Lokmanya Tilak College of Engineering

Navi Mumbai

DEPARTMENT OF COMPUTER ENGINEERING

Academic Year (2020-2021)

(EVEN SEM) (CBCGS)

Course Name: System Software Lab
Course Code: CSL602
Experiment No.: 1
Title of the Experiment: Design of single pass assembler.
Name of Student: Arin Mashta
Student Roll No.: 142
Year/Semester/Div: TE/Sem-VI/A

Expt-1:

Aim: Design of single pass assembler

Instructions to the students:

1) Consider an hypothetical processor with approx. 20 assembly language instructions. 2) Define a standard MOT for the instructions with only 3 columns namely Mnemonics, Op-code & size. (You may use the table shown below or develop a new)
3) Write a sample ALP using few instructions from the MOT.
4) Do not used label addresses as it leads to forward reference problem which is not tackled in single pass assembler.
5) Write a program using any language (preferably C) to perform the assembly of the program. 6) Execute the program to generate the output of single pass assembly preferably in a tabular form. 7) The output shall display in 3 columns namely Relative address, ALP instruction & machine code.

MOT format:

Mnemonics	Op-code	Size
MOV R	01	1
ADD R	02	1
SUB R	03	1
MUL R	04	1
DIV R	05	1
AND R	06	1
OR R	07	1
ADD data	08	2
SUB data	09	2
MUL data	10	2
DIV data	11	2
AND data	12	2

OR data	13	2
LOAD address	14	3
STORE address	15	3
DCR R	16	1
INC R	17	1
JMP address	18	3
JNZ address	19	3
HALT	20	1

Instruction with data is 2 byte, first byte is the op-code byte & second byte is data byte itself. Instruction with address is 3 byte (as address assumed here is 16 bit), first byte is the op-code byte & second & third bytes are the address bytes.

Input assembly language program:

Consider very simple ALP with only 5 instructions given below:

MOV R

ADD R

SUB 30

STORE 1000

HALT

Output after single pass assembly:

Relative address	Instruction	Machine code
0	MOV R	01
1	ADD R	02
2	SUB 30	09, 30
4	STORE 1000	15, 10, 00
7		20

	HALT	

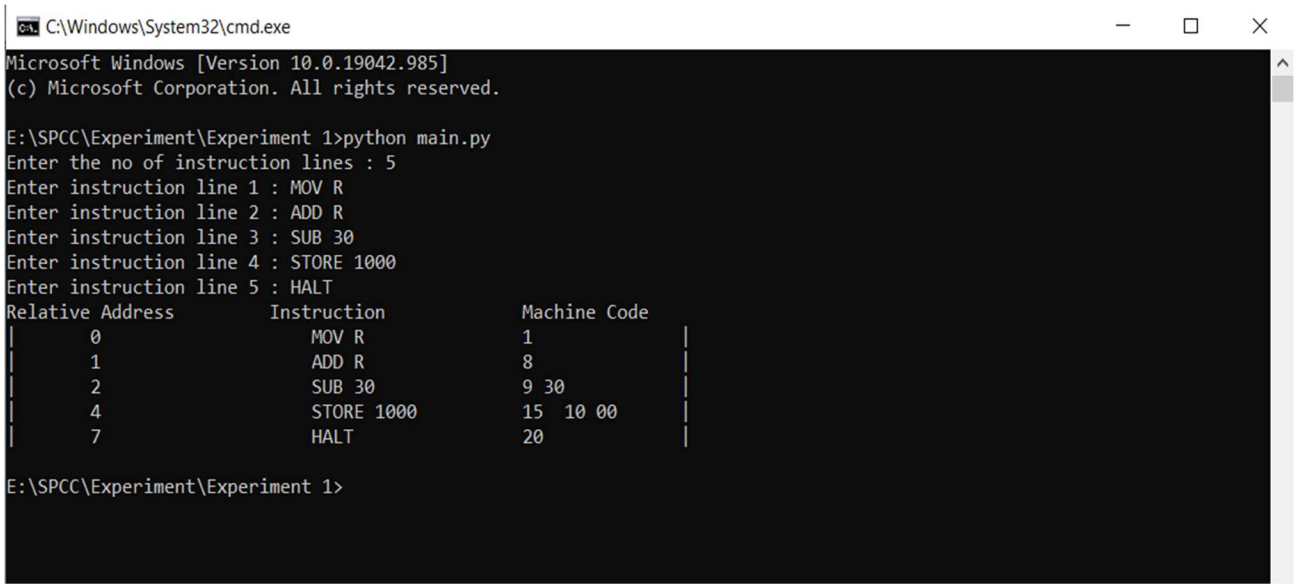
Program:

```
#Arin Mashta TE-A-142
#EXPT NO 1: -DESIGN OF SINGLE PASS ASSEMBLER
from sys import exit

motOpCode = {
    "MOV"   : 1,
    "A"     : 2,
    "S"     : 3,
    "M"     : 4,
    "D"     : 5,
    "AN"    : 6,
    "O"     : 7,
    "ADD"   : 8,
    "SUB"   : 9,
    "MUL"   : 10,
    "DIV"   : 11,
    "AND"   : 12,
    "OR"    : 13,
    "LOAD"  : 14,
    "STORE" : 15,
    "DCR"   : 16,
    "INC"   : 17,
    "JMP"   : 18,
    "JNZ"   : 19,
    "HALT"  : 20
}

motSize = {
    "MOV"   : 1,
    "A"     : 1,
    "S"     : 1,
    "M"     : 1,
    "D"     : 1,
    "AN"    : 1,
    "O"     : 1,
    "ADD"   : 1,
```

```
    "SUB"   : 2,
    "MUL"   : 2,
    "DIV"   : 2,
    "AND"   : 2,
    "OR "   : 2,
    "LOAD"  : 3,
    "STORE" : 3,
    "DCR"   : 1,
    "INC"   : 1,
    "JMP"   : 3,
    "JNZ"   : 3,
    "HALT"  : 1
}

l = []
relativeAddress = []
machineCode = []
RA = 0
current = 0
count = 0
n=int(input("Enter the no of instruction lines : "))
for i in range(n):
     instructions = input("Enter instruction line {} : ".format(i + 1))
     l.append(instructions)
# l = ['MOV R','ADD R','SUB 30','STORE 1000','HALT 20']
l = [x.upper() for x in l]      # Converting all the instructions to upper case

for i in range(n):
    x = l[i]
    if " " in x:
        s1 = ''.join(x)
        a, b = s1.split()
        if a in motOpCode:      # Checking if Mnemonics is present in MOT or not
            value = motOpCode.get(a)
            size = motSize.get(a)
            previous = size
            RA += current
            current = previous
            relativeAddress.append(RA)
            if b.isalpha() is True:
                machineCode.append(str(value))
            else:
                temp = list(b)
                for i in range(len(temp)):
                    if count == 2:
                        temp.insert(i,' ')
```

```
                        count = 0
                    else:
                        count = count + 1
                s =''.join(temp)
                machineCode.append(str(value) + " " + s)
        else:
            print("Instruction is not in Op Code Table.")
            exit(0) # EXIT if Mnemonics is not in MOT
    else:
        if x in motOpCode:
            value = motOpCode.get(x)
            size = motSize.get(x)
            previous = size
            RA += current
            current = previous
            relativeAddress.append(RA)
            machineCode.append(value)
        else:
            print("Instruction is not in Op Code Table.")
            exit(0)


print("{:<20}\t {:<15}\t {:<10}".format('Relative Address','Instruction','Machine C
ode'))
for i in range(n):
    print("|\t{:<20} {:<15}\t {:<15}|".format(relativeAddress[i], l[i], machineCode
[i]))
```

OUTPUT:



Lokmanya Tilak College of Engineering

Navi Mumbai

DEPARTMENT OF COMPUTER ENGINEERING

Academic Year (2020-2021)

(EVEN SEM) (CBCGS)

Course Name: System Software Lab
Course Code: CSL602
Experiment No.: 2
Title of Experiment: Design of Two pass assembler
Name of Student: Arin Mashta
Student Roll No.: 142
Year/Semester/Div: TE/Sem-VI/A

Expt-2:

Aim: Design of two pass assembler.

Instructions to the students:

1)Consider an hypothetical processor with approx. 20 assembly language instructions.
2) Define a standard MOT similar to that of used in Expt-1.
3) Write a ALP using few instructions from the MOT.
4) Use at least one address label in the program so it needs 2 passes for the assembly process.
5) Write a program using C to perform the assembly of the program.
6) Execute the program to generate the output of pass-1 & pass-2
7) In pass-1 output, develop Symbol Table (ST) & pass-1 output without reference to symbol.
8) In pass-2 output, generate the final machine code.
9) Output shall display in 3 columns namely Relative address, ALP instruction & machine code.

