EMB0000243c3415

**시스템프로그래밍 2020 보고서**

**보고서 제출서약서**

**나는 숭실대학교 소프트웨어학부의 일원으로 명예를 지키면서 생활하고 있습니다.**

**나는 보고서를 작성하면서 다음과 같은 사항을 준수하였음을 엄숙히 서약합니다.**

**1. 나는 자력으로 보고서를 작성하였습니다.**

**1.1. 나는 동료의 보고서를 베끼지 않았습니다.**

**1.2. 나는 비공식적으로 얻은 해답/해설을 기초로 보고서를 작성하지 않았습니다.**

**2. 나는 보고서에서 참조한 문헌의 출처를 밝혔으며 표절하지 않았습니다. (나는 특히**

**인터넷에서 다운로드한 내용을 보고서에 거의 그대로 복사하여 사용하지 않았습니다.)**

**3. 나는 보고서를 제출하기 전에 동료에게 보여주지 않았습니다.**

**4. 나는 보고서의 내용을 조작하거나 날조하지 않았습니다.**

|  |  |
| --- | --- |
| **과목** | **시스템프로그래밍 2020** |
| **과제명** | **프로젝트 1c** |
| **담당교수** | **최 재 영 교 수** |
| **제출인** | ***소프트웨어학부 20150286 최유준* (출석번호 310번)** |
| **제출일** | **2020년 5월 13일** |

**------------------- 차 례 -------------------**

1. **동기/목적**
2. **설계/구현 아이디어**

**2.1 설계 아이디어**

**2.2 구현 아이디어**

1. **수행결과**
2. **결론 및 보충할 점**
3. **소스코드**

**1장 동기/목적**

이번 시스템 프로그래밍의 프로젝트는 한번 작성해 보았던 어셈블리 프로그램을 파이썬에서 다시 완성하는 것이었습니다. 내용은 똑같지만 파이썬에서 새로운 환경으로 프로젝트를 함으로써 이해도를 높이는 것이 목적입니다. JAVA에서 프로젝트를 진행했던 것과 방식이 매우 유사했고 파이썬 특유의 편리함에 프로젝트 진행이 원활하였고, List와 Dictionary를 이용한 파이썬 프로젝트 작성을 익히는 것이 목적입니다.

**2장 설계/구현 아이디어**

**2.1 설계 아이디어**

프로그램의 실행 순서는 다음과 같습니다.

1. assembler 생성자를 통해 inst.data read
2. loadInputFile 함수 호출
3. pass1 함수 호출
   1. putToken 함수 호출하여 Token에 값 입력
   2. address 계산 후 Token.location에 저장
   3. section별 symboltable, literaltable 저장
4. SymbolTable, LiteralTable 출력
5. pass2 함수 호출
   1. 각 section마다 makeObjectCode 함수 호출
   2. 각 토큰 별 ObjectCode를 만든 후 저장
   3. getObjectCode를 통해 codeList에 objectcode 저장
6. ObjectCode 출력

**\*모듈별 설계 아이디어**

**(1) loadInputFile**

input file을 읽어서 line별로 lineList에 저장한다.

**(2) pass1**

TokenList SymbolTable

symbol location

TokenTable1

TokenTable2

TokenTable3..

LiteralTable

literal location

instTable

TokenList에 new TokenTable을 생성하고 그 안에서 TokenTable과 new SymbolTable, new LiteralTable, instTable을 link한다.

1. **putToken**

line을 받아서 new Token에 넣고 그 값을 tokenList에 add한다..

1. **parsing**

line을 받아서 의미를 분석한 후 Token의 각 변수에 저장한다.

1. **printSymbolTable, printLiteralTable**

file에 각 section별 SymbolTable(LiteralTable)들을 symbolList(iteralList)와 locationList의 형태로 출력한다.

1. **pass2**

각 TokenTable별로 ObjectCode를 만들어준 후 codeList에 add한다

1. **makeObjectCode**

index를 입력받아 해당 tokenList의 값을 get하여 objectcode를 만든다

1. **setFlag**

flag와 value를 입력받아 해당 flag의 값을 value로 만든다.

1. **getObjectCode**

index를 입력받아 해당 index의 objectCode를 return한다.

1. **printObjectCode**

출력 형태에 맞게 codeList를 출력한다.

1. **modifySymbol, modifyLiteral**

symbol(literal)값과 변경할 주소값을 입력하여 해당 symbol(literal)의 location을 변경한다.

