BHARATIYA VIDYA BHAVAN'S
SARDAR PATEL INSTITUTE OF TECHNOLOGY
(Empowered Autonomous Institute Affiliated to University of Mumbai)
[Knowledge is Nectar]
Department of Computer Engineering

Course - System Programming and Compiler Construction (SPCC)

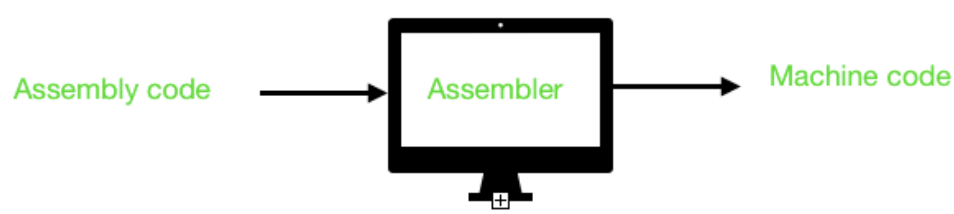
Aim	Design a two-pass assembler.
Objective	Implement a two-pass assembler to enhance assembly code translation, achieving robustness, efficiency, and error detection in programming.

Theory

Introduction of Assembler

Assembler is a program for converting instructions written in low-level assembly code into relocatable machine code and generating along information for the loader.

It is necessary to convert user written programs into a machinery code. This is called as translation of the high level language to low level that is machinery language. This type of translation is performed with the help of system software. Assembler can be defined as a program that translates an assembly language program into a machine language program. Self assembler is a program that runs on a computer and produces the machine codes for the same computer or same machine. It is also known as resident assembler. A cross assembler is an assembler which runs on a computer and produces the machine codes for other computer.



It generates instructions by evaluating the mnemonics (symbols) in operation field and find the value of symbol and literals to produce machine code. Now, if assembler do all this work in one scan then it is called single pass assembler, otherwise if it does in multiple scans then called multiple pass assembler. Here assembler divide these tasks in two passes:

- **Pass-1:**

1. Define symbols and literals and remember them in symbol table and literal table respectively.
2. Keep track of location counter
3. Process pseudo-operations
4. Defines program that assigns the memory addresses to the variables and translates the source code into machine code

- **Pass-2:**
  1. Generate object code by converting symbolic op-code into respective numeric op-code
  2. Generate data for literals and look for values of symbols
  3. Defines program which reads the source code two times
  4. It reads the source code and translates the code into object code.

Firstly, We will take a small assembly language program to understand the working in their respective passes. Assembly language statement format:

[Label] [Opcode] [operand]

Example: M ADD R1, ='3'
where, M - Label; ADD - symbolic opcode;
R1 - symbolic register operand; (='3') - Literal

Assembly Program:

```
Label  Op-code   operand   LC value(Location counter)
JOHN   START     200
       MOVER     R1, ='3'    200
       MOVEM     R1, X       201
L1     MOVER     R2, ='2'    202
       LTORG                 203
X      DS        1           204
       END                   205
```

Let's take a look on how this program is working:

1. **START:** This instruction starts the execution of program from location 200 and label with START provides name for the program.(JOHN is name for program)
2. **MOVER:** It moves the content of literal(='3′) into register operand R1.
3. **MOVEM:** It moves the content of register into memory operand(X).
4. **MOVER:** It again moves the content of literal(='2′) into register operand R2 and its label is specified as L1.
5. **LTORG:** It assigns address to literals(current LC value).
6. **DS(Data Space):** It assigns a data space of 1 to Symbol X.
7. **END:** It finishes the program execution.

Working of Pass-1:

Define Symbol and literal table with their addresses. Note: Literal address is specified by LTORG or END.

Step-1: START 200

(here no symbol or literal is found so both table would be empty)

Step-2: MOVER R1, ='3′ 200

(='3′ is a literal so literal table is made)

Literal	Address
='3′	– – –

Step-3: MOVEM R1, X 201

X is a symbol referred prior to its declaration so it is stored in symbol table with blank address field.

Symbol	Address
X	– – –

Step-4: L1 MOVER R2, ='2′ 202

L1 is a label and ='2′ is a literal so store them in respective tables

Symbol	Address
X	– – –
L1	202

Literal	Address
='3′	– – –
='2′	– – –

Step-5: LTORG 203

Assign address to first literal specified by LC value, i.e., 203

Literal	Address
='3′	203
='2′	– – –

Step-6: X DS 1 204

It is a data declaration statement i.e X is assigned data space of 1. But X is a symbol which was referred earlier in step 3 and defined in step 6.This condition is called Forward Reference Problem where variable is referred prior to its declaration and can be solved by back-patching. So now assembler will assign X the address specified by LC value of current step.

