.find('(')]))
                register = re.search('\((.*)\)', arg)

                # Location in memory is offset from memory location
                location = self.registerDefinitions[register.group(1)]
                register = location

                # Argument processing finishes
                args[counter] = register

            elif arg in self.registerDefinitions.keys():
                # Putting the corresponding value from the table given at the beginning for the symbol
                args[counter] = int(self.registerDefinitions[arg])

            elif arg in self.labelLoc:
                # Replacing label with its corresponding value
                args[counter] = self.labelLoc[arg]
```

```python
            # Incrementing counter to modify the argument list
            counter += 1

        # For branch instructions, we look at the location
        if instruction == 'beq' or instruction == 'bne':
            args[2] = (int(args[2]) - self.currentLoc + 4) / 4

        # For jump instructions, we know that their values are always location(location of the word)/4
        if instruction == 'j' or instruction == 'jal' or instruction == 'jr':
          #  print('\n this is: ' + args[0])
            args[0] = str(int(int(args[0])) / 4)

        # Converting the integer values to their hexadecimal equivalent
        for i in range(0, len(args)):
            args[i] = str(hex(int(args[i])))

        # R instructions with opcode of 6-bits
        if len(machineCode) == 6:
            # Initializing the addresses for destination and source registers
            rt = '0'
            rd = '0'
            rs = '0'
            if len(args) == 1:
                rs = args[0]
            else:
                # In the machine code, we are setting the values of offset, rt, rd, and rs
                rt = args[2]
                rs = args[1]
                rd = args[0]
```

```
        machineCode[1] = rs
        machineCode[2] = rt
        machineCode[3] = rd
        machineCode[4] = '0'

        # Finding the binary representation of machine code
        opcodeInBinary = self.hexToBinary(machineCode[0], 6)
        rsInBinary = self.hexToBinary(machineCode[1], 5)
        rtInBinary = self.hexToBinary(machineCode[2], 5)
        rdInBinary = self.hexToBinary(machineCode[3], 5)
        shamtBinary = self.hexToBinary(machineCode[4], 5)
        functBinary = self.hexToBinary(machineCode[5], 6)

        # Creating a string of 32-bits
        strInBinary = opcodeInBinary + rsInBinary + rtInBinary + rdInBinary + shamtBinary + functBinary
        self.bitStoredString(strInBinary, instruction, arguments)

        return

    # I instructions with opcode of 4-bits
    if len(machineCode) == 4:
        # Setting the values of rs, rt, immediate in the machineCode
        rt = args[0]
        rs = args[1]
        immediate = offset

        # Checking if it is one of the andi/addi no offset immediate instruction
        if len(args) == 3:
            immediate = hex(int(args[2], 16))
```

```
        # Checking if it is one of the lui/li no offset, no rs type of instructions
        elif immediate == 'not_valid':
            immediate = args[1]
            rs = '0'
        machineCode[1] = rs
        machineCode[2] = rt
        machineCode[3] = immediate

        # Here, we are getting the machine code in binary
        opcodeInBinary = hexToBinary(self, machineCode[0], 6)
        rsInBinary = hexToBinary(self, machineCode[1], 5)
        rtInBinary = hexToBinary(self, machineCode[2], 5)
        immediateInBinary = hexToBinary(self,machineCode[3], 16)

        # Creating a new string of 32-bits to divide into bytes
        strInBinary = opcodeInBinary + rsInBinary + rtInBinary + immediateInBinary

        bitStoredString(self, strInBinary, instruction, arguments)
        return

    # Jump instructions
    if len(machineCode) == 2:
        # Creating a machine code in hexadecimal
        address = args[0]
        machineCode[1] = hex(int(address, 16))

        # Creating a binary bit string
        opcodeInBinary = self.hexToBinary(machineCode[0], 6)
        binaryAddress = self.hexToBinary(machineCode[1], 26)
```

```python
            strInBinary = opcodeInBinary + binaryAddress

            # Storing bit string in memory
            self.bitStoredString(strInBinary, instruction, arguments)
            return
        return

def pseudoInstructionParsing(self, instruction, args):
    # Fetching the Pseudo Instructions and replacing them with corresponding instructions
    instructions = []
    arguments = []

    if instruction == 'move':
        instructions = ['add']
        arguments = [[args[0], args[1], '$zero']]
    if instruction == 'ble':
        instructions = ['slt', 'bne']
        arguments = [[args[0], args[1], args[0]], [args[0], '$zero', args[2]]]
    if instruction == 'bge':
        instructions = ['slt', 'beq']
        arguments = [[args[0], args[0], args[1]], [args[0], '$zero', args[2]]]
    if instruction == 'bgt':
        instructions = ['slt', 'bne']
        arguments = [[args[0], args[0], args[1]], [args[0], '$zero', args[2]]]
    count = 0

    for instr in instructions:
        self.instructionParsing(instr, arguments[count])
        count += 1
```

