Experiment No.	**03**
Experiment Title	**For any Assembly input for a hypothetical machine, implement pass 2 by using the intermediate code generated as input to pass 2 program which was generated as output in pass 1.**
Student Name	Ronak Surve
Roll No.	64
Objectives:	**1) Students will be able to learn various file handling operations** **2) Students will be able to produce the target code using the intermediate representation generated in Pass 1.** **3) Students will be able to implement the working of Pass 2 of 2 pass Assembler**
Theory /Algorithm:	The theory behind a two-pass assembler is to separate the tasks of assembling a source code written in assembly language into two distinct stages, each with a specific objective. The two passes are: Pass 1: This pass scans the source code to gather information about the labels used in the code, their addresses, and the size of the program. This information is recorded in a symbol table, which is used in the second pass to resolve symbolic references and replace them with absolute addresses. The main objective of the first pass is to gather information and prepare the necessary data for the second pass. Pass 2: This pass uses the information gathered in the first pass, such as the symbol table, to generate the final object code. During the second pass, the symbolic references are replaced with the corresponding absolute addresses, and the final object code is generated. The main objective of the second pass is to generate the final object code and perform error checking. The theory behind the two-pass approach is to optimize the accuracy and efficiency of the assembly process by dividing it into two separate stages. The first pass can detect errors and report them to the user before the final object code is generated, allowing for easier correction and debugging. The second pass can generate optimized and efficient object code using the information gathered in the first pass. Overall, the two-pass assembler theory is based on the idea of dividing the assembly process into two separate stages, each with a specific objective, in order to optimize the accuracy and

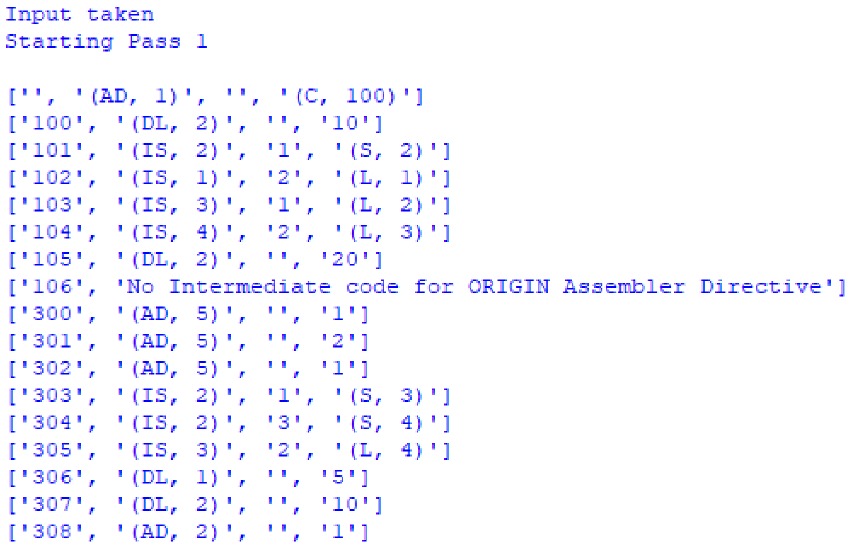
efficiency of the assembly process.