1. **search**

입력된 symbol(literal)의 location을 return한다.

1. **hash\_get**

operator를 입력하여 Instruction일 경우 그 format을 return한다.

**2.2 구현 아이디어**

1. loadInputFile

file을 open한 후 file의 마지막까지 readLine한 후 lineList에 add한다.

1. pass1

TokenList에 new TokenTable을 add하면서 new SymbolTable, new LiteralTable, instTable을 link한다. section별로 lineList에서 한 줄씩 읽어와 putToken한다. 각 Token의 operator를 분석하여 location 값을 저장한다.

1. parsing

line을 입력받아서 split 함수로 line을 label, operator, operand, comment로 분리한다. 각 요소를 변수에 저장하고 operand는 operand의 개수만큼 Stringp[]에 저장한다.

1. printSymbolTable, printLiteralTable

각 section별로 symbolList(literalList)의 size만큼 반복하여 해당 symtab(literaltab)의 symbolList(literalList)와 locationList를 출력한다.

1. pass2

각 section별로 makeObjectCode를 실행한 후 codeList에 add한다.

1. makeObjectCode

hash\_get함수를 통해 해당 index의 Token이 Instruction인지 판별한다. Instruction일 경우 각 Token의 format을 판별한다. 각 Token의 operand를 분석하여 #의 경우 immediate, @의 경우 indirect 등 setFlag해준다. 또한 같은 section의 symbol인지, 외부참조인지 등을 판별하여 address를 만든 후 opcode, nixbpe와 연산을 통해 objectcode를 완성한다.

1. setFlag

처음에 nixbpe를 parsing할 때 0으로 초기화하였다. 이 함수에서 value가 1일 경우 nixbpe에 flag를 더해주고 value가 0일 경우 빼준다.

1. printObjectCode

각 section별로 operator와 operand를 분석하여 H, D, R을 file에 출력한다. 그 후 txtlen 변수와 byteSize를 통해 각 T의 길이를 계산하여 txtlen에 저장한 후 T를 출력한다. 해당 token의 operand가 외부 참조 label을 사용하였을 경우 Modify를 출력한다. 이 과정을 TokenList의 size만큼 반복한다.

1. modifySymbol, modifyLiteral

iterator를 이용하여 hasNext를 조건으로 while문 반복한다. 입력한 symbol(literal)이 symbolList(literalList)와 equal할 경우 location을 newLocation으로 수정한다.

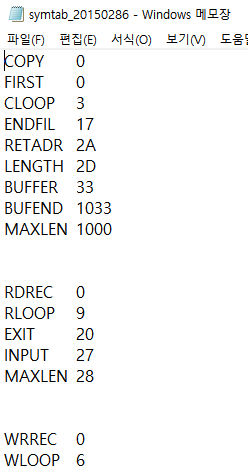
1. search

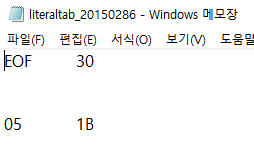
iterator를 이용하여 hasNext를 조건으로 while문 반복한다. 입력한 symbol(literal)이 symbolList(literalList)와 equal할 경우 location을 return한다.

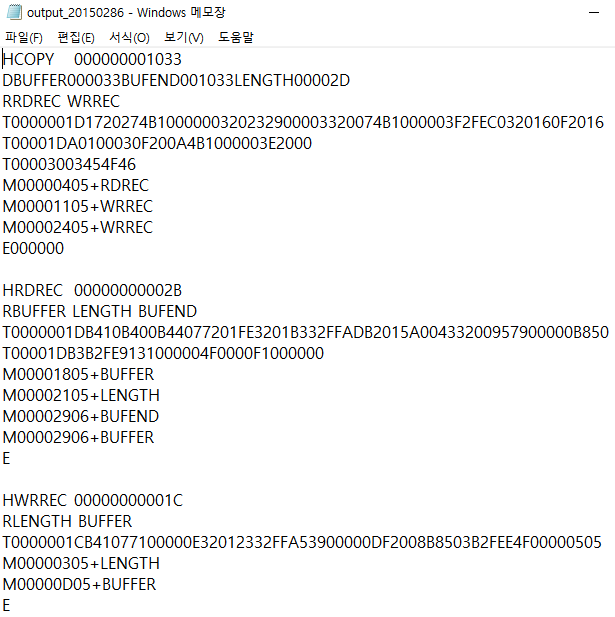
1. hash\_get

line을 입력받으면 해당 line이 4형식일 경우 +를 제외한 string으로 바꾼다. line이 Instruction일 경우 해당 Instruction의 format을 return한다. Instruction이 아니면 -1을 return한다.

