Jadavpur University

Department of Electronics and Tele-Communication Engineering

Faculty of Engineering and Technology

System Software Lab

UG-IV Semester-I

Submitted By – OUMI MANDAL

Roll - 002210701135

Group – G2

DAY-4

Q. 1. Implement a one-pass assembler for a simple assembly language without forward references.

Q. 2.Implement Pass-1 of a two-pass assembler to:
- Build the symbol table.
- Generate the intermediate code with addresses for labels.

Q. 3.Implement Pass-2 of a two-pass assembler to:
- Convert intermediate code into actual machine code.
- Replace label references with their addresses.

A.

```
import struct

import io


"""
Using SIC instruction set since it is fixed-length and sufficiently small.
"""


OPTAB: dict = { "ADD" :
            0x18,
      "AND" : 0x40,
  "COMP" : 0x28,
  "DIV" : 0x24,
  "J" : 0x3C,
  "JEQ" : 0x30,
  "JGT" : 0x34,
  "JLT" : 0x38,
  "JSUB" : 0x48,
  "LDA" :         0x00,
  "LDCH" : 0x60,
  "LDL" : 0x08,
  "LDX" : 0x04,
  "MUL" : 0x20,
  "OR" : 0x44,
  "RD" : 0xD8,
  "RSUB" : 0x4C,
  "STA" : 0x0C,
  "STCH" : 0x54
```

```
  "STL" : 0x14,
  "STSW" : 0xE8,
  "STX" : 0x10,
  "SUB" : 0x1C,
  "TD" : 0xE0,
  "TIX" : 0x2C,
  "WD" : 0xDC
}

POT: list = [ "RESB",
  "RESW",
  "BYTE",
  "WORD"
]

                def ceil_half(x: int):
            return (x>>1)+(x&0b1)

# Q1. Simple one-pass assembler class
Assembler:
  def __init__(self, file: str):
    self.filename: str = file
    self.optab: dict[str, int] = OPTAB self.pot: list = POT

  def parse_file(self) -> int:
    """
    Assembles given assembly file and generates out file.
    Returns number of lines written. If error is raised, returns 0 """
    try:
      with open(self.filename.replace(".asm", ".out"), 'wb') as outfile: with open(self.filename) as infile:
          count = 0
          for line in infile.readlines():
            if (not line.strip()) or line.lstrip().startswith(';'): continue
            line = line.replace(',', '').lstrip()
```

```
            line = line.split() # line[0] holds opcode and line[1] holds address now, if it exists
            machinecode = self.optab.get(line[0].upper()) if machinecode is
            None:
              if line[0].upper() in self.pot:
                continue
              raise Exception(f"Invalid opcode found: {line[0]}, quitting.") if (len(line) < 2):
              if (machinecode == 0x4C):
                count += 1
                outfile.write(struct.pack(">BBB", 0x4C, 0x00, 0x00)) continue
            machinecode = (machinecode << 16) | (int(line[1].replace("#", ''), 0)&0x7FFF)
#                                                 ^                                                 ^
#                                               opcode                                          operand
            if(len(line)>2 and line[2].lower()=='x'):
              machinecode = machinecode | (1<<15)
            outfile.write(
              struct.pack(
                ">BBB",          # 3-byte long instruction
                (machinecode>>16)&0xFF,       #       upper       byte
                (machinecode>>8)&0xFF,       #       middle       byte
                machinecode&0xFF # lower byte
              )
            )
            count += 1 return
        count
      except OSError:
        print("Could not open the file.") return 0

class TwoPass(Assembler):
  def __init__(self, file: str):
    super().__init__(file)
    self.intermediate: io.StringIO = io.StringIO() self.symtab: dict[str, int] = {}
    self.length: int = 0
```

```
  def parse_file(self) -> None:
    raise NotImplementedError("parse_file is not implemented for child.
Use method pass1, followed by pass2 instead")

  # Q2. First pass of two-pass assembler def pass1(self) ->
  None:
    try:
      LOCCTR = 0
      begin = LOCCTR
      with open(self.filename) as infile:
