GHARDA FOUNDATION

GIT GHARDA INSTITUTE OF TECHNOLOGY

GHARDA FOUNDATION'S

GHARDA INSTITUTE OF TECHNOLOGY, LAVEL

COMPUTER ENGINEERING DEPARTMENT

A/P: Lavel, Tal.Khed Dist. Ratnagiri

Evaluation Sheet

Class: TE-Computer

Semester: VI

Subject: **System Programming & Compiler Construction**

Experiment No: 1

Title of Experiment: Implementation of Two Pass Assembler

Sr. No.	Evaluation Criteria	Max Marks	Marks Obtained
1	Practical Performance	10	
2	Oral	5	
	Total	15	

Signature of Subject Teacher

Theory:

Two Pass Assembler

A two-pass assembler is a type of assembler that goes through the source code twice to generate the machine code. In the first pass, it reads the entire source code to build a symbol table and detects any syntax errors. In the second pass, it generates the machine code using the symbol table created in the first pass.

Steps involved in a two-pass assembler -

Pass 1: In this pass, the assembler reads the entire source code and builds a symbol table. The symbol table contains the names of all the labels, variables, and constants used in the code, along with their memory addresses. It also detects any syntax errors and generates error messages if necessary. The symbol table is used in the second pass to generate the machine code.

Pass 2: In this pass, the assembler generates the machine code using the symbol table created in the first pass. It reads the source code line by line and translates each instruction into its corresponding machine code. It also replaces all the labels, variables, and constants with their respective memory addresses. The machine code is then written to an output file.

Advantages of two-pass assembler –

1. It can handle forward references: A forward reference occurs when a label is used before it is defined. In a two-pass assembler, the symbol table is built in the first pass, so it can handle forward references in the second pass.
2. It can generate optimized code: Since a two-pass assembler has access to the entire source code in both passes, it can optimize the generated machine code for size and performance.
3. It can detect errors early: The first pass of a two-pass assembler detects syntax errors and generates error messages, so they can be fixed before the machine code is generated.