**3장 수행결과**







file output

**4장 결론 및 보충할 점**

자바에서 파이썬으로 변환하는 과정인 만큼 간단하고 편리한 부분이 많았습니다. 그러나 처음에 시작할 때 파이썬에서 클래스를 생성하는 과정에서 혼돈이 많았기 때문에 클래스 생성 및 초기화에 대한 이해를 할 수 있도록 노력하였습니다.

* **DEBUGING**

스크린샷, 검은색, 모니터, 디스플레이이(가) 표시된 사진

자동 생성된 설명

**5장 소스코드**

**Assembler.py**

from InstTable import InstTable  
from LiteralTable import LiteralTable  
from SymbolTable import SymbolTable  
from TokenTable import TokenTable  
  
class Assembler:  
 def \_\_init\_\_(self, filename):  
 self.instTable = InstTable(filename)  
 self.lineList = []  
 self.symtabList = []  
 self. literaltabList = []  
 self.TokenList = []  
 self.codeList = []  
  
 def loadInputFile(self, inputFile):  
 f = open(inputFile, 'r')  
 i = 0  
 while True:  
 line = f.readline()  
 if not line:  
 break  
 self.lineList.append(line[:-1])  
 f.close()  
  
 def pass1(self):  
 i = 0  
 j = 0  
 start = 0  
 addr = 0  
 lit = ""  
 self.TokenList.append(TokenTable(SymbolTable(), LiteralTable(), self.instTable))#section 1  
 while i < len(self.lineList):#token parsing  
 self.TokenList[0].putToken(self.lineList[i])  
 self.TokenList[0].tokenList[i].location = addr  
 if i > 1:  
 if self.TokenList[0].tokenList[i].location - self.TokenList[0].tokenList[i-1].location >= 0:  
 self.TokenList[0].Listlength = self.TokenList[0].tokenList[i].location  
 #location find  
 if self.instTable.hash\_get(self.TokenList[0].tokenList[i].operator) > 0:#Instruction  
 if self.instTable.hash\_get(self.TokenList[0].tokenList[i].operator) == 3:  
 if self.TokenList[0].tokenList[i].operator[0] == '+':#format 4  
 addr += 4  
 else:#format 3  
 addr += 3  
 elif self.instTable.hash\_get(self.TokenList[0].tokenList[i].operator) == 2:#format2  
 addr += 2  
 else:#not Instruction  
 if self.TokenList[0].tokenList[i].operator == "RESW":  
 tmpadr = int(self.TokenList[0].tokenList[i].operand[0])  
 addr += 3\*tmpadr  
 elif self.TokenList[0].tokenList[i].operator == "RESB":  
 tmpadr =int( self.TokenList[0].tokenList[i].operand[0])  
 addr += tmpadr  
 elif self.TokenList[0].tokenList[i].operator == "BYTE":  
 addr += 1  
 elif self.TokenList[0].tokenList[i].operator == "WORD":  
 addr += 3  
 elif self.TokenList[0].tokenList[i].operator == "EQU":#equation 처리  
 if self.TokenList[0].tokenList[i].operand[0] != "\*":  
 arr = self.TokenList[0].tokenList[i].operand[0].split('-')#ex) BUFEND-BUFFER  
 tmpadr1 = 0  
 tmpadr2 = 0  
 for k in range(len(self.TokenList[0].tokenList)):  
 if self.TokenList[0].tokenList[k].label == arr[0]:  
 tmpadr1 = self.TokenList[0].tokenList[k].location  
 if self.TokenList[0].tokenList[k].label == arr[1]:  
 tmpadr2 = self.TokenList[0].tokenList[k].location  
 self.TokenList[0].tokenList[i].location = tmpadr1-tmpadr2  
  
 elif self.TokenList[0].tokenList[i].operator == "LTORG":  
 addr += 3  
  
 if self.TokenList[0].tokenList[i].operator == "CSECT":#next section stop  
 self.TokenList[0].tokenList.remove(self.TokenList[0].tokenList[i])  
 break  
  
