Experiment 06 : Two Pass Assembler

Learning Objective: Student should be able to Apply 2 pass Assembler for X86 machine.

Tools: Jdk1.8,Turbo C/C++, Python, Notepad++

Theory: An assembler performs the following functions

1 Generate instructions

a. Evaluate the mnemonic in the operator field to produce its machine code.
b. Evaluate subfields- find value of each symbol, process literals & assign address.

2 Process pseudo ops: we can group these tables into passed or sequential scans over input associated with each task are one or more assembler modules.

Format of Databases:

a) POT (Pseudo Op-code Table):-

POT is a fixed length table i.e. the contents of these table are altered during the assembly process.

Pseudo Op-code (5 Bytes character)	Address of routine to process pseudo-op-code. (3 bytes= 24 bit address)
"DROPb"	P1 DROP
"ENDbb"	P1 END
"EQUbb"	P1 EQU
"START"	P1 START
"USING"	P1 USING

- The table will actually contain the physical addresses.
- POT is a predefined table.
- In PASS1 , POT is consulted to process some pseudo opcodes like-DS,DC,EQU
- In PASS2, POT is consulted to process some pseudo opcodes like DS,DC,USING,DROP

b) MOT (Mnemonic Op-code Table):-

MOT is a fixed length table i.e. the contents of these tables are altered during the assembly process.

Mnemonic Op-code (4 Bytes character)	Binary Op-code (1 Byte Hexadecimal)	Instruction Length (2 Bits binary)	Instruction Format (3 bits binary)	Not used in this design (3 bits)
"Abbb"	5A	10	001	
"AHbb"	4A	10	001	
"ALbb"	5E	10	001	
"ALRb"	1E	01	000	

b- Represents the char blanks.

Codes:-

Instruction Length
01= 1 Half word=2 Bytes
10= 2 Half word=4 Bytes

Instruction Format
000 = RR
001 = RX

11= 3 Half word=6 Bytes

010 = RS
011= SI
100= SS

- MOT is a predefined table.
- In PASS1 , MOT is consulted to obtain the instruction length.(to Update LC)
- In PASS2, MOT is consulted to obtain:
    a) Binary Op-code (to generate instruction)
    b) Instruction length (to update LC)
    c) Instruction Format (to assemble the instruction).

C) Symbol table (ST):

Symbol (8 Bytes charaters)	Value (4 Bytes Hexadecimal)	Length (1 Byte Hexadecimal)	Relocation (R/A) (1 Byte character)
"PRG1bbb"	0000	01	R
"FOURbbbb"	000C	04	R

- ST is used to keep a track on the symbol defined in the program.
- In pass1- whenever the symbol is defined an entry is made in the ST.
- In pass2- Symbol table is used to generate the address of the symbol.

D) Literal Table (LT):

Literal	Value	Length	Relocation (R/A)
= F '5'	28	04	R

- LT is used to keep a track on the Literals encountered in the program.
- In pass1- whenever the literals are encountered an entry is made in the LT.
- In pass2- Literal table is used to generate the address of the literal.

E) Base Table (BT):

Register Availability (1 Byte Character)	Contents of Base register (3 bytes= 24 bit address hexadecimal)
1 'N'	-
2 'N'	-
. . .	-
15 'N'	00

- Code availability-
- Y- Register specified in USING pseudo-opcode.
- N--Register never specified in USING pseudo-opcode.
- BT is used to keep a track on the Register availability.
- In pass1- BT is not used.

- In pass2- In pass2, BT is consulted to find which register can be used as base registers along with their contents.