Program Code:

```
# Initialisation

MOT = {
    "MOVEM": 1,
    "MOVER": 2,
    "ADD": 3,
    "SUB": 4,
    "MUL": 5,
    "DIV": 6,
    "BC": 7,
    "COMP": 8,
    "READ": 9,
    "PRINT": 10
}

POT = {
    "START": 1,
    "END": 2,
    "EQU": 3,
    "ORIGIN": 4,
    "LTORG": 5
}

DL = {
    "DS": 1,
    "DC": 2
}

REG = {
    "AREG": 1,
    "BREG": 2,
    "CREG": 3,
    "DREG": 4
}

SYMTAB = []

LITTAB = []

POOLTAB = []
pool_1st = 1

ASM = []
# Initialisation complete

# Take input program
ASM.append(input("Start typing the assembly program \n"))

while ASM[-1].upper() != "END":
    ASM.append(input())

print("Input taken")
# Pass 1
print("Starting Pass 1\n")
loc = 0
ic = []
for line in ASM:
    cmd = line.split(" ")

    if cmd[0].upper() in MOT:
        ic.append([f"{loc}", f"(IS, {MOT[cmd[0].upper()]})"])
        loc += 1
        if cmd[1].upper().replace(",", "") in REG: # e.g. MOVER AREG, B
            ic[-1].append(f"{REG[cmd[1].upper().replace(',', '')]}")

        if cmd[2] is None: # e.g. READ A
            continue

        elif cmd[2][0] == "=": # e.g. MOVEM CREG, ='5'
            LITTAB.append([f"{len(LITTAB) + 1}", f"{cmd[2]}", ""])
            ic[-1].append(f"(L, {len(LITTAB)})")

        else: # e.g. MOVER AREG, B
            SYMTAB.append([f"{len(SYMTAB) + 1}", f"{cmd[2]}", ""])
            ic[-1].append(f"(S, {len(SYMTAB)})")

    elif cmd[0].upper() in POT:
        if POT[cmd[0].upper()] == 1: # START
            loc = int(cmd[1])
            ic.append(["", "(AD, 1)", "", f"(C, {loc})"])

        elif POT[cmd[0].upper()] == 2: # END
            count = 0
            for record in LITTAB:
                if record[-1] == "": # Find all records with empty address fields
                    record.pop(-1)
                    record.append(f"{loc}")
                    ic.append([f"{loc}", "(AD, 2)", "", f"{record[1][-2]}"]) #
'record[1][-2]' is value of literal
                    count += 1
                    loc += 1
            POOLTAB.append([f"{pool_1st}", f"{count}"])
            pool_1st += count
            break

        elif POT[cmd[0].upper()] == 4: # ORIGIN
            ic.append([f"{loc}", "No Intermediate code for ORIGIN Assembler
Directive"])
            loc = int(cmd[1]) # Go to new address

        elif POT[cmd[0].upper()] == 5: # LTORG
            count = 0
            for record in LITTAB:
                if record[-1] == "": # Find all records with empty address fields
                    record.pop(-1)
                    record.append(f"{loc}")
                    ic.append([f"{loc}", "(AD, 5)", "", f"{record[1][-2]}"]) #
'record[1][-2]' is value of literal
                    count += 1
                    loc += 1
            POOLTAB.append([f"{pool_1st}", f"{count}"])
            pool_1st += count

    else:
        isPresent = False
        for record in SYMTAB:
            if record[1] == cmd[0].upper():
                record.pop(-1)
                record.append(f"{loc}")
                isPresent = True
                break
        if not isPresent:
            SYMTAB.append([f"{len(SYMTAB) + 1}", f"{cmd[0]}", f"{loc}"])

        if cmd[1].upper() in DL:
            ic.append([f"{loc}", f"(DL, {DL[cmd[1].upper()]})", "", f"{cmd[2]}"])

        elif cmd[1].upper() in MOT:
            ic.append([f"{loc}", f"(IS, {MOT[cmd[1].upper()]})"])
            if cmd[2].upper().replace(",", "") in REG: # e.g. MOVER AREG, B
                ic[-1].append(f"{REG[cmd[2].upper().replace(',', '')]}")

            if cmd[3] is None: # e.g. READ A
                continue

            elif cmd[3][0] == "=": # e.g. MOVEM CREG, ='5'
                LITTAB.append([f"{len(LITTAB) + 1}", f"{cmd[3]}", ""])
                ic[-1].append(f"(L, {len(LITTAB)})")

            else: # e.g. MOVER AREG, B
                SYMTAB.append([f"{len(SYMTAB) + 1}", f"{cmd[3]}", ""])
                ic[-1].append(f"(S, {len(SYMTAB)})")

        elif POT[cmd[1].upper()] == 3: # EQU
            ic.append([loc, "No Intermediate code for EQU Assembler Directive"])
            SYMTAB[-1].pop(-1)
            for record in SYMTAB:
                if record[1] == cmd[2].upper(): # Change 1st symbol's address to 2nd symbol's address
                    SYMTAB[-1].append(record[2])

        loc +=1

for line in ic:
    print(line)

# Pass 2

print("\nStarting Pass 2\n")
mc = []
for record in ic:
    if record[0] == "": # Skip the START statement
        continue
    if record[1][0] != '(': # Skip the EQU & ORIGIN statements
        continue
    statement = record[1].replace('(','').replace(')','').split(' ') # Split statement into statement type and its value
    operand = record[3].replace('(','').replace(')','').split(' ') # Split operand 2 into literal/ symbol type and its value
    if len(operand) > 1: # if operand is a value, skip
        record.pop(3)
        if operand[0] == 'S,':
            record.append(SYMTAB[int(operand[1]) - 1][-1]) # Find address location
        elif operand[0] == 'L,':
            record.append(LITTAB[int(operand[1]) - 1][-1]) # Find address location

    mc.append([f"{record[0]}+", statement[1] if statement[0] == 'IS,' else '', record[2], record[3]])
```

