ستایش سعیدی ۴۰۱۳۱۸۵۱

پروژه اسمبلر فاز ۱

Overview: This project implements an assembler that translates low-level assembly language instructions into machine code. It takes assembly instructions as input and validates them, converting them into their corresponding machine code representation.

Key Features:

1. **Input Flexibility:** Allows input from both the terminal and files, providing flexibility to the user in entering assembly instructions.
2. **Syntax Checking:** Includes a `check_error()` function to verify the correctness of instructions, ensuring they adhere to the defined syntax. Any syntax errors lead to an immediate program exit.
3. **Assembly to Machine Code Conversion:** Translates validated assembly instructions into machine code instructions.
4. **Output Formatting:** Displays the machine code representation of the provided assembly instructions along with the instructions themselves.
5. **Error Handling:** Handles file input errors (e.g., file not found) and invalid user inputs to maintain program stability.

How does it work?

First to understand the code that I wrote you need to know my logic here we are:

1. First I get input from user either from file or terminal until EOF and split it into a matrix that each element contains instructions every line represented with one array in matrix.

```
18
19  input_type = input("Enter 'T' for terminal input or 'F' for file input: ").lower() # ask the user that how it want to insert the codes
20  instruction2 = []
21  if input_type == 't': # if from terminal take take input from user
22      print("Enter the assembly command: ")
23      while True:
24          try:
25              instruction1 = input()
26              instruction2.append(instruction1)
27          except EOFError:
28              break  # insert code until EOF
29
30  elif input_type == 'f': # if from file
31      file_path = input("Enter the file path: ")
32      # get the path of file from user
33      try :
34          with open(file_path, 'r') as file:
35              for line in file:  # oopoen the file and read the file line to line until the end
36                  instruction1 = line.replace("\n" , "")
37                  instruction2.append(instruction1)
38      except FileNotFoundError:
39          print("File not found.") # if the path is invalid print the error
40          exit
41  else:
42      print("Invalid input type. Please enter 'T' or 'F'.") # if user enter anything beside t and f print this error
43      exit()
44
45
46  instruction2= "\n".join(instruction2).lower() # lower case all the instructions and registers
47  instruction2 = instruction2.replace("," , " ").split("\n") # remove , from between of instructions and every line will split to an array
48  length_instruction = len(instruction2)
49  for i in range (length_instruction):
50      instruction2[i] = instruction2[i].split() # every line split into an array with registers and instructions
51  print(instruction2)
52  check_error() # check if there is an syntax error in code to exit the program
53
```

2. Then I check generally that the code has some validation like having same bits of registers or pop doesn't come with two regs or there is a label in code or not

```
> setti > Documents > assembler.py > ...
def check_error(): # an function to check if commands are correct from some syntax mistake and if not exit the program
    for i in range(length_instruction):
        for j in range(len (instruction2[i])):
            if instruction2[i][0] == "add" or instruction2[i][0] == "sub" or instruction2[i][0] == "and" or instruction2[i][0] == "or" or instruction2[i][0] == "xor"
                if len(instruction2[i])==3 and ((len(instruction2[i][2]) == len(instruction2[i][1])) or len(instruction2[i][2]) == len(instruction2[i][1]) + 2 or len(
                    return # check if it has coreect size of two reg and also have 2 registers
            elif instruction2[i][0] == "push" or  instruction2[i][0] == "pop" or  instruction2[i][0] == "inc" or  instruction2[i][0] == "dec":
                if len(instruction2[i]) == 2:
                    return # check if has 1 reg after these instructions
            elif instruction2[i][0] == "jmp" or instruction2[i][0][-1]== ":":
                if instruction2[i][0] == "jmp" :
                    if len(instruction2[i]) == 2:
                        return # check if has 1 reg after these instructions
            else:
                print("something is wrong!")
                exit() # if not exit from program
```

3. After that I find the labels of the code and store it in an array called “lebels” and the line number of label in a array called “lebels_line” :