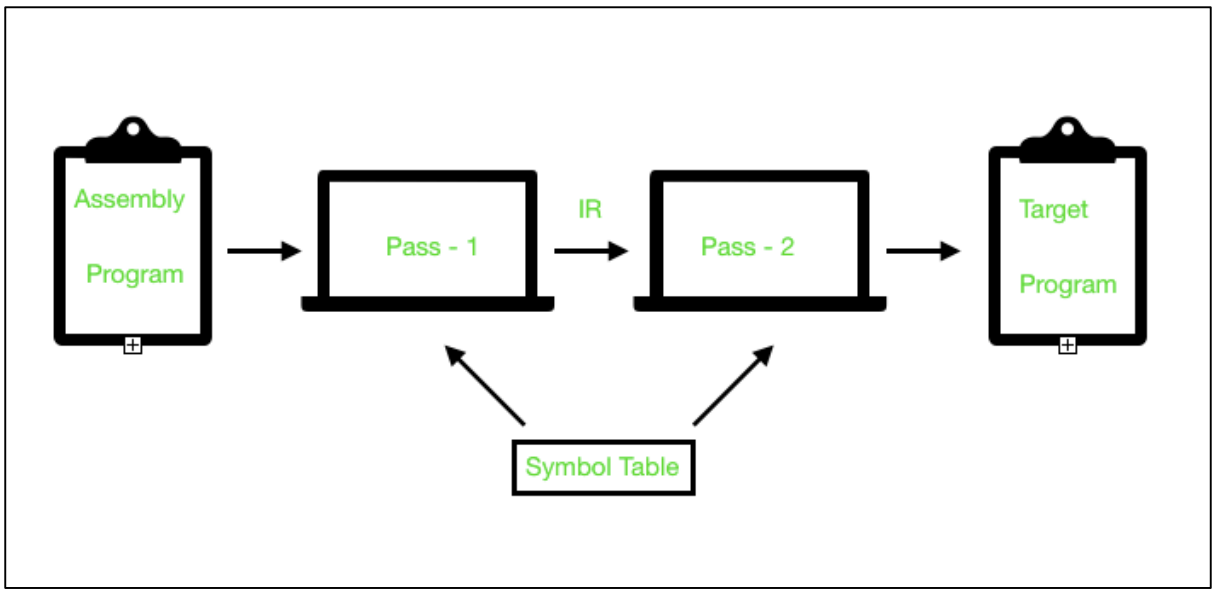

Fig. Working of Two Pass Assembler

Additional Details of Two-Pass Assembler –

1. **Symbol Table**: The symbol table is a data structure that is used to store information about the labels, variables, and constants used in the source code. It contains the name of the symbol, its type, and its memory location. The symbol table is created in the first pass of the assembler and is used in the second pass to generate the machine code.
2. **Pass 1**: In the first pass, the assembler reads the source code and builds the symbol table. It also checks for syntax errors and generates error messages if necessary. The symbol table is built by scanning each line of code to identify labels, variables, and constants and adding them to the symbol table.
3. **Pass 2**: In the second pass, the assembler generates the machine code using the symbol table created in the first pass. It reads each line of code and translates it into its corresponding machine code. The assembler uses the symbol table to replace all the labels, variables, and constants with their respective memory locations. The machine code is then written to an output file.
4. **Optimization**: A two-pass assembler can optimize the generated machine code for size and performance since it has access to the entire source code in both passes. It can eliminate redundant instructions, use shorter instructions where possible, and reorder instructions to improve performance.

5. **Error Handling**: A two-pass assembler can detect syntax errors and generate error messages in the first pass. This allows the programmer to fix the errors before the machine code is generated. It can also handle forward references, which occur when a label is used before it is defined.

Overall, a two-pass assembler is a powerful tool for generating machine code from assembly language source code. It allows for error detection and optimization, and can handle forward references, making it a popular choice for many programmers.

Program:-(main.py)

```
# -*- coding: utf-8 -*-
"""
@author: MangeshPoskar29

Title: Implementation of two pass assembler

This program will work properly for certain set of assembly commands only
"""

fp = open('program.txt', 'r')
program = fp.read().split("\n")

# print(program)



fp.close()

mnemonic_tab = {'STOP': '00', 'ADD': '01', 'SUB': '02', 'MULT': '03', 'MOVER': '04', 'MOVEM': '05',
'COMP': '06',
            'BC': '07', 'DIV': '08', 'READ': '09', 'PRINT': '10', 'DS': '01', 'DC': '02'}
reg_code = {'AREG': 1, 'BREG': 2, 'CREG': 3, 'DREG': 4}
condition_code = {'LT': 1, 'LE': 2, 'EQ': 3, 'GT': 4, 'GE': 5, 'ANY': 6}
optab = {'STOP': 'AD', 'ADD': 'IS', 'MULT': 'IS', 'MOVER': 'IS', 'MOVEM': 'IS', 'COMP': 'IS', 'BC': 'IS',
'DC': 'DL',
      'DS': 'DL',
      'READ': 'IS',
```

```
        'PRINT': 'IS'}  # declaring operands and their respective types of sentences Imp sent, Decl sent and AD
sym_table = {}  # empty symbol table

print('Content of Mnemonic Table is :\n')
print('Mnemonic', ' Code\n')
for k, v in mnemonic_tab.items():
    print('{0}      {1}'.format(k, v))

print()

print('Content of Opcode table is:\n')
print('Mnemonic', 'Class\n')
for k, v in optab.items():
    print('{0}      {1}'.format(k, v))

print()
print('Input Assembly Code')
print()

# print the source code
# set the value of lc
"""
START 200
MOVER AREG FIRST
ADD AREG SECOND
MOVEM AREG RESULT
```

```
PRINT RESULT

RESULT DS 1

FIRST DC 5

SECOND DC 7

END

"""

for line in program:  # traversing line by line throught the program

   a = line.split()  # splitting the line and it will get converted into list of strings of each line in assembly code

   if a[0] == 'START':  # checking the first element in the list if it is a start symbol and if it is not simply print the line.

      lc = int(a[1])  # converting the 1st element in the list into integer value and storing it into lc

      temp = lc  # the int value stored in lc is stored in temp (here storing the address value as a temporary value)

   print(line)  # printing the line


# Build the symbol table


for line in program:

   l = line.split()  # splitting the lines into list of single words and storing it into l

   for i in l:  # traversing through the l one by one

      if i not in optab and i not in reg_code and i.isdigit() != True and i not in condition_code:  # START FIRST SECOND RESULT END

         sym_table[i] = lc

         lc += 1

print()

print('Content of Symbol Table is:')

print()
```

```
print('Symbol Name', ' Address')
for k, v in sym_table.items():
    print('{0}        {1}'.format(k, v))


#
**************************************************************************
**************************************************************************
**********************
```

```
lc = temp  # taking the value from temp into lc (lc = 200)
print()
print('Intermediate code after PASS-1')
print()
a = list(sym_table.keys())  # making a list of all the keys from symbol table and storing them into
a
for line in program:  # traversing line by line through assembly program
    lexeme = line.split()  # Splitting the assembly code line by line and storing it in the list in the
variable lexeme
    if len(lexeme) == 4:  # if lexeme is greater than length 4 i.e. there are 4 elements in the list
then remove 1st lexeme it from the list
        lexeme.remove(lexeme[0])
    if lexeme[0] in optab:
        if optab[lexeme[0]] == 'AD':
            if len(lexeme) == 1:
                print(lc, (optab[lexeme[0]], mnemonic_tab[lexeme[0]]), '(C,', lexeme[0], ')')
```

```
        if lexeme[0] in optab:
            if optab[lexeme[0]] == 'IS':
                if len(lexeme) == 3:
                    if lexeme[0] == 'BC':
                        print(lc, (optab[lexeme[0]], mnemonic_tab[lexeme[0]]),
condition_code[lexeme[1]], '(S',
                              a.index(lexeme[2]), ')')
                        lc += 1
                    else:
                        print(lc, (optab[lexeme[0]], mnemonic_tab[lexeme[0]]), reg_code[lexeme[1]], '(S',
                              a.index(lexeme[2]), ')')
                        lc += 1
                    if len(lexeme) == 2:
                        print(lc, (optab[lexeme[0]], mnemonic_tab[lexeme[0]]), '(S', a.index(lexeme[1]),
')')
                        lc += 1
        if lexeme[0] not in optab:
            if len(lexeme) == 3:
                print(lc, (optab[lexeme[1]], mnemonic_tab[lexeme[1]]), '(C', lexeme[2], ')')
                lc += 1


            if len(lexeme) == 4:
                print(lc, (optab[lexeme[1]], mnemonic_tab[lexeme[1]]), )
                lc += 1
print()
print('Machine Code after PASS II \n')


lc = temp  # again we are reseting the lc as a starting address of the program here 200
```

```
for line in program:
    lexeme = line.split()  # making a list line by line and storing it into lexeme
    if len(lexeme) == 4:  # 4 elements in the list then remove the first element
        lexeme.remove(lexeme[0])


    if lexeme[0] in optab:
        if optab[lexeme[0]] == 'AD':
            if (len(lexeme) == 1):
                print()
                lc += 1
            else:
                if (lexeme[0] == 'START'): pass
    if lexeme[0] in optab:
        if optab[lexeme[0]] == 'IS':
            if len(lexeme) == 3:  # if there are 3 elements in the list
                if lexeme[0] == 'BC':
                    print(lc, mnemonic_tab[lexeme[0]], condition_code[lexeme[1]],
sym_table[lexeme[2]])
                    lc += 1
                else:
                    print(lc, mnemonic_tab[lexeme[0]], reg_code[lexeme[1]], sym_table[lexeme[2]])
                    lc += 1
            if (len(lexeme) == 2):
                print(lc, mnemonic_tab[lexeme[0]], sym_table[lexeme[1]])
                lc += 1
```

```
    if lexeme[0] not in optab:
        if len(lexeme) == 3:
            print(lc, mnemonic_tab[lexeme[1]], lexeme[2])
            lc += 1
        if len(lexeme) == 4:
            print(mnemonic_tab[lexeme[1]])
            lc += 1
```