MOT format:

Mnemonics	Op-code	Size
MOV R	01	1
ADD R	02	1
SUB R	03	1
MUL R	04	1
DIV R	05	1
AND R	06	1
OR R	07	1
ADD data	08	2
SUB data	09	2
MUL data	10	2
DIV data	11	2
AND data	12	2
OR data	13	2

LOADaddress	14	3
STORE address	15	3
DCR R	16	1
INC R	17	1
JMP address	18	3
JNZ address	19	3
HALT	20	1

Instruction with data is 2 byte, first byte is the op-code byte & second byte is data byte itself. Instruction with address is 3 byte (as address assumed here is 16 bit), first byte is the op-code byte & second & third bytes are the address bytes.

Input assembly language program:

Consider an ALP with following instructions:

```
        MOV R
Next: ADD R
        DCR R
        JNZ Next
        STORE 2000
        HALT
```

Output after Pass-1:

Symbol Table (ST):

Symbol	Value (Address)
Next	0001

Pass-1 machine code output without reference of the symbolic address:

Relative address	Instruction	Machine code
0	MOV R	01
1	ADD R	02
2	DCR R	16
3	JNZ Next	19, -- , --
6	STORE 2000	15, 20, 00
9	HALT	20

Pass-2 output: Machine code output

Relative address	Instruction	Machine code
0	MOV R	01
1	ADD R	02
2	DCR R	16
3	JNZ Next	19, 00 , 01
6	STORE 2000	15, 20, 00
9	HALT	20

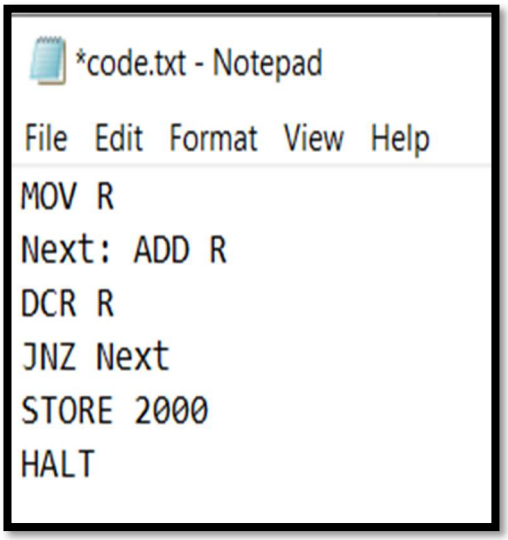
Program:

```
# Arin Mashta TE-A-142
# EXPT NO 2- DESIGN OF TWO PASS ASSEMBLER

import sys

data = {
"MOV": [1, 1],
"ADD": [2, 1, 8, 2],
"SUB": [3, 1, 9, 2],
"MUL": [4, 1, 10, 2],
"DIV": [5, 1, 11, 2],
"AND": [6, 1, 12, 2],
"OR": [7, 1, 13, 2],
"LOAD": [14, 3, 'a'],
"STORE": [15, 3, 'a'],
"DCR": [16, 1],
"INC": [17, 1],
"JMP": [18, 3, 'a'],
```

```
"JNZ": [19, 3, 'a'],
"HALT": [20, 1],
}


def display(sym):
    key =[]
    value=[]
    for x,y in sym.items():
        key.append(x)
        value.append(y)
    print("Symbol Table")
    print(" ____________________________")
    print("| symbol  | Value(Address) |")
    print("|_________|________________|")
    print(f"|  {key[0]}   |        {value[0]}       |")
    print("|_________|________________|")



def add_make(num):
    d = "{:04d}".format(num)
    return d[:2] + " " + d[2:]


if len(sys.argv) == 1:
    print("Error")
    exit(1)

with open(sys.argv[1], "r") as f:
    code = f.readlines()

rel_addr = [0]
mac_code = []
sym = {}
for i in code:
    line = i.strip().split(" ")
    if line[0].endswith(":"):
        sym[line[0][:-1]] = rel_addr[-1]
        line = line[1:]
    if line[0].upper() not in data:
        print("Error in parsing the instruction: ")
        print("".join(line))
        exit(1)
    else:
        ins = data[line[0].upper()]
```

```
        if len(line) == 0:
            continue
        elif len(line) == 1:
            rel_addr.append(rel_addr[-1] + ins[-1])
            mac_code.append("{:02d}".format(ins[0]))
            continue
        if len(ins) == 2:
            rel_addr.append(rel_addr[-1] + ins[-1])
            if len(line[1]) == 1:
                mac_code.append("{:02d}".format(ins[0]))
            else:
                mac_code.append("{:02d} {}".format(ins[0], ' '.join([line[1][j:j+2]
 for j in range(0, len(line[1]), 2)])))
        elif len(ins) == 3:
            rel_addr.append(rel_addr[-1] + ins[1])
            mac_code.append(["{:02d}".format(ins[0]), line[-1]])
        else:
            if len(line[1]) == 1:
                rel_addr.append(rel_addr[-1] + ins[1])
                mac_code.append("{:02d}".format(ins[0]))
            else:
                rel_addr.append(rel_addr[-1] + ins[3])
                mac_code.append("{:02d} {}".format(ins[0], ' '.join([line[1][j:j+2]
 for j in range(0, len(line[1]), 2)])))

display(sym)
for i in range(len(mac_code)):
    if type(mac_code[i]) == list:
        if mac_code[i][-1] not in sym:
            try:
                d = int(mac_code[i][-1])
                mac_code[i] = mac_code[i][0] + " " + add_make(d)
            except:
                print("Broken Link at: {}".format(code[i]))
                exit(-1)
        else:
            mac_code[i] = mac_code[i][0] + " " + add_make(sym[mac_code[i][-1]])

code[1] = "ADD R"
temp = mac_code[3]
mac_code[3] = "19,--,--"
print()
print("Pass-1 Machine code output without reference of the symbolic address")
print(" _____________________________________________")
print("|Relative address |    Instruction   | Machine Code  |")
print("|_________________|__________________|_______________|")
```

```
for i in range(len(code)):
    print("|  {:02d}             |  {:<14} | {:<14}|".format(rel_addr[i], code[i].s
trip(), mac_code[i].strip()))
print("|_________________|_________________|_____________|")

mac_code[3]=temp

print()
print("Pass-2 Output: Machine code output")
print(" ____________________________________________________")
print("|Relative address |    Instruction   | Machine Code  |")
print("|_________________|_________________|_____________|")
for i in range(len(code)):
    print("|  {:02d}             |  {:<14} | {:<14}|".format(rel_addr[i], code[i].s
trip(), mac_code[i].strip()))
print("|_________________|_________________|_____________|")
```



OUTPUT:

C:\Windows\System32\cmd.exe

```
 _____________________
| symbol  | Value(Address) |
|_________|________________|
|  Next   |        1       |
|_________|________________|

Pass-1 Machine code output without reference of the symbolic address
 ___________________________________________________
|Relative address |    Instruction   | Machine Code  |
|_________________|__________________|_______________|
|  00             |  MOV R           | 01            |
|  01             |  ADD R           | 02            |
|  02             |  DCR R           | 16            |
|  03             |  JNZ Next        | 19,--,--      |
|  06             |  STORE 2000      | 15 20 00      |
|  09             |  HALT            | 20            |
|_________________|__________________|_______________|

Pass-2 Output: Machine code output
 ___________________________________________________
|Relative address |    Instruction   | Machine Code  |
|_________________|__________________|_______________|
|  00             |  MOV R           | 01            |
|  01             |  ADD R           | 02            |
|  02             |  DCR R           | 16            |
|  03             |  JNZ Next        | 19 00 01      |
|  06             |  STORE 2000      | 15 20 00      |
|  09             |  HALT            | 20            |
|_________________|__________________|_______________|

C:\Users\admin\PycharmProjects\exp 2>
```

Lokmanya Tilak College of Engineering

Navi Mumbai

DEPARTMENT OF COMPUTER ENGINEERING

Academic Year (2020-2021)

(EVEN SEM) (CBCGS)

Course Name: System Software Lab
Course Code: CSL602
Experiment No.: 3
Title of Experiment: Defining a macro without argument, expanding macro calls & generating expanded source code.
Name of Student: Arin Mashta
Student Roll No.: 142
Year/Semester/Div: TE/Sem-VI/A

Expt-3:

Aim: Defining a macro without argument, expanding macro calls & generating expanded source code.

Instructions to the students:

1) Write a ALP using few instructions & Macro calls

2) Define domain/body of the Macro (3-5 instructions in a simple way)

3) Write a program which will replace each Macro call with its domain & performs expansion of the Macro.

4) Show the input source code with Macro call & output the expanded source code

5) Also output the following statistics:

- Number of instructions in input source code (excluding Macro calls)
- Number of Macro calls
- Number of instructions defined in the Macro call
- Total number of instructions in the expanded source code.