 i += 1  
  
 for j in range(i):#literal, symbol input  
 if self.TokenList[0].tokenList[j].label != "":  
 self.TokenList[0].symTab.putSymbol(self.TokenList[0].tokenList[j].label, self.TokenList[0].tokenList[j].location)  
 if len(self.TokenList[0].tokenList[j].operand) == 0:  
 if self.TokenList[0].tokenList[j].operator == "LTORG":  
 self.TokenList[0].literalTab.putLiteral(lit, self.TokenList[0].tokenList[j].location)  
 else:  
 if self.TokenList[0].tokenList[j].operand[0][0] == "=":#if operand is literal, save in lit  
 arr1 = self.TokenList[0].tokenList[j].operand[0].split('\'')  
 lit = arr1[1]  
  
 self.symtabList.append(self.TokenList[0].symTab)  
 self.literaltabList.append(self.TokenList[0].literalTab)  
  
 j = i  
 start = i  
 addr = 0  
 i = 0  
 self.TokenList.append(TokenTable(SymbolTable(), LiteralTable(), self.instTable))  
 while j < len(self.lineList):  
 if self.lineList[j][0] == '.':  
 j += 1  
 continue  
 self.TokenList[1].putToken(self.lineList[j])  
  
 self.TokenList[1].tokenList[i].location = addr  
 if i>1:  
 if self.TokenList[1].tokenList[i].location - self.TokenList[1].tokenList[i-1].location >= 0:#section 2  
 self.TokenList[1].Listlength = self.TokenList[1].tokenList[i].location  
 if self.instTable.hash\_get(self.TokenList[1].tokenList[i].operator) > 0:  
 if self.instTable.hash\_get(self.TokenList[1].tokenList[i].operator) == 3:  
 if self.TokenList[1].tokenList[i].operator[0] == "+":  
 addr += 4  
 else:  
 addr += 3  
 elif self.instTable.hash\_get(self.TokenList[1].tokenList[i].operator) == 2:  
 addr += 2  
 else:  
 if self.TokenList[1].tokenList[i].operator == "RESW":  
 tmpadr = int(self.TokenList[1].tokenList[i].operand[0])  
 addr += 3\*tmpadr  
 elif self.TokenList[1].tokenList[i].operator == "RESB":  
 tmpadr = int(self.TokenList[1].tokenList[i].operand[0])  
 addr += tmpadr  
 elif self.TokenList[1].tokenList[i].operator == "BYTE":  
 addr += 1  
 elif self.TokenList[1].tokenList[i].operator == "WORD":  
 addr += 3  
 elif self.TokenList[1].tokenList[i].operator == "EQU":  
 if self.TokenList[1].tokenList[i].operand != "\*":  
 arr = self.TokenList[1].tokenList[i].operand[0].split("-")  
 tmpadr1 = 0  
 tmpadr2 = 0  
 for k in range(len(self.TokenList[1].tokenList)):  
 if self.TokenList[1].tokenList[k].label == arr[0]:  
 tmpadr1 = self.TokenList[1].tokenList[k].location  
 if self.TokenList[1].tokenList[k].label == arr[1]:  
 tmpadr2 = self.TokenList[1].tokenList[k].location  
 addr = tmpadr1 - tmpadr2  
  
 elif self.TokenList[1].tokenList[i].operator == "LTORG":  
 addr += 3  
  
 if self.TokenList[1].tokenList[i].label == "WRREC":  
 self.TokenList[1].tokenList.remove(self.TokenList[1].tokenList[i])  
 break  
 j += 1  
 i += 1  
  
 start = j  
 for j in range(i):  
 if self.TokenList[1].tokenList[j].label != "":  
 self.TokenList[1].symTab.putSymbol(self.TokenList[1].tokenList[j].label, self.TokenList[1].tokenList[j].location)  
  
 self.symtabList.append(self.TokenList[1].symTab)  
 self.literaltabList.append(self.TokenList[1].literalTab)  
  
 j = start  
 addr = 0  
 i = 0  
 self.TokenList.append(TokenTable(SymbolTable(), LiteralTable(), self.instTable))#section 3  
 while(j < len(self.lineList)):  
 if self.lineList[j][0] == '.':  
 j += 1  
 continue  
 self.TokenList[2].putToken(self.lineList[j])  
  