Symbol	Address
X	204
L1	202

Step-7: END 205

Program finishes execution and remaining literal will get address specified by LC value of END instruction. Here is the complete symbol and literal table made by pass 1 of assembler.

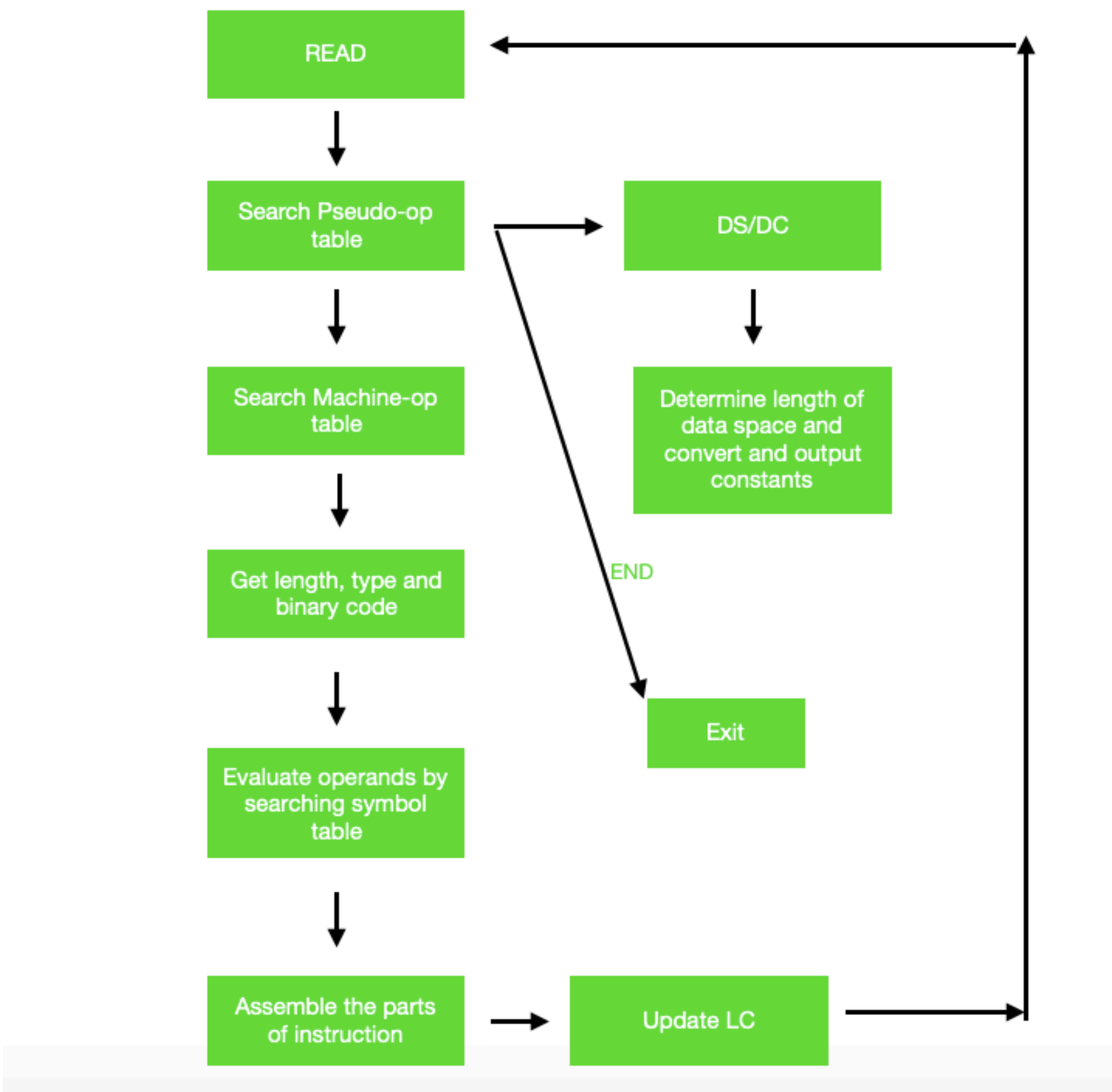
Symbol	Address
X	204
L1	202

Literal	Address
='3′	203
='2′	205

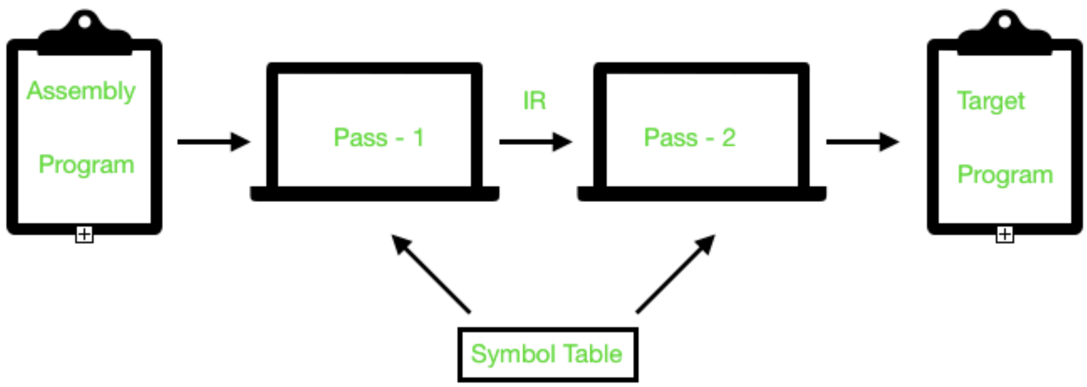
Now tables generated by pass 1 along with their LC value will go to pass-2 of assembler for further processing of pseudo-opcodes and machine op-codes.

Working of Pass-2:

	Pass-2 of assembler generates machine code by converting symbolic machine-opcodes into their respective bit configuration(machine understandable form). It stores all machine-opcodes in MOT table (op-code table) with symbolic code, their length and their bit configuration. It will also process pseudo-ops and will store them in POT table(pseudo-op table). Various Data bases required by pass-2: 1. MOT table(machine opcode table) 2. POT table(pseudo opcode table) 3. Base table(storing value of base register) 4. LC (location counter) Take a look at flowchart to understand:



As a whole assembler works as:

Implementation/ Code

```
import sys

def RemoveSpaces(x):
    if (x != " ") or (x != ", "):
        return x

def RemoveCommas(x):
    if x[-1] == ",":
        return x[ : len(x) - 1]
    else:
        return x

def CheckLiteral(element):
    if element[ : 2] == "='":
        return True
    else:
        return False

def CheckSymbol(Elements):
    global SymbolTable, Opcodes

    if (len(Elements) > 1) and ([Elements[-1], None, None, "Variable"] not in
SymbolTable) and (Elements[-1] != "CLA") and (Elements[-2] not in ["BRP", "BRN",
"BRZ"]) and (Elements[-1][ : 2] != "='") and (Elements[-1][ : 3] != "REG") and
(not Elements[-1].isnumeric()):
        return True
    else:
        return False

def CheckLabel(Elements):
```

```
    global SymbolTable, Opcodes

    if (len(Elements) >= 2) and (Elements[1] in Opcodes):
        if Elements[0] not in SymbolTable:
            return True
    else:
        return False

Opcodes = ["CLA", "LAC", "SAC", "ADD", "SUB", "BRZ", "BRN", "BRP", "INP", "DSP",
"MUL", "DIV", "STP", "DATA", "START"]
AssemblyOpcodes = {"CLA" : "0000", "LAC" : "0001", "SAC" : "0010", "ADD" : "0011",
"SUB" : "0100", "BRZ" : "0101","BRN" : "0110",
                   "BRP" : "0111", "INP" : "1000", "DSP" : "1001", "MUL" : "1010",
"DIV" : "1011", "STP" : "1100"}
SymbolTable = []
LiteralTable = []
Variables = []
Declarations = []
AssemblyCode = []
location_counter = 0
stop_found = False
end_found = False

file = open("/Users/pranaysinghvi/Library/CloudStorage/OneDrive-Personal/SPIT
College/3)Class/Semester 6/5)SPCC/1)Experiment/8_/Assembly Code Input.txt", "rt")

# ERROR 1 : Checking for missing START statement
for line in file:
    # Checking for comments
    if line[ : 2] != "//":
        if line.strip() != "START":
            print("STARTError : 'START' statement is missing. " + "( Line " +
str(location_counter) + " )")
            sys.exit(0)
        else:
            file.seek(0, 0)
            break

# First Pass
for line in file:
    # Checking for comments
    if line[ : 2] != "//":
        Elements = line.strip().split(" ")
        Elements = list(filter(RemoveSpaces, Elements))
        Elements = list(map(RemoveCommas, Elements))

        # Removing comments
```

```
        for i in range(len(Elements)):
            if Elements[i][ : 2] == "//":
                Elements = Elements[ : i]
                break

        # ERROR 2 : Checking for too many operands
        # If the instruction doesn't contain a Label
        if (len(Elements) >= 3) and (Elements[0] in Opcodes):
            print("TooManyOperandsError : Too many operands used for the '" +
Elements[0] + "' assembly opcode. " + "( Line " + str(location_counter) + " )")
            sys.exit(0)
        # If the instruction contains a Label
        elif (len(Elements) >= 4) and (Elements[1] in Opcodes):
            print("TooManyOperandsError : Too many operands used for the '" +
Elements[1] + "' assembly opcode. " + "( Line " + str(location_counter) + " )")
            sys.exit(0)

        # ERROR 3 : Checking for less operands
        # If the instruction doesn't contain a Label
        if (len(Elements) == 1) and (Elements[0] in ["LAC", "SAC", "ADD", "SUB",
"BRZ", "BRN", "BRP", "INP", "DSP", "MUL", "DIV"]):
            print("LessOperandsError : Less operands used for the '" + Elements[0]
+ "' assembly opcode. " + "( Line " + str(location_counter) + " )")
            sys.exit(0)
        # If the instruction contains a Label
        elif (len(Elements) == 2) and (Elements[1] in ["LAC", "SAC", "ADD", "SUB",
"BRZ", "BRN", "BRP", "INP", "DSP", "MUL", "DIV"]):
            print("LessOperandsError : Less operands used for the '" + Elements[1]
+ "' assembly opcode. " + "( Line " + str(location_counter) + " )")
            sys.exit(0)

        # ERROR 4 : Checking for invalid opcodes
        if stop_found is False:
            if len(Elements) == 3:
                # If the instruction contains a Label
                if Elements[1] not in Opcodes:
                    print("InvalidOpcodeError : '" + Elements[1] + "' is an
invalid opcode. " + "( Line " + str(location_counter) + " )")
                    sys.exit(0)
            if (len(Elements) == 2) and (Elements[1] == "CLA"):
                pass
            elif len(Elements) == 2:
                # If the instruction doesn't contain a Label