Input 1: Input Source code with Macro calls

```
MOV R
RAHUL
DCR R
AND R
RAHUL
MUL 88
HALT
```

Input 2: Macro definition

```
MACRO
RAHUL
 ADD 30
 SUB 25
```

OR R

MEND

Output source code after Macro expansion:

MOV R

ADD 30

SUB 25

OR R

DCR R

AND R

ADD 30

SUB 25

OR R

MUL 88

HALT

Statistical output:

Number of instructions in input source code (excluding Macro calls) = 5

Number of Macro calls = 2

Number of instructions defined in the Macro call = 3

Total number of instructions in the expanded source code = 11

Program:

```
#Arin Mashta TE-A-142
#EXPT NO 3:- Defining a macro without argument, expanding macro calls & generating
expanded source code.
code        = open("E:\SPCC\Experiment\Experiment3\code.txt", "r")
macro       = open("E:\SPCC\Experiment\Experiment3\macro.txt","r")

macrol      = []
codel       = []
macrocount  = 0


for x in macro:
    macrol.append(x.split(' \n')[0].upper())

macroname = macrol[macrol.index('MACRO')+1]
macrol.remove(macroname)
macrol.remove('MACRO')
macrol.remove('MEND')



for x in code:
    codel.append(x.split(' \n')[0])

codelen     = len(codel)

for i in range(len(codel)):
    if codel[i] == macroname:
        codel[i : i+len(macrol)-2] = tuple(macrol)
        macrocount += 1

for i in codel:
    print(i)

print("\n\nStatistical Output")
print("Number of instructions in input source code (excluding Macro calls) = {}".fo
rmat(codelen - macrocount))
print("Number of Macro calls = {}".format(macrocount))
print("Number of instructions defined in the Macro call = {}".format(len(macrol)))
print("Total number of instructions in the expanded source code = {}".format(len(co
del)))

macro.close()
code.close()
```

macro.txt - Notepad

File Edit Format View Help

```
MACRO
RAHUL
ADD 30
SUB 25
OR R
MEND
```

*code.txt - Notepad

File Edit Format View Help

```
MOV R
RAHUL
DCR R
AND R
RAHUL
MUL 88
HALT
```

OUTPUT:

C:\Windows\System32\cmd.exe

```
Microsoft Windows [Version 10.0.19042.985]
(c) Microsoft Corporation. All rights reserved.

E:\SPCC\Experiment\Experiment3>python exp3.py
MOV R
ADD 30
SUB 25
OR R
DCR R
AND R
ADD 30
SUB 25
OR R
MUL 88
HALT


Statistical Output
Number of instructions in input source code (excluding Macro calls) = 5
Number of Macro calls = 2
Number of instructions defined in the Macro call = 3
Total number of instructions in the expanded source code = 11

E:\SPCC\Experiment\Experiment3>
```

Lokmanya Tilak College of Engineering

Navi Mumbai

DEPARTMENT OF COMPUTER ENGINEERING

Academic Year (2020-2021)

Second Half 2020 (EVEN SEM) (CBCGS)

Course Name: System Software Lab
Course Code: CSL602
Experiment No.: 4
Title of Experiment: Defining a macro of one argument, expansion of macro & generating expanded source code
Name of Student: Arin Mashta
Student Roll No.: 142
Year/Semester/Div: TE/Sem-VI/A

Expt-4:

Aim: Defining a macro with one argument, expansion of macro & generating expanded source code.

Instructions to the students:

1) Write an ALP using few instructions & Macro calls with only one argument.

2) Define domain/body of the Macro (3-5 instructions).

3) Write a program which will replace each Macro call with its domain & performs expansion of the Macro.

4) Show the input source code with Macro call & output the expanded source code.

5) Also output the following statistics:
- Number of instructions in input source code (excluding Macro calls)
- Number of Macro calls
- Number of instructions defined in the Macro call
- Actual argument during each Macro call
- Total number of instructions in the expanded source code.

Input 1: Input Source code with Macro calls

```
MOV R
RAHUL 30
DCR R
AND R
RAHUL 55
MUL 88
HALT
```

Input 2: Macro definition

```
MACRO
RAHUL &ARG
 ADD & ARG
```

SUB &ARG

OR &ARG

MEND

Output source code after Macro expansion:

MOV R

ADD 30

SUB 30

OR 30

DCR R

AND R

ADD 55

SUB 55

OR 55

MUL 88

HALT

Statistical output:

Number of instructions in input source code (excluding Macro calls) = 5

Number of Macro calls = 2

Number of instructions defined in the Macro call = 3

Actual argument during first Macro call “RAHUL” = 30

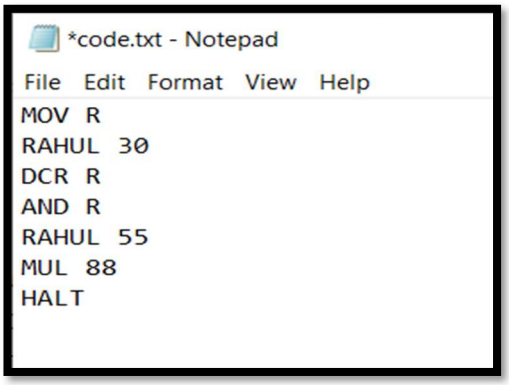
Actual argument during second Macro call “RAHUL” = 55

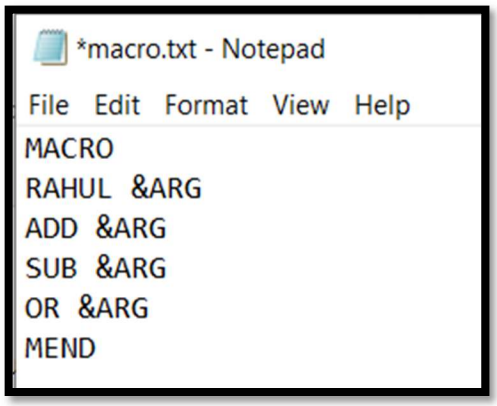
Total number of instructions in the expanded source code = 11

Program:

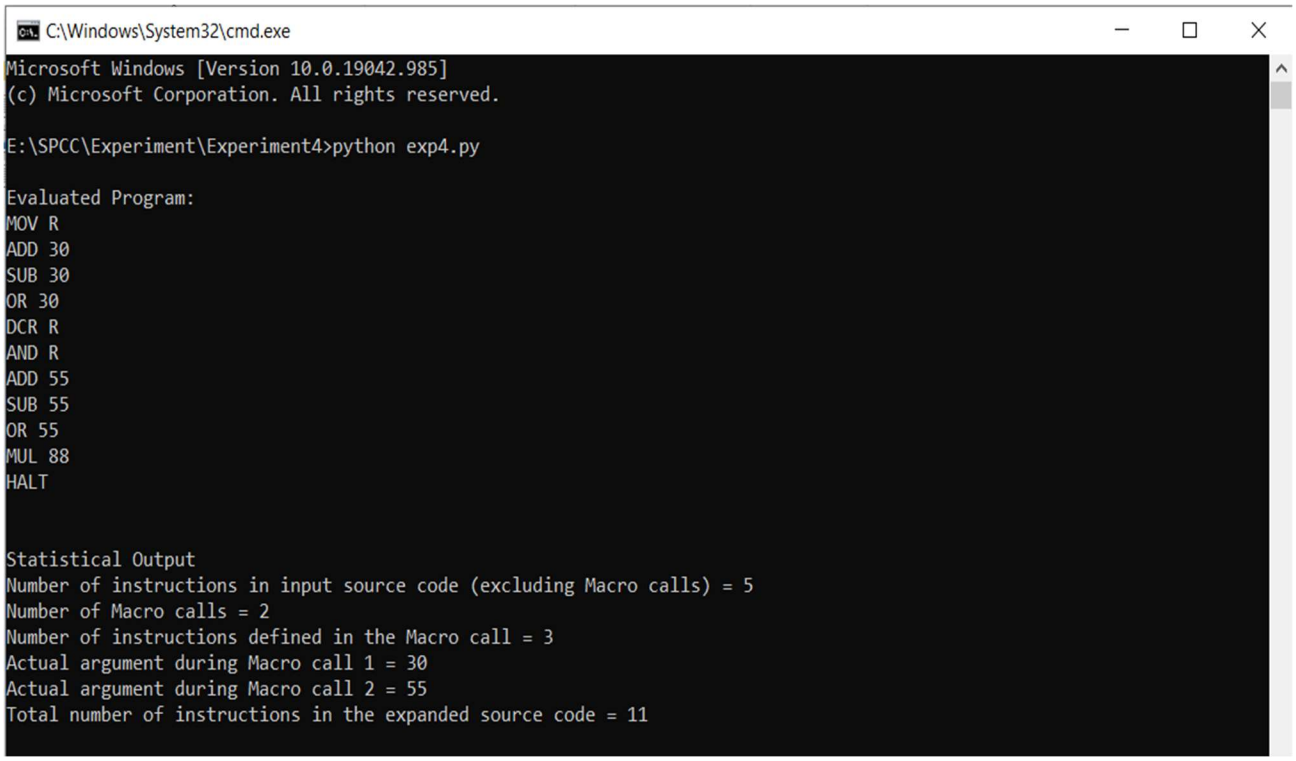
```
#Arin Mashta TE-A-142
#EXPT NO 4:- Defining a macro with one argument, expansion of macro & generating ex
panded source code.

code        = open("code.txt", "r")
macro       = open("macro.txt","r")

macrol      = []
codel       = []
argl        = []
macrocount  = 0


for x in macro:
    macrol.append(x.removesuffix(' \n').upper())

macroname = macrol[macrol.index('MACRO')+1]
macrol.remove(macroname)
macrol.remove('MACRO')
macrol.remove('MEND')
macroname = macroname.split()[0]

def callMacro(arg, macrol):
    temp = []
    for i in macrol:
        temp.append(i.replace('&ARG', arg))
    return tuple(temp)

for x in code:
    codel.append(x.removesuffix(' \n'))

codelen     = len(codel)

for i in range(len(codel)):
    a = codel[i].split()
    if a[0] == macroname:
        codel[i : i+len(macrol)-2] = callMacro(a[1],macrol)
        argl.append(a[1])
        macrocount += 1

print("\nEvaluated Program: ")
for i in codel:
    print(i)

print("\n\nStatistical Output")
```

```
print("Number of instructions in input source code (excluding Macro calls) = {}".fo
rmat(codelen - macrocount))
print("Number of Macro calls = {}".format(macrocount))
print("Number of instructions defined in the Macro call = {}".format(len(macrol)))
for i,data in enumerate(argl):
    print("Actual argument during Macro call {} = {}".format(i+1,data))
print("Total number of instructions in the expanded source code = {}".format(len(co
del)))

macro.close()
code.close()
```