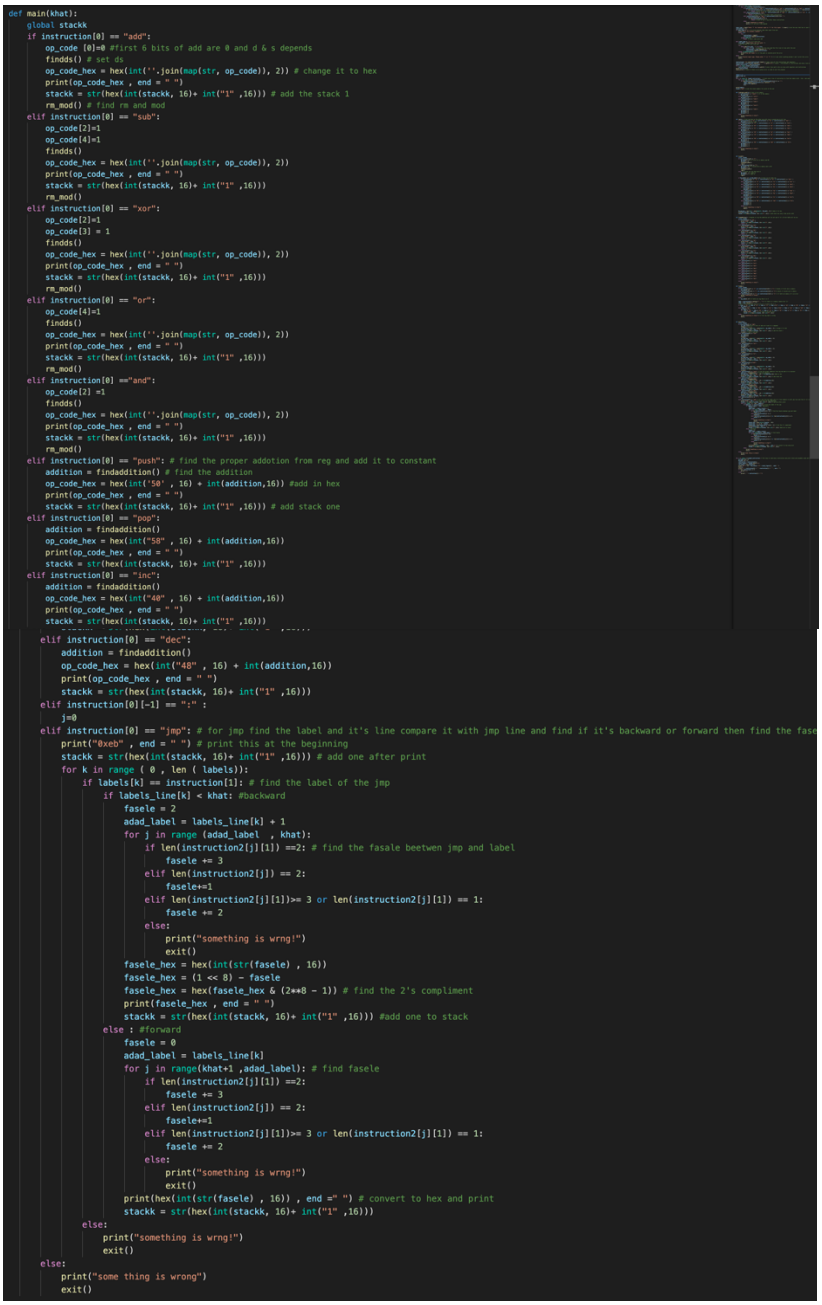
```
labels = []
labels_line =[]
k=0
for i in range (0, length_instruction):   # check every line of instruction to find the labels with : this  and save the label and line that label is in 2 arrays
    for j in range (0, len (instruction2[i])):
        if len(instruction2[i][j]) > 0 and instruction2[i][j][-1] == ":":
            labels.append(instruction2[i][j].replace(":",""))
            labels_line.append(i)
            k+=1

print(labels)
stackk ="0x0" # creat the stack numbers for print at the end
```

4. Now I create a loop that will read every line of the code and process it into a machine code with main function called then print the stack and machine codes and at the end the instruction on that line(I will explain how stack will be calculate) :

```
for i in range ( 0,length_instruction): # the loop to read every instruction and print stack and assemble code and the instruction it self
    op_code= [0] * 8
    Rm_mod= [0] * 8
    instruction = instruction2[i]
    stack_figure =  stackk.split("x")
    len_stack = len(stackk[1])
    print("0x" + (16 - len_stack)* "0" + stack_figure[1] , end=": ")
    main(i)
    print("(" + instruction[0] + " " +instruction[1] + " " , end = "")
    if len( instruction) == 2:
        print(")")
    else:
        print(", " + instruction[2] + ")")
```