        for line in infile.readlines():
          line = line.replace(',', ' ').lstrip()
          if line.lstrip().startswith(';') or (not line.strip()): continue

          line = line.split()

          if line[0].lower() == 'start' or (len(line)==3 and line[1].lower()=='start'):
            LOCCTR = int(line[1], 0) begin =
            LOCCTR
            continue

          if len(line) > 2:
            if line[0].replace(':', '') in self.symtab: raise Exception("Duplicate label
              found.")
            self.symtab[line[0].replace(':', '')] = LOCCTR line = line[1:]



          if line[0].upper() in self.optab:
            LOCCTR += 3

          elif line[0].upper() == 'RESW':
            LOCCTR += 3*int(line[1].replace('#', ''), 0)

          elif line[0].upper() == 'RESB':
            LOCCTR += int(line[1].replace('#', ''), 0)
```

```
        elif line[0].upper() == 'WORD':
          LOCCTR += 3

        elif line[0].upper() == 'BYTE':
          if line[1].upper().startswith('X'):
            l = line[1].replace("'", "").replace('X', '').replace('x', '') LOCCTR += ceil_half(len(l))
          elif line[1].upper().startswith('C'):
            l = line[1].replace("'", "").replace('c', '').replace('C', '').replace('\\', '')
            LOCCTR += len(l)
          else:
            raise Exception("Invalid format for BYTE")

        else:
          raise Exception("Unknown Directive")

        self.intermediate.write(' '.join(line) + '\n') self.length = LOCCTR - begin
      self.intermediate.seek(0)

  except OSError:
    print("Could not open the file.")


#Q3. Second pass of two-pass assembler def pass2(self) ->
None:
  try:
    with open(self.filename.replace('.asm', '.out'), 'wb') as outfile: for line in self.intermediate.readlines():

        if (not line.strip()) or line.lstrip().startswith(';'): continue
        line = line.lstrip().split()
        if line[0].upper() == "START":
          continue
        machinecode = self.optab.get(line[0].upper())
```

```
        if machinecode is None:
          if line[0].upper() == "BYTE":
            operand = line[1]
            if operand.upper().startswith("C"):
              data = operand[2:-1].encode("ascii") outfile.write(data)
            elif operand.upper().startswith("X"):
              hexstr = operand[2:-1] if
              len(hexstr) % 2:
                hexstr = '0' + hexstr
              data = bytes.fromhex(hexstr) outfile.write(data)
            else:
              raise Exception("Invalid format for BYTE")

          elif line[0].upper() == "WORD":
            val = int(line[1], 0)
            outfile.write(struct.pack(">I", val)[1:])                    # take 3 low bytes (big-
endian)

          elif line[0].upper() == "RESB":
            outfile.write(b"\x00" * int(line[1], 0))

          elif line[0].upper() == "RESW":
            outfile.write(b"\x00" * (3 * int(line[1], 0)))

          else:
            raise Exception(f"Invalid opcode found: {line[0]}, quitting.") continue

        if (len(line) < 2):
          if (machinecode == 0x4C):
            outfile.write(struct.pack(">BBB", 0x4C, 0x00, 0x00)) continue
        if line[1] in self.symtab:
          addr = self.symtab[line[1]] else:
          addr = int(line[1], 0)
```

```
        machinecode = (machinecode << 16) | (addr & 0x7FFF) #        ^      ^
#                                                opcode                    operand label
        if(len(line)>2 and line[2].lower()=='x'):
          machinecode = machinecode | (0b1<<15)


        outfile.write( struct.pack(
            ">BBB",          # 3-byte long instruction
            (machinecode>>16)&0xFF, # upper byte
            (machinecode>>8)&0xFF, # middle byte
            machinecode&0xFF # lower byte
          )
        )

    except OSError:
      print("Could not create the output file.")
```