Pass 1: Purpose - To define symbols & literals

1) Determine length of machine instruction (MOTGET)
2) Keep track of location counter (LC)
3) Remember values of symbols until pass2 (STSTO)
4) Process some pseudo ops. EQU
5) Remember literals (LITSTO)

Pass 2: Purpose - To generate object program

1) Look up value of symbols (STGET)
2) Generate instruction (MOTGET2)
3) Generate data (for DC, DS)
4) Process pseudo ops. (POT, GET2)

Data Structures:

Pass 1: Database

1) Input source program
2) Location counter (LC) to keep the track of each instruction location
3) MOT (Machine OP table that gives mnemonic & length of instruction
4) POT (Pseudo op table) which indicate mnemonic and action to be taken for each pseudo-op
5) Literals table that is used to store each literals and its location
6) A copy of input to be used later by pass-2.

Pass 2: Database

1) Copy of source program from Pass1
2) Location counter
3) MOT which gives the length, mnemonic format op-code
4) POT which gives mnemonic & action to be taken
5) Symbol table from Pass1
6) Base table which indicates the register to be used or base register
7) A work space INST to hold the instruction & its parts
8) A work space PRINT LINE, to produce printed listing
9) A work space PUNCH CARD for converting instruction into format needed by loader
10) An output deck of assembled instructions needed by loader.

Algorithm:

Pass 1

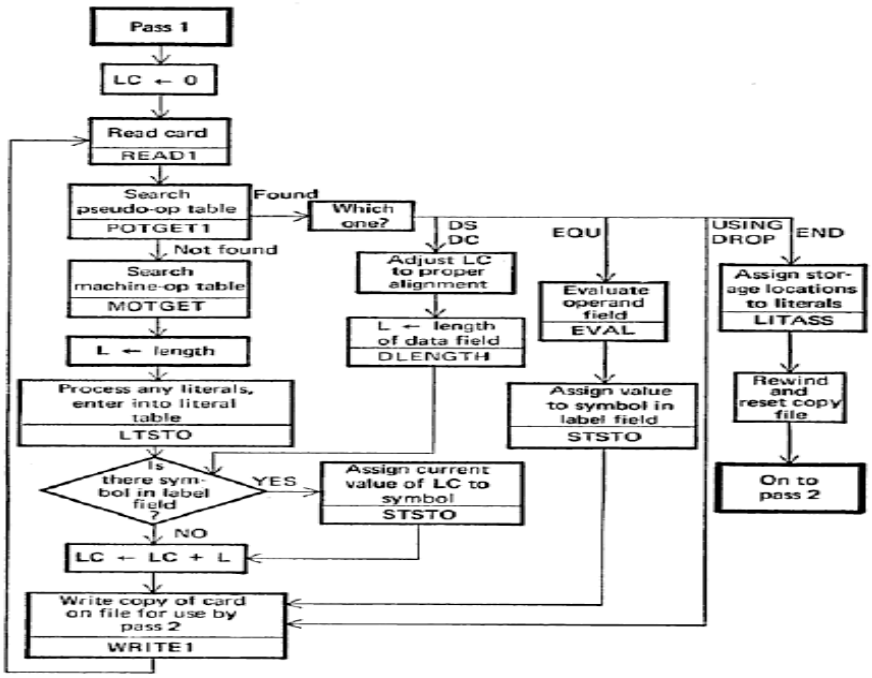
1. Initialize LC to 0
2. Read instruction
3. Search for pseudo-op table and process it.
    a. If its a USING & DROP pseudo-op then pass it to pass2 assembler
    b. If its a DS & DC then Adjust LC and increment LC by L
    c. If its EQU then evaluate the operand field and add value of symbol in symbol table
    d. If its END then generates Literal Table and terminate pass1
4. Search for machine op table
5. Determine length of instruction from MOT
6. Process any literals and enter into literal table

7. Check for symbol in label field
    a. If yes assign current value of LC to Symbol in ST and increment LC by length
    b. If no increment LC by length
8. Write instruction to file for pass 2
9. Go to statement 2