                if Elements[0] not in Opcodes:
                    print("InvalidOpcodeError : '" + Elements[0] + "' is an
invalid opcode. " + "( Line " + str(location_counter) + " )")
                    sys.exit(0)
```

```
        # Check for STP
        if (len(Elements) == 3) and (Elements[1] == "DATA"):
            stop_found = True

        # Check for END
        if (len(Elements) == 1) and (Elements[0] == "END"):
            end_found = True

            for i in range(len(LiteralTable)):
                if LiteralTable[i][1] == -1:
                    LiteralTable[i][1] = location_counter
                    location_counter += 1

            break

        if not stop_found:
            # Check for Literal
            for x in Elements:
                if CheckLiteral(x):
                    LiteralTable.append([x, -1])

            # Check for Labels
            if CheckLabel(Elements):
                SymbolTable.append([Elements[0], location_counter, None, "Label"])

            # Check for Symbols
            if CheckSymbol(Elements):
                SymbolTable.append([Elements[-1], None, None, "Variable"])
        elif stop_found:
            if (Elements[0] != "STP") and (Elements[0] != "END"):
                # ERROR 5 : Checking for multiple definations
                if Elements[0] not in Variables:
                    Variables.append(Elements[0])
                    Declarations.append((Elements[0], Elements[2]))
                else:
                    print("DefinationError : Variable '" + Elements[0] + "'
defined multiple times. " + "( Line " + str(location_counter) + " )")
                    sys.exit(0)

                # ERROR 6 : Checking for redundant declarations
                if [Elements[0], None, None, "Variable"] not in SymbolTable:
                    print("RedundantDeclarationError : " + Elements[0] + "
declared but not used.")
                    sys.exit(0)
                location = SymbolTable.index([Elements[0], None, None,
"Variable"])
```

```
                SymbolTable[location][1] = location_counter
                SymbolTable[location][2] = Elements[2]

        location_counter += 1

# ERROR 7 : Checking for missing END statement
if end_found is False:
    print("ENDError : 'END' statement is missing." + "( Line " +
str(location_counter) + " )")
    sys.exit(0)

# ERROR 8 : Checking for undefined variables
for x in SymbolTable:
    if x[1] is None and x[3] == "Variable":
        print("UndefinedVariableError : Variable '" + x[0] + "' not defined.")
        sys.exit(0)

# Printing Tables after First Pass
print(">>> Opcode Table <<<\n")
print("ASSEMBLY OPCODE     OPCODE")
print("--------------------------")

for key in AssemblyOpcodes:
    print(key.ljust(20) + AssemblyOpcodes[key].ljust(6))

print("--------------------------")
print("\n>>> Literal Table <<<\n")
print("LITERAL     ADDRESS")
print("-------------------")

for i in LiteralTable:
    print(i[0].ljust(12) + str(i[1]).ljust(7))

print("-------------------")
print("\n>>> Symbol Table <<<\n")
print("SYMBOL          ADDRESS     VALUE     TYPE")
print("----------------------------------------------")

for i in SymbolTable:
    print(i[0].ljust(16) + str(i[1]).ljust(12) + str(i[2]).ljust(10) +
i[3].ljust(10))

print("----------------------------------------------")
print("\n>>> Data Table <<<\n")
print("VARIABLES     VALUE")
print("-------------------")
```

```python
for i in Declarations:
    print(i[0].ljust(14) + str(i[1]).ljust(10))

print("------------------\n")

# Second Pass
file.seek(0, 0)

print(">>> MACHINE CODE <<<\n")

for line in file:
    # Checking for comments
    if line[ : 2] != "//":
        Elements = line.strip().split(" ")
        Elements = list(filter(RemoveSpaces, Elements))
        Elements = list(map(RemoveCommas, Elements))
        s = ""

        # Removing comments
        for i in range(len(Elements)):
            if Elements[i][ : 2] == "//":
                Elements = Elements[ : i]
                break

        # To terminate machine code conversion
        if (len(Elements) == 3) and (Elements[1] == "DATA"):
            break

        if Elements[0] == "STP":
            AssemblyCode.append("00 "+ AssemblyOpcodes["STP"] + " 00 00 00")
            print("00 " + AssemblyOpcodes["STP"] + " 00 00 00")
        # If the CLA opcode has a Label before it