OUTPUT:



Lokmanya Tilak College of Engineering

Navi Mumbai

DEPARTMENT OF COMPUTER ENGINEERING

Academic Year (2020-2021)

(EVEN SEM) (CBCGS)

Course Name: System Software Lab
Course Code: CSL602
Experiment No.: 5
Title of Experiment: Design a macro with multiple arguments, expansion of macro calls & generating expanded source code.
Name of Student: Arin Mashta
Student Roll No.: 142
Year/Semester/Div: TE/Sem-VI/A

Expt-5:

Aim: Defining a macro with multiple arguments, expansion of macro calls & generating expanded source code.

Instructions to the students:

1) Write a ALP using few instructions & Macro calls with multiple arguments.
2) Define domain/body of the Macro with arguments
3) Write a program which will replace each Macro call with its domain along with proper arguments & performs expansion of the Macro.
4) Show the input source code with Macro call & output the expanded source code
5) Also output the following statistics:
   - Number of instructions in input source code (excluding Macro calls)
   - Number of Macro calls
   - Number of instructions defined in the Macro call
   - Actual argument during each Macro call
   - Total number of instructions in the expanded source code.

Input 1: Input Source code with Macro calls

```
MOV R
RAHUL 30, 40, 50
DCR R
AND R
RAHUL 33, 44, 55
MUL 88
HALT
```

Input 2: Macro definition

```
MACRO
RAHUL &ARG1, &ARG2, &ARG3
 ADD & ARG1
 SUB &ARG2
 OR &ARG3
MEND
```

Output source code after Macro expansion:

MOV R

ADD 30

SUB 40

OR 50

DCR R

AND R

ADD 33

SUB 44

OR 55

MUL 88

HALT

Statistical output:

Number of instructions in input source code (excluding Macro calls) = 5

Number of Macro calls = 2

Number of instructions defined in the Macro call = 3

Actual argument during first Macro call “RAHUL” = 30, 40, 50

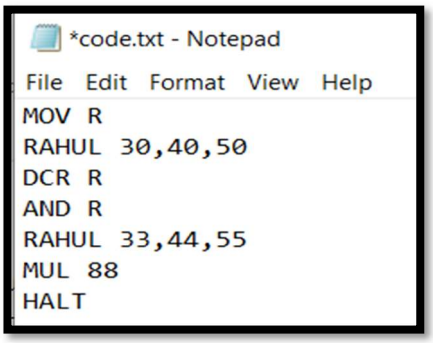
Actual argument during second Macro call “RAHUL” = 33, 44, 55

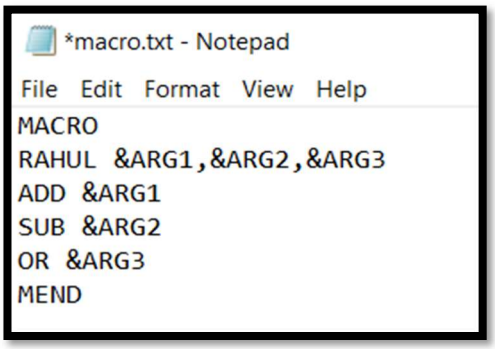
Total number of instructions in the expanded source code = 11

Program:

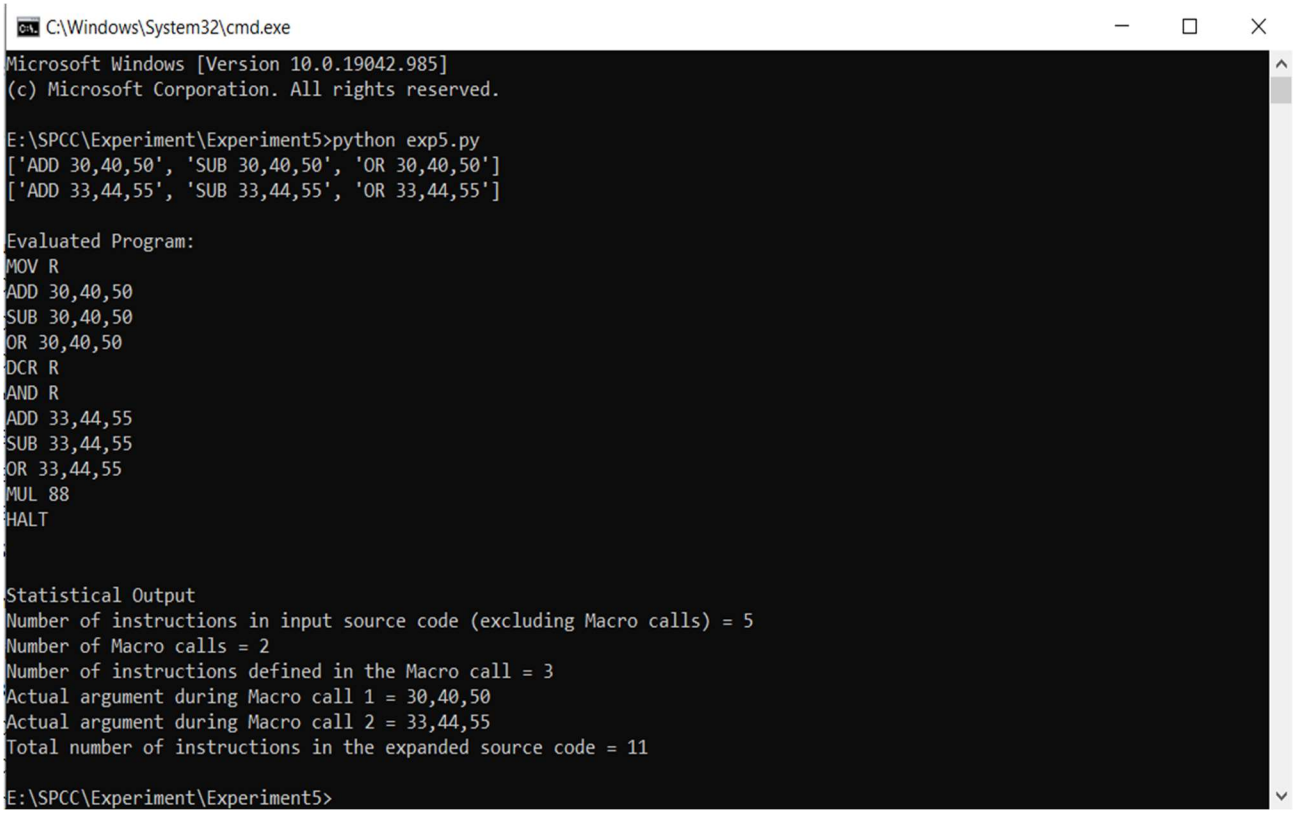
```
#Arin Mashta TE-A-142
#EXPT NO 5:-
Defining a macro with multiple arguments, expansion of macro calls & generating exp
anded source code.
code        = open("code.txt", "r")
macro       = open("macro.txt","r")

macrol      = []
codel       = []
argl        = []
macrocount  = 0

for x in macro:
    macrol.append(x.removesuffix(' \n').upper())

macroname = macrol[macrol.index('MACRO')+1]
macrol.remove(macroname)
macrol.remove('MACRO')
macrol.remove('MEND')
macroname = macroname.split()[0]

def callMacro(arg, macrol):
    temp = []
    for i in macrol:
        a = i.split()
        a[1] = arg
        temp.append(' '.join(a))
    print(temp)
    return tuple(temp)

for x in code:
    codel.append(x.removesuffix(' \n'))

codelen = len(codel)

for i in range(len(codel)):
    a = codel[i].split()
    if a[0] == macroname:
        codel[i : i+len(macrol)-2] = callMacro(a[1],macrol)
        argl.append(a[1])
        macrocount += 1
print("\nEvaluated Program: ")
for i in codel:
    print(i)
print("\n\nStatistical Output")
```

```
print("Number of instructions in input source code (excluding Macro calls) = {}".fo
rmat(codelen - macrocount))
print("Number of Macro calls = {}".format(macrocount))
print("Number of instructions defined in the Macro call = {}".format(len(macrol)))
for i,data in enumerate(argl):
    print("Actual argument during Macro call {} = {}".format(i+1,data))
print("Total number of instructions in the expanded source code = {}".format(len(co
del)))

macro.close()
code.close()
```