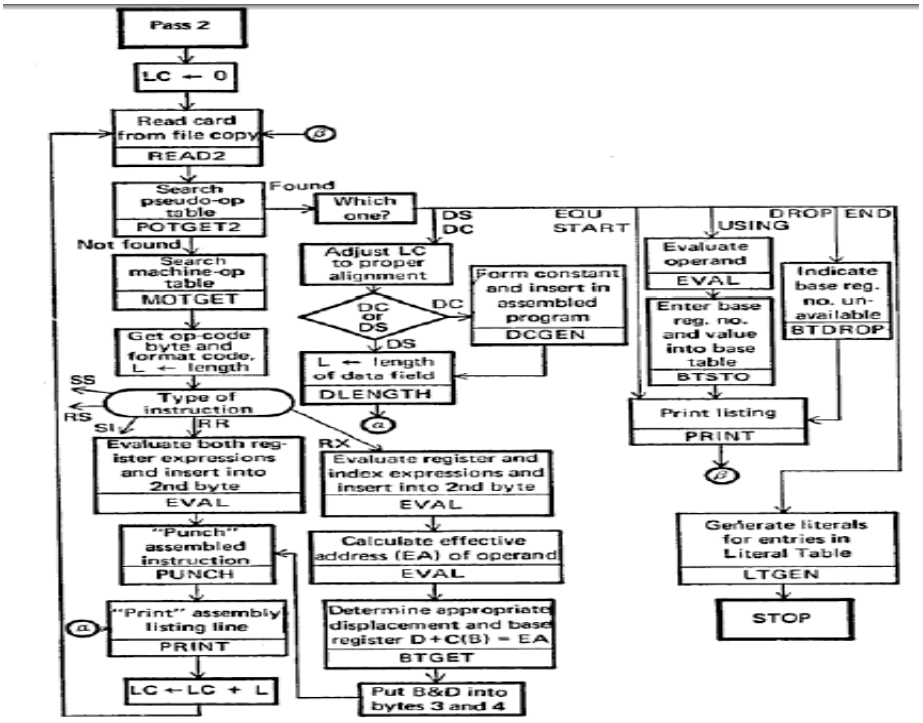
Pass 2

1. Initialize LC to 0.
2. Read instruction
3. Search for pseudo-op table and process it.
    a. If it's a USING then check for base register number and find the contents of the base register
    b. If it's a DROP then base register is not available
    c. If it's a DS then find length of data field and print it
    d. If DC then form constant and insert into machine code.
    e. If its EQU and START then print listing
    f. If its END then generates Literal Table and terminate pass1
    g. Generate literals for entries in literal table
    h. stop
4. Search for machine op table
5. Get op-code byte and format code
6. Set L = length
7. Check for type of instruction
    a. evaluate all operands and insert into second byte
    b. increment LC by length
    c. print listing
    d. Write instruction to file
8. Go to step 2

Flowchart:Pass1



Flowchart: Pass 2



Design:

```
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("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("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")
                                      else:
                                   for i in
range(len(SymbolTable)):
                                        if
SymbolTable[i][0]                       ==
AssemblyCode.append(s + "00 00 " +
str(SymbolTable[i][1]).rjust(2,
"0"))

file.close
```

Output:

>>> Literal Table <<<

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

>>> Data Table <<<

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

>>> 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

>>> 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
```

Result and Discussion: A two-pass assembler is a type of assembler that processes the source code in two passes. In the first pass, it reads the entire source code, generates a symbol table, and performs some initial processing like detecting labels and assigning addresses. In the second pass, it translates the instructions into machine code using the information gathered in the first pass.

Learning Outcomes: The student should have the ability to

LO1: ***Describe*** the different database formats of 2-pass Assembler with the help of examples.
LO2: **Design** 2 pass Assembler for X86 machine.
LO3: ***Develop***2-pass Assembler for X86 machine.
LO4: Illustrate the working of 2-Pass Assembler.

Course Outcomes: Upon completion of the course students will be able to Describe the various data structures and passes of assembler design.

Conclusion:

For Faculty Use

Correction Parameters	Formative Assessment [40%]	Timely completion of Practical [40%]	Attendance / Learning Attitude [20%]	
Marks Obtained				