5. Let’s go to the main function to explain how machine codes calculated:



```
def main(khat):
    global stackk
    if instruction[0] == "add":
        op_code [0]=0 #first 6 bits of add are 0 and d & s depends
        findds() # set ds
        op_code_hex = hex(int(''.join(map(str, op_code)), 2)) # change it to hex
        print(op_code_hex , end = " ")
        stackk = str(hex(int(stackk, 16)+ int("1" ,16))) # add the stack 1
        rm_mod() # find rm and mod
    elif instruction[0] == "sub":
        op_code[2]=1
        op_code[4]=1
        findds()
        op_code_hex = hex(int(''.join(map(str, op_code)), 2))
        print(op_code_hex , end = " ")
        stackk = str(hex(int(stackk, 16)+ int("1" ,16)))
        rm_mod()
    elif instruction[0] == "xor":
        op_code[2]=1
        op_code[3] = 1
        findds()
        op_code_hex = hex(int(''.join(map(str, op_code)), 2))
        print(op_code_hex , end = " ")
        stackk = str(hex(int(stackk, 16)+ int("1" ,16)))
        rm_mod()
    elif instruction[0] == "or":
        op_code[4]=1
        findds()
        op_code_hex = hex(int(''.join(map(str, op_code)), 2))
        print(op_code_hex , end = " ")
        stackk = str(hex(int(stackk, 16)+ int("1" ,16)))
        rm_mod()
    elif instruction[0] =="and":
        op_code[2] =1
        findds()
        op_code_hex = hex(int(''.join(map(str, op_code)), 2))
        print(op_code_hex , end = " ")
        stackk = str(hex(int(stackk, 16)+ int("1" ,16)))
        rm_mod()
    elif instruction[0] == "push": # find the proper addotion from reg and add it to constant
        addition = findaddition() # find the addition
        op_code_hex = hex(int('50' , 16) + int(addition,16)) #add in hex
        print(op_code_hex , end = " ")
        stackk = str(hex(int(stackk, 16)+ int("1" ,16))) # add stack one
    elif instruction[0] == "pop":
        addition = findaddition()
        op_code_hex = hex(int("58" , 16) + int(addition,16))
        print(op_code_hex , end = " ")
        stackk = str(hex(int(stackk, 16)+ int("1" ,16)))
    elif instruction[0] == "inc":
        addition = findaddition()
        op_code_hex = hex(int("40" , 16) + int(addition,16))
        print(op_code_hex , end = " ")
        stackk = str(hex(int(stackk, 16)+ int("1" ,16)))
    elif instruction[0] == "dec":
        addition = findaddition()
        op_code_hex = hex(int("48" , 16) + int(addition,16))
        print(op_code_hex , end = " ")
        stackk = str(hex(int(stackk, 16)+ int("1" ,16)))
    elif instruction[0][-1] == ":" :
        j=0
    elif instruction[0] == "jmp": # for jmp find the label and it's line compare it with jmp line and find if it's backward or forward then find the fase
        print("0xeb" , end = " ") # print this at the beginning
        stackk = str(hex(int(stackk, 16)+ int("1" ,16))) # add one after print
        for k in range ( 0 , len ( labels)):
            if labels[k] == instruction[1]: # find the label of the jmp
                if labels_line[k] < khat: #backward
                    fasele = 2
                    adad_label = labels_line[k] + 1
                    for j in range (adad_label  , khat):
                        if len(instruction2[j][1]) ==2: # find the fasale beetwen jmp and label
                            fasele += 3
                        elif len(instruction2[j]) == 2:
                            fasele+=1
                        elif len(instruction2[j][1])>= 3 or len(instruction2[j][1]) == 1:
                            fasele += 2
                        else:
                            print("something is wrng!")
                            exit()
                    fasele_hex = hex(int(str(fasele) , 16))
                    fasele_hex = (1 << 8) - fasele
                    fasele_hex = hex(fasele_hex & (2**8 - 1)) # find the 2's compliment
                    print(fasele_hex , end = " ")
                    stackk = str(hex(int(stackk, 16)+ int("1" ,16))) #add one to stack
                else : #forward
                    fasele = 0
                    adad_label = labels_line[k]
                    for j in range(khat+1 ,adad_label): # find fasele
                        if len(instruction2[j][1]) ==2:
                            fasele += 3
                        elif len(instruction2[j]) == 2:
                            fasele+=1
                        elif len(instruction2[j][1])>= 3 or len(instruction2[j][1]) == 1:
                            fasele += 2
                        else:
                            print("something is wrng!")
                            exit()
                    print(hex(int(str(fasele) , 16)) , end =" ") # convert to hex and print
                    stackk = str(hex(int(stackk, 16)+ int("1" ,16)))
                else:
                    print("something is wrng!")
                    exit()
    else:
        print("some thing is wrong")
        exit()
```

Main function comes in 3 parts 1. Add , sub ,.. instructions 2. Pop, push,… instructions 3. Jmp

First. these have 2 machine code first one come with an specific bits like 000000 for add and then d/s