OUTPUT:



Lokmanya Tilak College of Engineering

Navi Mumbai

DEPARTMENT OF COMPUTER ENGINEERING

Academic Year (2020-2021)

(EVEN SEM) (CBCGS)

Course Name: System Software Lab
Course Code: CSL602
Experiment No.: 6
Title of Experiment: Defining a macro with more positional arguments & label argument, expansion of macro & generating expanded source code.
Name of Student: Arin Mashta
Student Roll No.: 142
Year/Semester/Div: TE/Sem-VI/A

Expt-6:

Aim: Defining a macro with more positional arguments & label argument, expansion of macro & generating expanded source code.

Instructions to the students:

1) Write a ALP using few instructions & Macro calls with a label argument & multiple arguments.
2) Define domain/body of the Macro with arguments
3) Write a program which will replace each Macro call with its domain along with proper arguments & performs expansion of the Macro.
4) Show the input source code with Macro call & output the expanded source code
5) Also output the following statistics:
   - Number of instructions in input source code (excluding Macro calls)
   - Number of Macro calls
   - Number of instructions defined in the Macro call
   - Actual argument during each Macro call
   - Label argument during each Macro call
   - Total number of instructions in the expanded source code.

Input 1: Input Source code with Macro calls

```
        MOV R
STAR: RAHUL 30, 40, 50
        DCR R
        AND R
NEXT: RAHUL 33, 44, 55
        MUL 88
        HALT
```

Input 2: Macro definition

```
        MACRO
        &LAB RAHUL &ARG1, &ARG2, &ARG3
        &LAB ADD &ARG1
        SUB &ARG2
        OR &ARG3
```

MEND

Output source code after Macro expansion:

```
        MOV R
STAR: ADD 30
        SUB 40
        OR 50
        DCR R
        AND R
NEXT: ADD 33
        SUB 44
        OR 55
        MUL 88
        HALT
```

Statistical output:

Number of instructions in input source code (excluding Macro calls) = 5

Number of Macro calls = 2

Number of instructions defined in the Macro call = 3

Actual argument during first Macro call "RAHUL" = 30, 40, 50

Actual Label argument during first Macro call = STAR

Actual argument during second Macro call "RAHUL" = 33, 44, 55

Actual Label argument during second Macro call = NEXT

Total number of instructions in the expanded source code = 11

Program:

```
#Arin Mashta TE-A-142
#EXPT NO 6:Defining a macro with more positional arguments & label argument, expans
ion of macro &  generating expanded source code.

import sys
macro_file = sys.argv[1]
program_file = sys.argv[2]

macro_cache = {}

with open(macro_file) as f:
    data = [i.strip() for i in f.readlines()]

macro_state = False
for i in range(len(data)):

    if data[i] == 'MACRO':
        i = i + 1
        label = data[i].split(" ")[0]
        mname = data[i].split(" ")[1]
        pholders = ''.join(data[i].split(" ")[2:]).split(',')
        pholder = {}
        count = 0
        for j in pholders:
            pholder[j] = "{" + f"{count}" + "}"
            count += 1
        # print(pholders)
        macro_cache[mname] = []
        i += 1
        while data[i] != 'MEND':
            for j in pholders:
                data[i] = data[i].replace(j, pholder[j], -1)
            data[i] = data[i].replace(label, "{"+f"{count}"+"}")
                # print(j, data[i])

            macro_cache[mname].append(data[i])
            i += 1
        i += 1
macro_calls = 0
src_inst = 0
macro_calls_inst = 0
total = 0
# print(macro_cache)
print()
with open(program_file) as f:
```

```
    data = [i.strip() for i in f.readlines()]
    for i in data:
        if len(i.split(" ")) > 1 and i.split(" ")[1] in macro_cache:
            macro_calls += 1
            macro_calls_inst = len(macro_cache[i.split(" ")[1]])
            print()
            for j in macro_cache[i.split(" ")[1]]:
                print(j.format(*''.join(i.split(" ")[2:]).split(","), i.split(" ")[
0]))
                total += 1
            print()
        else:
            src_inst += 1
            print(i)
            total += 1
print()
print(f"No. of instructions in input source code (excluding Macro calls): {src_inst
}")
print(f"No. of macro calls: {macro_calls}")
print(f"Actual argument during first Macro call {}: {macro_calls_inst}")
print(f"Actual label argument during first Macro call: {macro_calls}.format(j)")
print(f"Actual argument during second Macro call {}: {macro_calls}")
print(f"Actual Label argument during second Macro call
{}: {macro_calls}.format(f)")
print(f"Total instructions: {total}")
```

marco-def.in

Exp6 > marco-def.in

```
1  MACRO
2  &LAB RAHUL &ARG1, &ARG2, &ARG3
3  &LAB ADD &ARG1
4  SUB &ARG2
5  OR &ARG3
6  MEND
7
8
```

program.in

Exp6 > program.in

```
1  MOV R
2  STAR: RAHUL 30, 40, 50
3  DCR R
4  AND R
5  NEXT: RAHUL 33, 44, 55
6  MUL 88
7  HALT
8
```

OUTPUT:

```
PS D:\experiments\SPCC\SPCC4\Exp6> py main.py marco-def.in program.in

MOV R

STAR: ADD 30
SUB 40
OR 50

DCR R
AND R

NEXT: ADD 33
SUB 44
OR 55

MUL 88
HALT

No. of instructions in input source code (excluding Macro calls): 5
No. of macro calls: 2
Actual argument during first Macro call Rahul=30, 40, 50
Actual label argument during first Macro call= STAR
Actual argument during second Macro call Rahul=33, 44, 55
Actual Label argument during second Macro call Rahul=Next
Total instructions: 11
PS D:\experiments\SPCC\SPCC4\Exp6>
```

Lokmanya Tilak College of Engineering

Navi Mumbai

DEPARTMENT OF COMPUTER ENGINEERING

Academic Year (2020-2021)

(EVEN SEM) (CBCGS)

Course Name: System Software Lab
Course Code: CSL602
Experiment No.: 7
Title of Experiment: Implementation of 2-levels nested macro & its expansion
Name of Student: Arin Mashta
Student Roll No.: 142
Year/Semester/Div: TE/Sem-VI/A

Expt-7:

Aim: Implementation of 2-levels nested macro & its expansion.

Instructions to the students:

1) Write a ALP using few instructions & First level macro call with no arguments.
2) Define domain/body of the first level Macro with some other macros defined with/without arguments.
3) Define domains of each of the macros called in the first level macro.
4) Show the expansion of the Macro at the first level.
5) Also show the expansion of each of the macro at the second level.
6) Output the final expanded source code.
7) Also output the following statistics:
   - Number of instructions in input source code (excluding Macro calls)
   - Number of Macro calls at first level
   - Number of instructions & other macro calls defined in the first level Macro call - Total number of instructions in the final expanded source code.