        elif (len(Elements) == 2) and (Elements[1] == "CLA"):
            for i in range(len(SymbolTable)):
                if Elements[0] == SymbolTable[i][0]:
                    AssemblyCode.append(str(SymbolTable[i][1]).rjust(2, "0") + " "
+ AssemblyOpcodes["CLA"] + " 00 00 00")
                    print(str(SymbolTable[i][1]).rjust(2, "0") + " "+
AssemblyOpcodes["CLA"] + " 00 00 00")
        elif Elements[0] != "START":
            if (len(Elements) == 1) and (Elements[0] == "CLA"):
                AssemblyCode.append("00 " + AssemblyOpcodes["CLA"] + " 00 00 00")
                print("00 " + AssemblyOpcodes["CLA"] + " 00 00 00")
            # If there is no Label
            elif (len(Elements) == 2) and (Elements[1] != "CLA"):
                print("00 " + AssemblyOpcodes[Elements[0]], end = " ")
                s = "00 " + AssemblyOpcodes[Elements[0]] + " "
```

```
                # Dealing with Literals
                if CheckLiteral(Elements[1]):
                    for i in range(len(LiteralTable)):
                        if LiteralTable[i][0] == Elements[1]:
                            AssemblyCode.append(s + "00 00 " +
str(LiteralTable[i][1]).rjust(2, "0"))
                            print("00 00 " + str(LiteralTable[i][1]).rjust(2,
"0"))
                # Dealing with Lables (BRP, BRZ, BRN)
                elif Elements[0] in ["BRP", "BRN", "BRZ"]:
                    for i in range(len(SymbolTable)):
                        if SymbolTable[i][0] == Elements[1]:
                            AssemblyCode.append(s +
str(SymbolTable[i][1]).rjust(2, "0") + " 00 00")
                            print(str(SymbolTable[i][1]).rjust(2, "0") + " 00 00")
                # Dealing with Registers
                elif Elements[1][ : 3] == "REG":
                    AssemblyCode.append(s + "00 " + Elements[1][-1].rjust(2, "0")
+ " 00")
                    print("00 " + Elements[1][-1].rjust(2, "0") + " 00")
                # Dealing with Variables
                else:
                    for i in range(len(SymbolTable)):
                        if SymbolTable[i][0] == Elements[1]:
                            AssemblyCode.append(s + "00 00 " +
str(SymbolTable[i][1]).rjust(2, "0"))
                            print("00 00 " + str(SymbolTable[i][1]).rjust(2, "0"))
            # If the instruction conatins a Label
            elif len(Elements) == 3:
                for i in range(len(SymbolTable)):
                        if SymbolTable[i][0] == Elements[0]:
                            print(str(SymbolTable[i][1]).rjust(2, "0") + " " +
AssemblyOpcodes[Elements[1]], end = " ")
                            s = str(SymbolTable[i][1]).rjust(2, "0") + " " +
AssemblyOpcodes[Elements[1]] + " "

                # Dealing with Literals
                if CheckLiteral(Elements[2]):
                    for i in range(len(LiteralTable)):
                        if LiteralTable[i][0] == Elements[2]:
                            AssemblyCode.append(s + "00 00 " +
str(LiteralTable[i][1]).rjust(2, "0"))
                            print("00 00 " + str(LiteralTable[i][1]).rjust(2,
"0"))
                # Dealing with Lables (BRP, BRZ, BRN)
                elif Elements[1] in ["BRP", "BRN", "BRZ"]:
```

```
                    for i in range(len(SymbolTable)):
                        if SymbolTable[i][0] == Elements[2]:
                            AssemblyCode.append(s +
str(SymbolTable[i][1]).rjust(2, "0") + " 00 00")
                            print(str(SymbolTable[i][1]).rjust(2, "0") + " 00 00")
                # Dealing with Registers
                elif Elements[2][ : 3] == "REG":
                    AssemblyCode.append(s + "00 " + Elements[2][-1].rjust(2, "0")
+ " 00")
                    print("00 " + Elements[2][-1].rjust(2, "0") + " 00")
                # Dealing with Variables
                else:
                    for i in range(len(SymbolTable)):
                        if SymbolTable[i][0] == Elements[2]:
                            AssemblyCode.append(s + "00 00 " +
str(SymbolTable[i][1]).rjust(2, "0"))
                            print("00 00 " + str(SymbolTable[i][1]).rjust(2, "0"))

file.close()

file = open("./Machine Code.txt", "x")

file.write("------------\nMACHINE CODE\n------------\n\n")

for x in AssemblyCode:
    file.write(x + "\n")

file.close()
```