The second machine code comes from regs and sizes that will be found in the related functions
Second . these instructions only have only one machine code and they calculated with the specific constant plus reg rw/rd number in hex
Third. It comes with 2 machine codes first one is always 0xeb and the second one calculate the space between label and jmp instruction and if it is backward 2's compliment it otherwise print it in positive way in hexadecimal
. note that if the instruction is 16 bits then we always have an extra machine code that is 0x66

Functions :

1.findds:

```
def findds():
    global stackk
    if instruction[1][0] == "[" and instruction[2][0] != "[": # checks if first one is memory
        op_code[6] =0
    elif instruction[1][0] != "[" and instruction[2][0] == "[":# checks if second one is memory
        op_code[6] =1
    elif instruction[1][0] == "[" and instruction[2][0] == "[": # if both are memory it's an error
        print("something is wrong")
        exit()
    else :
        op_code[6] =0 # if both are reg then d is 0

    temp = instruction[1].replace("[" , "") # if there is a memory remove the "[]"
    temp = temp.replace("]" , "")
    # set s in op code with reg 1 if it's 8 -> 0 16 or 32 -> 1
    if temp == "al" or temp == "bl" or temp == "cl" or temp =="dl" or temp == "ah" or temp == "bh" or temp== "ch" or temp == "dh":
        op_code[7] = 0
    elif temp == "eax" or temp == "ebx" or temp == "ecx" or temp =="edx" or temp == "esi" or temp == "edi" or temp == "ebp" or temp == "esp" or temp == "ax"
        op_code[7] = 1
        if  temp == "ax" or temp == "bx" or temp == "cx" or temp == "dx" or temp == "si" or temp == "di" or temp == "bp" or temp == "sp":
            print("0x66" , end=" ") # if reg is 16 print this
            stackk = str(hex(int(stackk, 16)+ int("1" ,16)))
    else:
        print("something is wrng!") # if the reg name is wrong
        exit()
```

D: if the first reg is a memory or both regs are not memory the d will be 0 or not the d will be 1(if both are memory then it's a syntax error)
S:if instruction is 8 bits the s is 0 otherwise it's 1

If reg is invalid an error will ocourd

2. rm_mod:

```
def rm_mod():
    global stackk
    if instruction[1][0] == "[" :
        Rm_mod[0] = 0 # if first on is memory mod 00
        Rm_mod[1] =0
        findregrm_mod1(1)
        reg(2)
    elif instruction[2][0] == "[":
        Rm_mod[0] =0 # if second one is memory mod is 00
        Rm_mod[1] = 0
        findregrm_mod1(2)
        reg(1)
    else: # if both are reg then mod 11
        Rm_mod[0]  = 1  #mod 11
        Rm_mod[1] = 1  #mod 11
        reg(2)

    if Rm_mod[0] ==1 and Rm_mod[1] ==1: # find r/m for both reg
        if (instruction[1] == "al" or instruction[1] == "ax" or instruction[1] == "eax" ) :
            Rm_mod[7] =0
        elif (instruction[1] == "cl" or instruction[1] == "cx" or instruction[1] == "ecx" ) :
            Rm_mod[7] =1
        elif (instruction[1] == "dl" or instruction[1] == "dx" or instruction[1] == "edx") :
            Rm_mod[6] =1
        elif (instruction[1] == "bl" or instruction[1] == "bx" or instruction[1] == "ebx") :
            Rm_mod[6] =1
            Rm_mod[7] =1
        elif (instruction[1] == "ah" or instruction[1] == "sp" or instruction[1] == "esp" ) :
            Rm_mod[5] =1
        elif (instruction[1] == "ch" or instruction[1] == "bp" or instruction[1] == "ebp") :
            Rm_mod[5] =1
            Rm_mod[7] =1
        elif (instruction[1] == "dh" or instruction[1] == "si" or instruction[1] == "esi"):
            Rm_mod[5] =1
            Rm_mod[6] =1
        elif (instruction[1] == "bh" or instruction[1] == "edi" or instruction[1] == "di"):
            Rm_mod[5] =1
            Rm_mod[6] =1
            Rm_mod[7] =1
        else:
            print("something is wrng!")
            exit()

    Rm_mod_hex = hex(int(''.join(map(str, Rm_mod)), 2)) # make it to hex
    print(Rm_mod_hex , end = " ")
    stackk = str(hex(int(stackk, 16)+ int("1" ,16))) # add stack one every time prints smth
```

Checks if it is memory then find rm of memory with another function the find the reg with a function that works with memory and reg also if it is memory mod will be 00
If both are registers then mod is 11 and reg will calculate with reg function and rm in this function as shown abow