```python
def convertToMachineCode(self, lines):
    #Converting the assembly code to machine code

    self.currentLoc = self.defaultMemLoc
    for line in lines:
        # Dealing with unnecessary parts in the string
        if '#' in line:     #dealing with comments
            line = line[0:line.find('#')]
            line = line.strip()
        if not len(line):       #dealing with spaces
            continue

        checkWordSize(self)

        # Dealing with labels
        if ':' in line:
            label = line[0:line.find(':')]
            self.labelLoc[label] = str(self.currentLoc)
            line = line[line.find(':') + 1:].strip()
        checkWordSize(self)

        # Parsing each line
        instruction = line[0:line.find(' ')].strip()
        args = line[line.find(' ') + 1:].replace(' ', '').split(',')

        if not instruction:
            continue
        #Converting all the values in the argument to decimal
        count = 0
```

```python
     # Creating function code from the table of instructions
    if instruction in self.pseudoInstructionDefinitions.keys():
        pseudoInstructionParsing(self, instruction, args)
    elif instruction in self.instructionDefinitions.keys():
        self.instructionParsing(instruction, args)
    else:
        print("Error: Undefined instruction " + instruction + " is used!")
        exit()
    machineCodePrinter(self)


def findInstructionSize(self, instruction, args):
    # Calculating the size of the instruction (in bytes)

    if instruction in self.pseudoInstructionDefinitions:
        # Assigning corresponding values for each of the pseudo instructions
        if instruction == 'move':
            return 4
        if instruction == 'la' or instruction == 'sgt' or instruction == 'bge' or instruction == 'bgt' or instruction == 'ble' :
            return 8

    if instruction in self.instructionDefinitions:
        return 4
    else:
        print("Error: An invalid instruction " + instruction + " is detected! ")
        exit()
```

```python
def hexToBinary(self, hexaVal, bits):
    # Converting the hexadecimal value to its corresponding binary value
    # For the negative values, performing Two's Complement

    answer = False
    if '-' in hexaVal:
        answer = True
        hexaVal = hexaVal.replace('-', '')
    strInBinary = '0' * bits
    binaryVal = str(bin(int(hexaVal, 16)))[2:]
    strInBinary = strInBinary[0: bits - len(binaryVal)] + binaryVal + strInBinary[bits:]

    # Two's complement if negative hex value
    if answer:
        stringT = strInBinary[0:strInBinary.rfind('1')]
        stringR = strInBinary[strInBinary.rfind('1'):]
        stringT = stringT.replace('1', 'X')
        stringT = stringT.replace('0', '1')
        stringT = stringT.replace('X', '0')
        strInBinary = stringT + stringR
    return strInBinary


def binaryToHex(self, strInBinary):
    # Converting the values in binary to their hexadecimal equivalence
    strInBinary = '0b' + strInBinary
    strInHex = str(hex(int(strInBinary, 2)))[2:]
    strInHex = strInHex.zfill(2)
    return strInHex
```

```python
def bitStoredString(self, strInBinary, instruction, arguments):
    # Storing bit string into the current memory block
    if self.currentLoc % 4 == 0:
        self.updatedList.append('\n\t\t' + hex(self.currentLoc) + ': \t\t\t0x')
    for i in range(0, len(strInBinary) - 1, 8):
        self.updatedList[-1] += binaryToHex(self, strInBinary[i : i + 8])
        self.currentLoc += 1

def machineCodePrinter(self):
    # Displaying the machine code created or writing it to output.o file

    # Batch operation
    if operation == 'b':
        # If the output.o file is not created yet, then creating it in this step
        file = open("output.o", "w+")
        file.write('Corresponding machine code to the source.s file\n\n')
        file.write('\tThe semantic address locations and their machine instructions in hexadecimal are as below: \n')
        file.write('\n\t\t Semantic Address Location\t\t Machine Equivalent in Hexadecimal\n')

        for output in self.updatedList:
            file.write(output + "\n")
        file.close()

    # Interactive operation
    elif operation == 'i':
        print('\n\n\t Semantic Address Location\t Machine Equivalent in Hexadecimal\n')
        for output in self.updatedList:
            print(output)
```

```python
def checkWordSize(self):
    # Checking the memory location to be sure that it is a multiple of word size by 4
    if self.currentLoc % 4 != 0:
        self.currentLoc += 4 - self.currentLoc % 4


if __name__ == '__main__':
    print(sys.argv[1:])
    main(sys.argv[1:])
```

Conclusion:

This project was an interesting task to perform as we were able to see a practical implementation of what we have been studying throughout the semester. Due to some hiccups, we weren’t able to complete it 100% as we faced errors. We wanted to eliminate the errors and make it work 100% but as the exams dates are close, we are unable to do that unfortunately. All in all, in our point of view, we were able to do a major portion but couldn’t really verify the functionality of the assembler completely as we were unable to make our batch mode part work due to the errors. We were able to learn a lot from this project such as how to implement an assembler using a programming language, understand machine level operation and implementation etc.