Input 1: Input Source code with First level macro calls

```
MOV R
AND R
RAHUL
MUL 88
HALT
```

Input 2: First level macro definition "RAHUL"

```
MACRO
RAHUL
 SUB R
 TILAK 77
 MUL R
 TILAK 99
MEND
```

Input 3: Second level macro definition "TILAK"

```
MACRO
TILAK &ARG
 ADD &ARG
 MUL &ARG
MEND
```

Output code after first level macro expansion"RAHUL":

```
MOV R
AND R
SUB R
TILAK 77
 MUL R
 TILAK 99
MUL 88
HALT
```

Final Expanded source code (after second level macro expansion"TILAK"):

```
MOV R
AND R
SUB R
ADD 77
MUL 77
MUL R
ADD 99
MUL 99
MUL 88
```

HALT

Statistical output:

Number of instructions in input source code (excluding Macro calls) = 4

Number of Macro calls at first level = 1

Number of instructions & other macro calls defined in the first level Macro

call = 2, 2

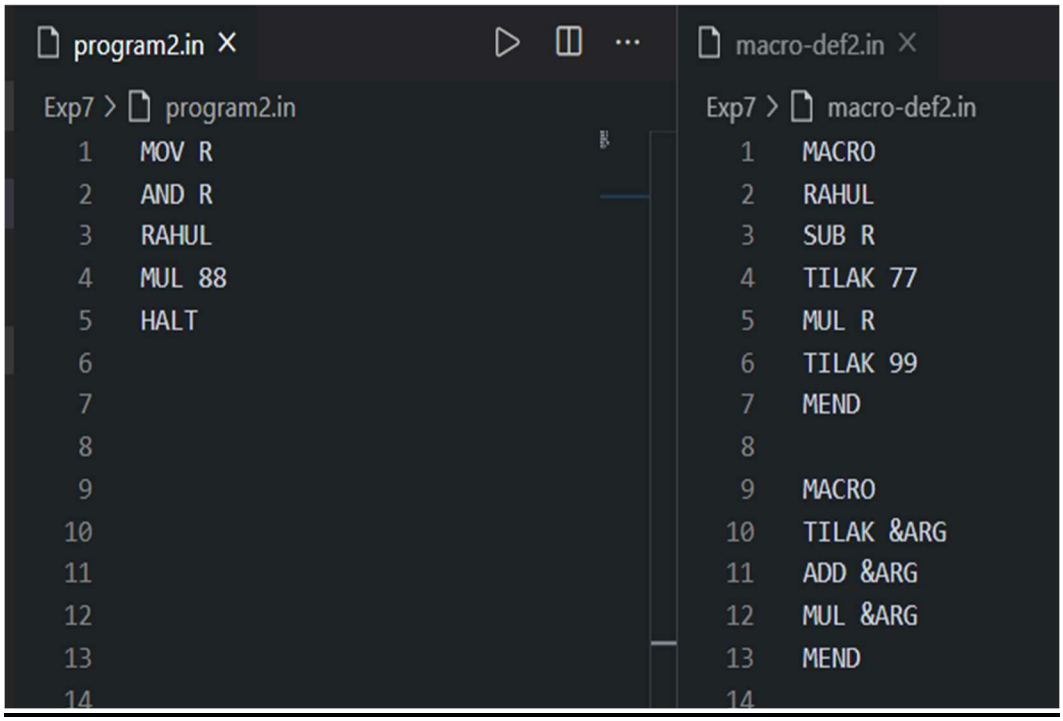
Total number of instructions in the final expanded source code = 10

Program:

```
#Arin Mashta TE-A-142
#EXPT NO 7:- Implementation of 2-levels nested macro & its expansion.
import sys
import sys

macro_file = sys.argv[1]
program_file = sys.argv[2]

macro_cache = {}

with open(macro_file) as f:
    data = [i.strip() for i in f.readlines()]

macro_state = False
for i in range(len(data)):

    if data[i] == 'MACRO':
        i = i + 1
        if data[i].split(" ")[0].startswith("&"):
            label = data[i].split(" ")[0]
            mname = data[i].split(" ")[1]
            pholders = ''.join(data[i].split(" ")[2:]).split(',')
        else:
            label = None
            mname = data[i].split(" ")[0]
            pholders = ''.join(data[i].split(" ")[1:]).split(',')


        pholder = {}
        count = 0
```

```
        for j in pholders:
            pholder[j] = "{" + f"{count}" + "}"
            count += 1
        # print(pholders)
        macro_cache[mname] = []
        i += 1
        while data[i] != 'MEND':
            for j in pholders:
                data[i] = data[i].replace(j, pholder[j], -1)
            if label != None:
                data[i] = data[i].replace(label, "{"+f"{count}"+"}")
                # print(j, data[i])

            macro_cache[mname].append(data[i])
            i += 1
        i += 1

macro_calls = 0
src_inst = 0
macro_calls_inst = 0
total = 0
# print(macro_cache)
print()
with open(program_file) as f:
    data = [i.strip() for i in f.readlines()]

    for qwe in range(2):
        output = []
        for i in data:
            if len(i.split(" ")) > 1 and i.split(" ")[1] in macro_cache:
                macro_calls += 1
                macro_calls_inst = len(macro_cache[i.split(" ")[1]])
                output.append("")
                for j in macro_cache[i.split(" ")[1]]:
                    output.append(j.format(*''.join(i.split(" ")[2:]).split(","), i
.split(" ")[0]))
                    total += 1
                output.append("")
            elif i.split(" ")[0] in macro_cache:
                macro_calls += 1
                macro_calls_inst = len(macro_cache[i.split(" ")[0]])
                output.append("")
                for j in macro_cache[i.split(" ")[0]]:
                    output.append(j.format(*''.join(i.split(" ")[1:]).split(",")))
                    total += 1
                output.append("")
```

```
            else:
                src_inst += 1
                output.append(i)
                total += 1
        data = output
    for i in data:
        print(i)
print()
print(f"No. of instructions in input source code (excluding Macro calls): {src_inst
}")
print(f"No. of macro calls: {macro_calls}")
print(f"No. of instructions in macro calls: {macro_calls_inst}")
print(f"Total instructions: {total}")
```



OUTPUT:

```
PS D:\experiments\SPCC\SPCC\Exp6> py main2.py macro-def2.in program2.in

MOV R
AND R

SUB R

ADD 77
MUL 77

MUL R

ADD 99
MUL 99


MUL 88
HALT

No. of instructions in input source code (excluding Macro calls): 4
No. of macro calls: 1
No. of instructions in macro calls: 2,2
Total instructions: 10
PS D:\experiments\SPCC\SPCC\Exp6>
```

Lokmanya Tilak College of Engineering

Navi Mumbai

DEPARTMENT OF COMPUTER ENGINEERING

Academic Year (2020-2021)

(EVEN SEM) (CBCGS)

Course Name: System Software Lab
Course Code: CSL602
Experiment No.: 8
Title of the Experiment: Design of absolute loader with example.
Name of Student: Arin Mashta
Student Roll No.: 142
Year/Semester/Div: TE/Sem-VI/A

Expt-8:

Aim: Design of absolute loader with example.

Instructions to the students:

1) Consider an hypothetical processor with approx. 20 assembly language instructions.
2) Define a standard MOT for the instructions with only 3 columns namely Mnemonics, Op-code & size. (You may use the table shown below or develop a new)
3) Write a sample ALP using few instructions from the MOT.
4) Write a program using any language (preferably C) to generate machine code of ALP.
5) Input the absolute address of the machine code where the first byte is to be loaded.
6) The output shall display in 3 columns namely Absolute address, ALP instruction & machine code.
7) Also display the statistical data: Number of ALP instructions, size of machine code (in bytes) & the absolute address where the object module is loaded.

MOT format:

Mnemonics	Op-code	Size
MOV R	01	1
ADD R	02	1
SUB R	03	1
MUL R	04	1
DIV R	05	1
AND R	06	1
OR R	07	1
ADD data	08	2
SUB data	09	2
MUL data	10	2
DIV data	11	2
AND data	12	2
OR data	13	2

LOAD adaddress	14	3
STORE address	15	3
DCR R	16	1
INC R	17	1
JMP address	18	3
JNZ address	19	3
HALT	20	1

Instruction with data is 2 byte, first byte is the op-code byte & second byte is data byte itself. Instruction with address is 3 byte (as address assumed here is 16 bit), first byte is the op-code byte & second & third bytes are the address bytes.