3.findreegrm_mod1

```
def findregrm_mod1(n):# find r/m for memory
    if instruction[n] == "[eax]": # if it has memory
        Rm_mod[7]=0
    elif instruction[n] == "[ecx]":
        Rm_mod[7] =1
    elif instruction[n] == "[edx]":
        Rm_mod[6] =1
    elif instruction[n] == "[ebx]":
        Rm_mod[6] =1
        Rm_mod[7] =1
    elif instruction[n] == "[esi]":
        Rm_mod[5] =1
        Rm_mod[6] =1
    elif instruction[n] == "[edi]":
        Rm_mod[6] =1
        Rm_mod[5] =1
        Rm_mod[7] =1
    else:
        print("something is wrng!")
        exit()
```

The n will clearfy which one is memory first one or second you have to note that this function is only for memory instructions then will adjust the rm for instruction depend on reg name

4. reg

```
91  def reg(n): # find reg base on the other one with reg or second one in all reg
92      if (instruction[n] == "al" or instruction[n] == "ax" or instruction[n] == "eax" ) :
93          Rm_mod[2] = 0 #reg = 000
94      elif (instruction[n] == "cl" or instruction[n] == "cx" or instruction[n] == "ecx") :
95          Rm_mod[4] = 1
96      elif (instruction[n] == "dl" or instruction[n] == "dx" or instruction[n] == "edx") :
97          Rm_mod[3] = 1
98      elif (instruction[n] == "bl" or instruction[n] == "bx" or instruction[n] == "ebx") :
99          Rm_mod[3] =1
100         Rm_mod[4] = 1
101     elif (instruction[n] == "ah" or instruction[n] == "sp" or instruction[n] == "esp" ) :
102         Rm_mod[2] = 1
103     elif (instruction[n] == "ch" or instruction[n] == "bp" or instruction[n] == "ebp") :
104         Rm_mod[2] = 1
105         Rm_mod[4] =1
106     elif (instruction[n] == "dh" or instruction[n] == "si" or instruction[n] == "esi"):
107         Rm_mod[2]= 1
108         Rm_mod[3] =1
109     elif (instruction[n] == "bh" or instruction[n] == "edi" or instruction[n] == "di"):
110         Rm_mod[2] = 1
111         Rm_mod[3] = 1
112         Rm_mod[4] = 1
113     else:
114         print("something is wrng!")
115         exit()
116
117
```

This function will adjust reg for every instruction in first part of main if there is a memory in instruction it will be adjusted with other one or if both are reg then it will be adjusted with second register depends on reg name

5.findaddition :

```
def findaddition(): # depends on reg the addition will be set and if
    global stackk
    if instruction[1] == "ax" :
        print("0x66" , end=" ")
        stackk = str(hex(int(stackk, 16)+ int("1" ,16)))
        return '0'
    elif instruction[1] == "cx":
        print("0x66" , end=" ")
        stackk = str(hex(int(stackk, 16)+ int("1" ,16)))
        return '1'
    elif instruction[1] == "dx":
        print("0x66" , end=" ")
        stackk = str(hex(int(stackk, 16)+ int("1" ,16)))
        return '2'
    elif instruction[1] == "bx":
        print("0x66" , end=" ")
        stackk = str(hex(int(stackk, 16)+ int("1" ,16)))
        return '3'
    elif instruction[1] == "sp":
        print("0x66" , end=" ")
        stackk = str(hex(int(stackk, 16)+ int("1" ,16)))
        return '4'
    elif instruction[1] == "bp":
        print("0x66" , end=" ")
        stackk = str(hex(int(stackk, 16)+ int("1" ,16)))
        return '5'
    elif instruction[1] == "si":
        print("0x66" , end=" ")
        stackk = str(hex(int(stackk, 16)+ int("1" ,16)))
        return '6'
    elif instruction[1] == "di":
        print("0x66" , end=" ")
        stackk = str(hex(int(stackk, 16)+ int("1" ,16)))
        return '7'
    elif instruction[1] == "eax":
        return '0'
    elif instruction[1] == "ecx":
        return '1'
    elif instruction[1] == "edx":
        return '2'
    elif instruction[1] == "ebx":
        return '3'
    elif instruction[1] == "esp":
        return '4'
    elif instruction[1] == "ebp":
        return '5'
    elif instruction[1] == "esi":
        return '6'
    elif instruction[1] == "edi":
        return '7'
    else:
        print("something is wrng!")
        exit()
```

For instructions like pop that you have to add an constant that depends on the reg this function will find the constant .

Stack:
The way I calculate my stack is when any machine code prints I add one to my stack cause it has a space in stack and every time a line will called the stack will show the current space of stack that has been fully covered

I think it’s all you need to know about my code !