Output

```
pranaysinghvi@Pranays-MacBook-Air 5)SPCC % /opt/homebrew/bin/pyt
ersonal/SPIT College/3)Class/Semester 6/5)SPCC/1)Experiment/8_/E
>>> Opcode Table <<<

ASSEMBLY OPCODE     OPCODE
--------------------------
CLA                 0000
LAC                 0001
SAC                 0010
ADD                 0011
SUB                 0100
BRZ                 0101
BRN                 0110
BRP                 0111
INP                 1000
DSP                 1001
MUL                 1010
DIV                 1011
STP                 1100
--------------------------

>>> Literal Table <<<

LITERAL     ADDRESS
-------------------
='1'        28
='35'       29
='5'        30
='600'      31
-------------------

>>> Symbol Table <<<

SYMBOL          ADDRESS     VALUE     TYPE
----------------------------------------------
LoopOne         1           None      Label
X               27          0         Variable
A               22          250       Variable
Loop            5           None      Label
Subtraction     6           None      Label
B               23          125       Variable
C               24          90        Variable
D               25          88        Variable
Division        12          None      Label
E               26          5         Variable
Zero            16          None      Label
Positive        19          None      Label
----------------------------------------------
```

```
>>> Data Table <<<

VARIABLES     VALUE
-------------------
A             250
B             125
C             90
D             88
E             5
X             0
-------------------


>>> MACHINE CODE <<<

01 0000 00 00 27
00 0001 00 00 22
00 0011 00 00 28
00 0100 00 00 29
05 0111 06 00 00
06 0100 00 00 30
00 0011 00 00 23
00 1010 00 00 24
00 0100 00 00 25
00 1010 00 00 31
00 0101 12 1011 00 00 26
00 0000 00 00 00
00 0001 00 01 00
00 0111 19 00 00
16 0010 00 00 27
00 1001 00 00 27
00 1100 00 00 00
19 0000 00 00 00
00 1001 00 01 00
00 1001 00 02 00
pranaysinghvi@Pranays-MacBoo
```

Conclusion	In conclusion, I successfully developed a two-pass assembler, ensuring accurate conversion of assembly code to machine code with improved efficiency and error detection capabilities, enhancing programming reliability and productivity.
References	[1] Geeksforgeeks (25 Sep, 2023) Introduction of Assembler https://www.geeksforgeeks.org/introduction-of-assembler/