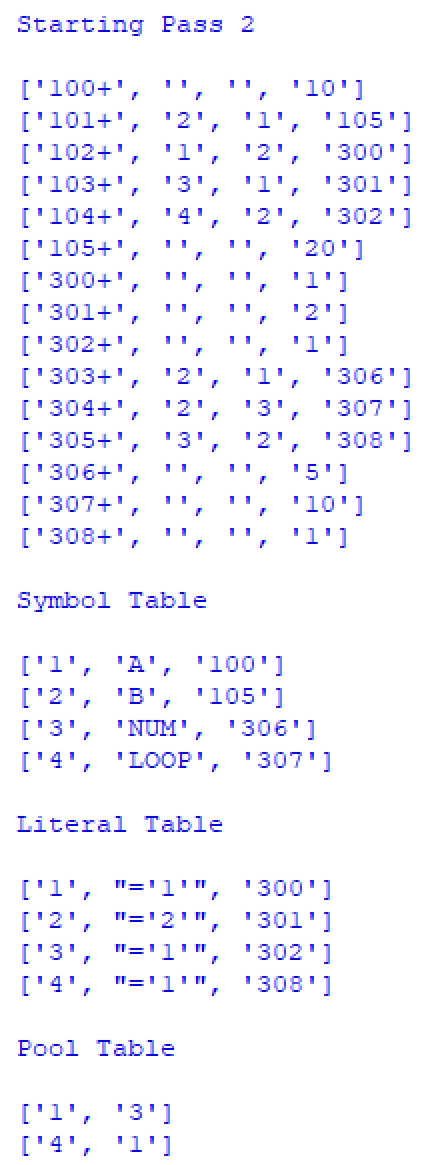
```
for line in mc:
    print(line)

print("\nSymbol Table\n")
for line in SYMTAB:
    print(line)
print("\nLiteral Table\n")
for line in LITTAB:
    print(line)
print("\nPool Table\n")

for line in POOLTAB:
    print(line)
```

Input to the Program:



```
Input taken
Starting Pass 1

['', '(AD, 1)', '', '(C, 100)']
['100', '(DL, 2)', '', '10']
['101', '(IS, 2)', '1', '(S, 2)']
['102', '(IS, 1)', '2', '(L, 1)']
['103', '(IS, 3)', '1', '(L, 2)']
['104', '(IS, 4)', '2', '(L, 3)']
['105', '(DL, 2)', '', '20']
['106', 'No Intermediate code for ORIGIN Assembler Directive']
['300', '(AD, 5)', '', '1']
['301', '(AD, 5)', '', '2']
['302', '(AD, 5)', '', '1']
['303', '(IS, 2)', '1', '(S, 3)']
['304', '(IS, 2)', '3', '(S, 4)']
['305', '(IS, 3)', '2', '(L, 4)']
['306', '(DL, 1)', '', '5']
['307', '(DL, 2)', '', '10']
['308', '(AD, 2)', '', '1']
```

Output of the program:



```
Starting Pass 2

['100+', '', '', '10']
['101+', '2', '1', '105']
['102+', '1', '2', '300']
['103+', '3', '1', '301']
['104+', '4', '2', '302']
['105+', '', '', '20']
['300+', '', '', '1']
['301+', '', '', '2']
['302+', '', '', '1']
['303+', '2', '1', '306']
['304+', '2', '3', '307']
['305+', '3', '2', '308']
['306+', '', '', '5']
['307+', '', '', '10']
['308+', '', '', '1']

Symbol Table

['1', 'A', '100']
['2', 'B', '105']
['3', 'NUM', '306']
['4', 'LOOP', '307']

Literal Table

['1', "='1'", '300']
['2', "='2'", '301']
['3', "='1'", '302']
['4', "='1'", '308']

Pool Table

['1', '3']
['4', '1']
```

Outcome of the Experiment:

The outcome of a pass 2 assembler is the final object code, which is a machine-readable representation of the source code written in assembly language. During the second pass, the assembler uses the information gathered in the first pass, such as the symbol table and the listing of the source code, to generate the object code. The final object code is typically stored in a file that can be loaded into memory and executed by the computer. The second pass also performs error checking and generates error messages for any problems detected in the source code, such as invalid operands or undefined symbols. The final object code generated by the pass 2 assembler can also be optimized for size or performance by applying various code optimization techniques.

References:

https://www.geeksforgeeks.org/single-pass-two-pass-and-multi-pass-compilers/