 self.TokenList[2].tokenList[i].location = addr  
 if i>1:  
 if self.TokenList[2].tokenList[i].location - self.TokenList[2].tokenList[i-1].location >= 0:  
 self.TokenList[2].Listlength = self.TokenList[2].tokenList[i].location  
 if self.instTable.hash\_get(self.TokenList[2].tokenList[i].operator) > 0:  
 if self.instTable.hash\_get(self.TokenList[2].tokenList[i].operator) == 3:  
 if self.TokenList[2].tokenList[i].operator[0] == "+":  
 addr += 4  
 else:  
 addr += 3  
 elif self.instTable.hash\_get(self.TokenList[2].tokenList[i].operator) == 2:  
 addr += 2  
 else:  
 if self.TokenList[2].tokenList[i].operator == "RESW":  
 tmpadr = int(self.TokenList[2].tokenList[i].operand[0])  
 addr += 3 \* tmpadr  
 elif self.TokenList[2].tokenList[i].operator == "RESB":  
 tmpadr = int(self.TokenList[2].tokenList[i].operand[0])  
 addr += tmpadr  
 elif self.TokenList[2].tokenList[i].operator == "BYTE":  
 addr += 1  
 elif self.TokenList[2].tokenList[i].operator == "WORD":  
 addr += 3  
 elif self.TokenList[2].tokenList[i].operator == "EQU":  
 if self.TokenList[2].tokenList[i].operand != "\*":  
 arr = self.TokenList[2].tokenList[i].operand[0].split("-")  
 tmpadr1 = 0  
 tmpadr2 = 0  
 for k in range(len(self.TokenList[2].tokenList)):  
 if self.TokenList[2].tokenList[k].label == arr[0]:  
 tmpadr1 = self.TokenList[2].tokenList[k].location  
 if self.TokenList[2].tokenList[k].label == arr[1]:  
 tmpadr2 = self.TokenList[2].tokenList[k].location  
 addr = tmpadr1 - tmpadr2  
  
 elif self.TokenList[2].tokenList[i].operator == "LTORG":  
 addr += 3  
 if self.TokenList[2].tokenList[i].operator == "END":  
 self.TokenList[2].Listlength += 1  
 break  
 j += 1  
 i += 1  
  
 for j in range(i+1):  
 if self.TokenList[2].tokenList[j].label != "":  
 self.TokenList[2].symTab.putSymbol(self.TokenList[2].tokenList[j].label, self.TokenList[2].tokenList[j].location)  
 if len(self.TokenList[2].tokenList[j].operand) > 0:  
 if self.TokenList[2].tokenList[j].operator != "RSUB":  
 if self.TokenList[2].tokenList[j].operand[0][0] == '=':  
 arr = self.TokenList[2].tokenList[j].operand[0].split("'")  
 lit = arr[1]  
 if self.TokenList[2].tokenList[j].operator == "END":  
 self.TokenList[2].literalTab.putLiteral(lit, self.TokenList[2].tokenList[j].location)  
  
 self.symtabList.append(self.TokenList[2].symTab)  
 self.literaltabList.append(self.TokenList[2].literalTab)  
  
 def printSymbolTable(self, filename):  
 f = open(filename, 'w')  
 for i in range(len(self.symtabList[0].symbolList)):  
 line = str.format("%s\t%X\n" % (self.symtabList[0].symbolList[i], self.symtabList[0].locationList[i]))  
 f.write(line)  
 f.write("\n")  
 for i in range(len(self.symtabList[1].symbolList)):  
 line = str.format("%s\t%X\n" % (self.symtabList[1].symbolList[i], self.symtabList[1].locationList[i]))  
 f.write(line)  
 f.write("\n")  
 for i in range(len(self.symtabList[2].symbolList)):  
 line = str.format("%s\t%X\n" % (self.symtabList[2].symbolList[i], self.symtabList[2].locationList[i]))  
 f.write(line)  
 f.close()  
  
 def printLiteralTable(self, filename):  
 f = open(filename, 'w')  
 for i in range(len(self.literaltabList[0].literalList)):  
 line = str.format("%s\t%X\n" % (self.literaltabList[0].literalList[i], self.literaltabList[0].locationList[i]))  
 f.write(line)  
 f.write("\n")  
 for i in range(len(self.literaltabList[2].literalList)):  
 line = str.format("%s\t%X\n" % (self.literaltabList[2].literalList[i], self.literaltabList[2].locationList[i]))  
 f.write(line)  
  