Input ALP:

ADD 20

MOV R

OR 55

MUL R

STORE 2000

HALT

Input absolute address of the first instruction = 1000

Output of the absolute loader:

Absolute address	ALP instruction	Object code
1000	ADD 20	08
1001		20
1002	MOV R	01
1003	OR 55	13

1004		55
1005	MUL R	04
1006	STORE 2000	15
1007		20
1008		00
1009	HALT	20

Statistical output:

Number of assembly language instruction in the program = 6

Size of the object code (in bytes) = 10

Object code is loaded in memory from absolute address 1000 to 1009

Program:

main3.py

```
# Arin Mashta TE-A-142
# EXPT NO 8- DESIGN OF ABSOLUTE LOADER
import os
import sys
with os.popen(f"python sub-main2.py {sys.argv[1]} {sys.argv[2]}") as f:
    code = f.read()
    with open(".temp_cache.in", "w") as f:
        f.write(code.strip())
print("Enter Absolute address for the first instruction:")

with os.popen("python sub-main1.py .temp_cache.in") as f:
    print(f.read())
os.remove(".temp_cache.in")
```

sub-main1.py

```
# Arin Mashta TE-A-142
# EXPT NO 8- DESIGN OF ABSOLUTE LOADER
import sys

data = {
"MOV": [1, 1],
```

```
"ADD": [2, 1, 8, 2],
"SUB": [3, 1, 9, 2],
"MUL": [4, 1, 10, 2],
"DIV": [5, 1, 11, 2],
"AND": [6, 1, 12, 2],
"OR": [7, 1, 13, 2],
"LOAD":  [14, 3, 'a'],
"STORE": [15, 3, 'a'],
"DCR": [16, 1],
"INC": [17, 1],
"JMP": [18, 3, 'a'],
"JNZ": [19, 3, 'a'],
"HALT": [20, 1],
}

def add_make(num):
    d = "{:04d}".format(num)
    return d[:2] + " " + d[2:]


if len(sys.argv) == 1:
    print("Error")
    exit(1)

with open(sys.argv[1], "r") as f:
    code = f.readlines()

rel_addr = [0]
mac_code = []
sym = {}
for i in code:
    line = i.strip().split(" ")
    if line[0].endswith(":"):
        sym[line[0][:-1]] = rel_addr[-1]
        line = line[1:]
    if line[0].upper() not in data:
        print("Error in parsing the instruction: ")
        print("".join(line))
        exit(1)
    else:
        ins = data[line[0].upper()]
        if len(line) == 0:
            continue
        elif len(line) == 1:
            rel_addr.append(rel_addr[-1] + ins[-1])
            mac_code.append("{:02d}".format(ins[0]))
```

```
            continue
        if len(ins) == 2:
            rel_addr.append(rel_addr[-1] + ins[-1])
            if len(line[1]) == 1:
                mac_code.append("{:02d}".format(ins[0]))
            else:
                mac_code.append("{:02d} {}".format(ins[0], ' '.join([line[1][j:j+2]
 for j in range(0, len(line[1]), 2)])))
        elif len(ins) == 3:
            rel_addr.append(rel_addr[-1] + ins[1])
            mac_code.append(["{:02d}".format(ins[0]), line[-1]])
        else:
            if len(line[1]) == 1:
                rel_addr.append(rel_addr[-1] + ins[1])
                mac_code.append("{:02d}".format(ins[0]))
            else:
                rel_addr.append(rel_addr[-1] + ins[3])
                mac_code.append("{:02d} {}".format(ins[2], ' '.join([line[1][j:j+2]
 for j in range(0, len(line[1]), 2)])))

# print(sym)
for i in range(len(mac_code)):
    if type(mac_code[i]) == list:
        if mac_code[i][-1] not in sym:
            try:
                d = int(mac_code[i][-1])
                mac_code[i] = mac_code[i][0] + " " + add_make(d)
            except:
                print("Broken Link at: {}".format(code[i]))
                exit(-1)
        else:
            mac_code[i] = mac_code[i][0] + " " + add_make(sym[mac_code[i][-1]])
print()
absAdd = int(input(""))
print()
print(" _____________________________________________")
print("|Absolute address | Alp Instruction |  Object Code  |")
print("|_________________|_________________|_______________|")
for i in range(len(code)):
    print("|     {:02d}        | {:<14}  | {:<14}|".format(absAdd + rel_addr[i], co
de[i].strip(), mac_code[i].strip()))
print("|_________________|_________________|_______________|")
print()
print(f"Number of Assembly language instruction in the program: {len(mac_code)}")
print(f"Size of the object code (in bytes): {size}")
```

```
print(f"Object code is loaded in memory from absolute address {rel_addr[0] + absAdd
} to {rel_addr[-2] + absAdd}")
print()
```

sub-main2.py

```
# Arin Mashta TE-A-142
# EXPT NO 8- DESIGN OF ABSOLUTE LOADER
import sys

data = {
"MOV": [1, 1],
"ADD": [2, 1, 8, 2],
"SUB": [3, 1, 9, 2],
"MUL": [4, 1, 10, 2],
"DIV": [5, 1, 11, 2],
"AND": [6, 1, 12, 2],
"OR": [7, 1, 13, 2],
"LOAD":  [14, 3, 'a'],
"STORE": [15, 3, 'a'],
"DCR": [16, 1],
"INC": [17, 1],
"JMP": [18, 3, 'a'],
"JNZ": [19, 3, 'a'],
"HALT": [20, 1],
}

def add_make(num):
    d = "{:04d}".format(num)
    return d[:2] + " " + d[2:]


if len(sys.argv) == 1:
    print("Error")
    exit(1)

with open(sys.argv[1], "r") as f:
    code = f.readlines()

rel_addr = [0]
mac_code = []
sym = {}
for i in code:
    line = i.strip().split(" ")
    if line[0].endswith(":"):
        sym[line[0][:-1]] = rel_addr[-1]
        line = line[1:]
```

```
    if line[0].upper() not in data:
        print("Error in parsing the instruction: ")
        print("".join(line))
        exit(1)
    else:
        ins = data[line[0].upper()]
        if len(line) == 0:
            continue
        elif len(line) == 1:
            rel_addr.append(rel_addr[-1] + ins[-1])
            mac_code.append("{:02d}".format(ins[0]))
            continue
        if len(ins) == 2:
            rel_addr.append(rel_addr[-1] + ins[-1])
            if len(line[1]) == 1:
                mac_code.append("{:02d}".format(ins[0]))
            else:
                mac_code.append("{:02d} {}".format(ins[0], ' '.join([line[1][j:j+2]
 for j in range(0, len(line[1]), 2)])))
        elif len(ins) == 3:
            rel_addr.append(rel_addr[-1] + ins[1])
            mac_code.append(["{:02d}".format(ins[0]), line[-1]])
        else:
            if len(line[1]) == 1:
                rel_addr.append(rel_addr[-1] + ins[1])
                mac_code.append("{:02d}".format(ins[0]))
            else:
                rel_addr.append(rel_addr[-1] + ins[3])
                mac_code.append("{:02d} {}".format(ins[2], ' '.join([line[1][j:j+2]
 for j in range(0, len(line[1]), 2)])))

# print(sym)
for i in range(len(mac_code)):
    if type(mac_code[i]) == list:
        if mac_code[i][-1] not in sym:
            try:
                d = int(mac_code[i][-1])
                mac_code[i] = mac_code[i][0] + " " + add_make(d)
            except:
                print("Broken Link at: {}".format(code[i]))
                exit(-1)
        else:
            mac_code[i] = mac_code[i][0] + " " + add_make(sym[mac_code[i][-1]])
print()
absAdd = int(input(""))
print()
```

```
print(" _____________________________________________")
print("|Absolute address | Alp Instruction |  Object Code  |")
print("|_________________|_________________|_______________|")
for i in range(len(code)):
    print("|     {:02d}        | {:<14}  | {:<14}|".format(absAdd + rel_addr[i], co
de[i].strip(), mac_code[i].strip()))
print("|_________________|_________________|_______________|")
print()

import sys

macro_file = sys.argv[1]
program_file = sys.argv[2]

macro_cache = {}

with open(macro_file) as f:
    data = [i.strip() for i in f.readlines()]

macro_state = False
for i in range(len(data)):

    if data[i] == 'MACRO':
        i = i + 1
        if data[i].split(" ")[0].startswith("&"):
            label = data[i].split(" ")[0]
            mname = data[i].split(" ")[1]
            pholders = ''.join(data[i].split(" ")[2:]).split(',')
        else:
            label = None
            mname = data[i].split(" ")[0]
            pholders = ''.join(data[i].split(" ")[1:]).split(',')


        pholder = {}
        count = 0
        for j in pholders:
            pholder[j] = "{" + f"{count}" + "}"
            count += 1
        # print(pholders)
        macro_cache[mname] = []
        i += 1
        while data[i] != 'MEND':
            for j in pholders:
                data[i] = data[i].replace(j, pholder[j], -1)
            if label != None:
```

```
                data[i] = data[i].replace(label, "{"+f"{count}"+"}")
                # print(j, data[i])

            macro_cache[mname].append(data[i])
            i += 1
        i += 1

macro_calls = 0
src_inst = 0
macro_calls_inst = 0
total = 0
# print(macro_cache)
print()
with open(program_file) as f:
    data = [i.strip() for i in f.readlines()]

    for qwe in range(2):
        output = []
        for i in data:
            if len(i.split(" ")) > 1 and i.split(" ")[1] in macro_cache:
                macro_calls += 1
                macro_calls_inst = len(macro_cache[i.split(" ")[1]])
                output.append("")
                for j in macro_cache[i.split(" ")[1]]:
                    output.append(j.format(*''.join(i.split(" ")[2:]).split(","), i
.split(" ")[0]))
                    total += 1
                output.append("")
            elif i.split(" ")[0] in macro_cache:
                macro_calls += 1
                macro_calls_inst = len(macro_cache[i.split(" ")[0]])
                output.append("")
                for j in macro_cache[i.split(" ")[0]]:
                    output.append(j.format(*''.join(i.split(" ")[1:]).split(",")))
                    total += 1
                output.append("")
            else:
                src_inst += 1
                output.append(i)
                total += 1
        data = output
    for i in data:
        print(i)
```

program3.txt

Exp6 > program3.txt

```
1    ADD 20
2    MOV R
3    OR 55
4    MUL R
5    STORE 2000
6    HALT
7
```