(program.txt):-

```
START 200
MOVER AREG FIRST
ADD AREG SECOND
MOVEM AREG RESULT
PRINT RESULT
RESULT DS 1
FIRST DC 5
SECOND DC 7
END
```

Output:-

```
Console × Shell × +
Content of Mnemonic Table is :

Mnemonic  Code

STOP      00
ADD      01
SUB      02
MULT      03
MOVER      04
MOVEM      05
COMP      06
BC      07
DIV      08
READ      09
PRINT      10
DS      01
DC      02

Content of Opcode table is:

Mnemonic Class

STOP      AD
ADD      IS
MULT      IS
MOVER      IS
MOVEM      IS
COMP      IS
BC      IS
DC      DL
DS      DL
READ      IS
PRINT      IS

Input Assembly Code

START 200
MOVER AREG FIRST
ADD AREG SECOND
```

```
Input Assembly Code

START 200
MOVER AREG FIRST
ADD AREG SECOND
MOVEM AREG RESULT
PRINT RESULT
RESULT DS 1
FIRST DC 5
SECOND DC 7
END

Content of Symbol Table is:

Symbol Name  Address
START         200
FIRST         206
SECOND         207
RESULT         205
END         208

Intermediate code after PASS-1

200 ('IS', '04') 1 (S 1 )
201 ('IS', '01') 1 (S 2 )
202 ('IS', '05') 1 (S 3 )

Machine Code after PASS II

200 04 1 206
201 01 1 207
202 05 1 205
203 10 205
204 01 1
205 02 5
206 02 7
>
```