 def pass2(self):  
 for i in range(len(self.TokenList[0].tokenList)):#section 1  
 self.TokenList[0].makeObjectCode(i)  
 self.codeList.append(self.TokenList[0].getObjectCode(i))  
 for i in range(len(self.TokenList[1].tokenList)):#section 2  
 self.TokenList[1].makeObjectCode(i)  
 self.codeList.append(self.TokenList[1].getObjectCode(i))  
 for i in range(len(self.TokenList[2].tokenList)):#section 3  
 self.TokenList[2].makeObjectCode(i)  
 self.codeList.append(self.TokenList[2].getObjectCode(i))  
  
 def printObjectCode(self, filename):  
 f = open(filename, 'w')  
 start = 0  
 i = 0  
 for k in range(len(self.TokenList)):  
 for i in range(len(self.TokenList[k].tokenList)):#for Starting point, EXTDEF, EXTREF  
 if self.codeList[start+i] == '':  
 if self.TokenList[k].tokenList[i].operator == "START" or self.TokenList[k].tokenList[i].operator == "CSECT":  
 str1 = "H"+self.TokenList[k].tokenList[i].label+"\t000000"  
 str2 = str.format("%06X" % self.TokenList[k].Listlength)  
 f.write(str1+str2)  
 elif self.TokenList[k].tokenList[i].operator == "EXTDEF":  
 f.write("D")  
 for j in range(len(self.TokenList[k].tokenList[i].operand)):  
 str1 = self.TokenList[k].tokenList[i].operand[j]  
 f.write(str1+str.format("%06X" % self.TokenList[k].symTab.search(str1)))  
 elif self.TokenList[k].tokenList[i].operator == "EXTREF":  
 f.write("R")  
 for j in range(len(self.TokenList[k].tokenList[i].operand)):  
 f.write(self.TokenList[k].tokenList[i].operand[j]+ " ")  
 f.write("\n")  
 else:  
 break  
  
 startadr = 0  
 txtlen = 0  
 j = i  
 txt = 0  
 while txtlen <= 30:#first T line  
 if txtlen+self.TokenList[k].tokenList[j].byteSize <= 30:  
 if self.codeList[start+j] == "05":  
 txtlen += 1  
 break  
 txtlen += self.TokenList[k].tokenList[j].byteSize  
 else:  
 break  
 j += 1  
  
 f.write("T"+str.format("%06X%02X" % (startadr, txtlen)))  
 for i in range(i, len(self.TokenList[k].tokenList)):#write T codeList  
 f.write(self.codeList[start+i])  
 txt += self.TokenList[k].tokenList[i].byteSize  
 if txt == txtlen:  
 break  
 i = j  
 startadr = txtlen  
 txtlen = 0  
 txt = 0  
 while txtlen <= 30:#second T line  
 if k == 2:  
 break  
 if txtlen + self.TokenList[k].tokenList[j].byteSize <= 30:  
 if self.codeList[start+j+1] == '':  
 break  
 txtlen += self.TokenList[k].tokenList[j].byteSize  
 else:  
 break  
 j += 1  
 if k != 2:  
 f.write("\nT"+str.format("%06X%02X" % (startadr, txtlen)))#write second T codeList  
 for i in range(i, len(self.TokenList[k].tokenList)):  
 f.write(self.codeList[start+i])  
 txt += self.TokenList[k].tokenList[i].byteSize  
 if i == j:  
 break  
 for i in range(i, len(self.TokenList[k].tokenList)):#for literal  
 if self.TokenList[k].tokenList[i].operator == "LTORG":  
 f.write("\nT"+str.format("%06X" % self.TokenList[k].literalTab.locationList[0])+str.format("%02X" % self.TokenList[k].tokenList[i].byteSize)+str.format("%X%X%X" % (ord(self.TokenList[k].literalTab.literalList[0][0]), ord(self.TokenList[k].literalTab.literalList[0][1]), ord(self.TokenList[k].literalTab.literalList[0][2]))))  
  