OUTPUT:

C:\Windows\System32\cmd.exe

```
C:\Users\admin\PycharmProjects\exp 8>python main3.py macro-def3.txt program3.txt
Enter Absolute address for the first instruction:
1000


 _____________________________________
|Absolute address | Alp Instruction |  Object Code  |
|_________________|_________________|_______________|
|      1000       | ADD 20          | 08 20         |
|      1002       | MOV R           | 01            |
|      1003       | OR 55           | 13 55         |
|      1005       | MUL R           | 04            |
|      1006       | STORE 2000      | 15 20 00      |
|      1009       | HALT            | 20            |
|_________________|_________________|_______________|

Number of Assembly language instructions in the program = 6
Size of the object code (in bytes) = 10
Object code is loaded in memory from absolute address 1000 to 1009


C:\Users\admin\PycharmProjects\exp 8>
```

Lokmanya Tilak College of Engineering

Navi Mumbai

DEPARTMENT OF COMPUTER ENGINEERING

Academic Year (2020-2021)

(EVEN SEM) (CBCGS)

Course Name: System Software Lab
Course Code: CSL602
Experiment No.: 9
Title of Experiment: Write a lex program to count blank spaces, words etc in given statement.
Name of Student: Arin Mashta
Student Roll No.: 142
Year/Semester/Div: TE/Sem-VI/A

Expt9

Aim: Write a lex program to count blank spaces, words etc. in the statement.

Theory:

- Lex is a program that generates lexical analyzer. It is used with YACC parser generator.
- The lexical analyzer is a program that transforms an input stream into a sequence of tokens.
- It reads the input stream and produces the source code as output through implementing the lexical analyzer in the C program.

The function of Lex is as follows:

- Firstly lexical analyzer creates a program lex.1 in the Lex language. Then Lex compiler runs the lex.1 program and produces a C program lex.yy.c.
- Finally C compiler runs the lex.yy.c program and produces an object program a.out.
- a.out is lexical analyzer that transforms an input stream into a sequence of tokens.

Lex file format:

A Lex program is separated into three sections by %% delimiters. The formal of Lex source is as follows:

1. { definitions }
2. %%
3. { rules }
4. %%
5. { user subroutines }

Definitions include declarations of constant, variable and regular definitions.
Rules define the statement of form p1 {action1} p2 {action2} ...pn {action}.
Where **pi** describes the regular expression and **action1** describes the actions what action the lexical analyzer should take when pattern pi matches a lexeme.
User subroutines are auxiliary procedures needed by the actions. The subroutine can be loaded with the lexical analyzer and compiled separately.

Program:

```
/* Arin Mashta TE-A-142 */
/* EXPT NO 9- Write a lex program to count blank spaces, words etc in given stateme
nt. */

%{
#include <stdio.h>
int sc=0, ws=0;
%}

%%
([ ])+ sc++;
([a-zA-Z0-9\.\,'";:?!`])* ws++;
%%

int main() {
    yylex();
    printf("No. of spaces: %d\n", sc);
    printf("No. of words: %d\n", ws);
}
```

sonnet.txt

```
Exp9 > sonnet.txt
1   Lex is a program that generates lexical analyzer It is used with YACC parser generator
2
```

OUTPUT:

```
C:\Users\admin\Downloads\Experiments\Exp9\lex> ls
a.out gfg.l lex.yy.c
C:\Users\admin\Downloads\Experiments\Exp9\lex> cat ../sonnet.txt | ./a.out




No. of spaces: 14
No. of words: 15

C:\Users\admin\Downloads\Experiments\Exp9\lex>
```

Lokmanya Tilak College of Engineering

Navi Mumbai

DEPARTMENT OF COMPUTER ENGINEERING

Academic Year (2020-2021)

(EVEN SEM) (CBCGS)

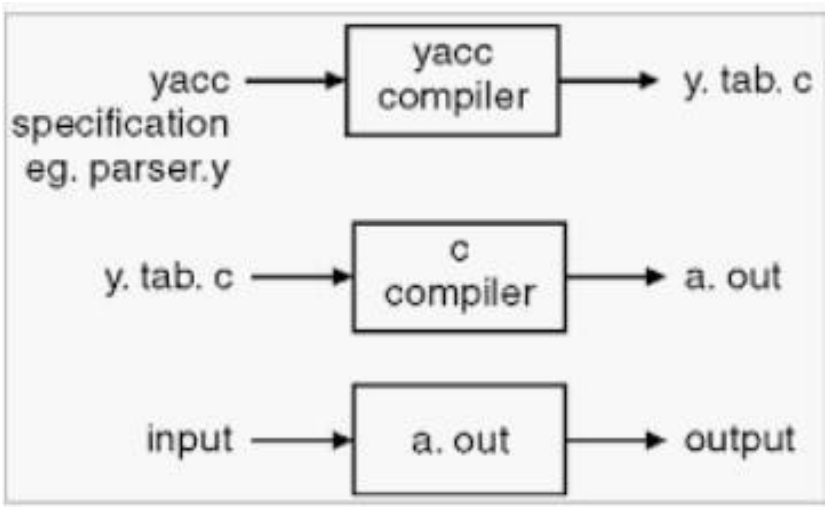
Course Name: System Software Lab
Course Code: CSL602
Experiment No.: 10
Title of Experiment: Write a program to recognize valid arithmetic expression that uses operators +, -, * and / using YACC
Name of Student: Arin Mashta
Student Roll No.: 142
Year/Semester/Div: TE/Sem-VI/A

Expt10

Aim: Write a program to recognize valid arithmetic expression that uses operators +,-,* and / using YACC.

Theory:

YACC (Yet Another Compiler Compiler) is a tool used to generate a parser. This document is a tutorial for the use of YACC to generate a parser for ExpL. YACC translates a given Context Free Grammar (CFG) specifications (input in input_file.y) into a C implementation (y.tab.c) of a corresponding push down automaton (i.e., a finite state machine with a stack). This C program when compiled, yields an executable parser.



The source SIL program is fed as the input to the generated parser (a.out). The *parser* checks whether the program satisfies the syntax specification given in the input_file.y file.
YACC was developed by Stephen C. Johnson at Bell labs.

ALGORITHM:

Step1: Start the program.
Step2: Reading an expression.
Step3: Checking the validating of the given expression according to the rule using yacc.
Step4: Using expression rule print the result of the given values
Step5: Stop the program.

Program:

```
/* Arin Mashta TE-A-142 */
/* EXPT NO 10- Write a program to recognize valid arithmetic expression that uses o
perators +,-,* and / using YACC. */

%{
#include <stdio.h>
#include <string.h>
    int operators_count = 0, operands_count = 0, valid = 1, top = -1, l = 0, j = 0;
    char operands[10][10], operators[10][10], stack[100];
%}
%%
"(" {
    top++;
    stack[top] = '(';
}
"{" {
    top++;
    stack[top] = '{';
}
"[" {
    top++;
    stack[top] = '[';
}
")" {
    if (stack[top] != '(') {
        valid = 0;
    }
    else if(operands_count>0 && (operands_count-operators_count)!=1){
        valid=0;
    }
    else{
        top--;
        operands_count=1;
        operators_count=0;
    }
}
"}" {
    if (stack[top] != '{') {
        valid = 0;
    }
    else if(operands_count>0 && (operands_count-operators_count)!=1){
        valid=0;
    }
    else{
        top--;
```

```
        operands_count=1;
        operators_count=0;
    }
}
"]" {
    if (stack[top] != '[') {
        valid = 0;
    }
    else if(operands_count>0 && (operands_count-operators_count)!=1){
        valid=0;
    }
    else{
        top--;
        operands_count=1;
        operators_count=0;
    }

}
"+"|"-"|"*"|"/" {
    operators_count++;
    strcpy(operators[l], yytext);
    l++;
}
[0-9]+|[a-zA-Z][a-zA-Z0-9_]* {
    operands_count++;
    strcpy(operands[j], yytext);
    j++;
}
%%

int yywrap()
{
    return 1;
}
int main()
{
    int k;
    printf("Enter the arithmetic expression: ");
    yylex();

    if (valid == 1 && top == -1) {
        printf("\nValid Expression\n");
    }
    else
        printf("\nInvalid Expression\n");
```

```
    return 0;
}
```

OUTPUT:

```
C:\Users\admin\Downloads\Experiments\Exp9\yacc> ./a.out
Enter the arthimetic expression: a+b*c

Valid Expression
C:\Users\admin\Downloads\Experiments\Exp9\yacc> ./a.out
```