 for cnt in range(3, len(self.TokenList[k].tokenList)):#for modify  
 if len(self.TokenList[k].tokenList[cnt].operand) == 0:  
 cnt += 1  
 continue  
 if k == 0:  
 if "RDREC" in self.TokenList[k].tokenList[cnt].operand[0] or "WRREC" in self.TokenList[k].tokenList[cnt].operand[0]:  
 str1 = str.format("%06X" % (self.TokenList[k].tokenList[cnt].location+1))  
 f.write("\nM"+str1+"05+"+self.TokenList[k].tokenList[cnt].operand[0])  
 elif k == 1:  
 if "BUFEND" in self.TokenList[k].tokenList[cnt].operand[0] or "BUFFER" in self.TokenList[k].tokenList[cnt].operand[0] or "LENGTH" in self.TokenList[k].tokenList[cnt].operand[0]:  
 if self.TokenList[k].tokenList[cnt].operand[0] == "BUFEND-BUFFER":  
 f.write("\nM"+str.format("%06X" % (self.TokenList[k].tokenList[cnt].location))+"06+BUFEND")  
 f.write("\nM"+str.format("%06X" % (self.TokenList[k].tokenList[cnt].location))+"06+BUFFER")  
 else:  
 f.write("\nM"+str.format("%06X" % (self.TokenList[k].tokenList[cnt].location+1))+"05+"+self.TokenList[k].tokenList[cnt].operand[0])  
  
 elif k == 2:  
 if "BUFEND" in self.TokenList[k].tokenList[cnt].operand[0] or "BUFFER" in self.TokenList[k].tokenList[cnt].operand[0] or "LENGTH" in self.TokenList[k].tokenList[cnt].operand[0]:  
 f.write("\nM"+str.format("%06X" % (self.TokenList[k].tokenList[cnt].location+1))+"05+"+self.TokenList[k].tokenList[cnt].operand[0])  
 f.write("\nE")  
 if k == 0:  
 f.write("000000")  
 start += (i+1)  
 f.write("\n\n")  
  
  
  
  
  
assembler = Assembler("inst.data")  
assembler.loadInputFile("input.txt")  
assembler.pass1()  
assembler.printSymbolTable("symtab\_20150286.txt")  
assembler.printLiteralTable("literaltab\_20150286.txt")  
assembler.pass2()  
assembler.printObjectCode("output\_20150286.txt")

**InstTable.py**

class InstTable:  
 def \_\_init\_\_(self, filename):  
 self.instMap = {}#instMap dictionary  
 f = open(filename, 'r')  
 while True:  
 line = f.readline()  
 if not line: break  
 arr = line.split('\t')  
 inst = Instruction(arr[1])  
 self.instMap[arr[0]] = inst  
  
 f.close()  
  
 def hash\_get(self, line):#if line is instruction, return format of line  
 if line[0] == '+':  
 newline = line[1:]  
 return self.instMap[newline].format  
 else:  
 if line in self.instMap.keys():  
 return self.instMap[line].format  
 else:  
 return -1  
  
  
class Instruction:  
 def \_\_init\_\_(self, line):  
 self.format = 0  
 self.opcode = ""  
 self.numberOfOperand = 0  
 self.parsing(line)  
  
 def parsing(self, line):  
 arr = line.split(",")  
 self.format = int(arr[0])  
 self.opcode = arr[1]  
 self.numberOfOperand = int(arr[2])

**LiteralTable.py**

class LiteralTable:  
 def \_\_init\_\_(self):  
 self.literalList = []  
 self.locationList = []  
  
 def putLiteral(self, literal, location):  
 self.literalList.append(literal)  
 self.locationList.append(location)  
  
 def modifyLiteral(self, literal, newlocation):  
 for i in range(len(self.literalList)):  
 if self.literalList[i] == literal:  
 self.locationList[i] = newlocation  
 break  
  
 def search(self, literal):  
 for i in range(len(self.literalList)):  
 if self.literalList[i] == literal:  
 return self.locationList[i]  
 return -1

**SymbolTable.py**

class SymbolTable:  
 def \_\_init\_\_(self):  
 self.symbolList = []  
 self.locationList = []  
  
 def putSymbol(self, symbol, location):  
 self.symbolList.append(symbol)  
 self.locationList.append(location)  
  
 def modifySymbol(self, symbol, newlocation):  
 for i in range(len(self.symbolList)):  
 if self.symbolList[i] == symbol:  
 self.locationList[i] = newlocation  
 break  
 return -1  
  
 def search(self, symbol):  
 for i in range(len(self.symbolList)):  
 if self.symbolList[i] == symbol:  
 return self.locationList[i]  
 return -1

**TokenTable.py**

from LiteralTable import LiteralTable  
from SymbolTable import SymbolTable  
from InstTable import InstTable  
  
class TokenTable:  
 def \_\_init\_\_(self, symTab, literalTab, instTab):  
 self.nFlag = 32  
 self.iFlag = 16  
 self.xFlag = 8  
 self.bFlag = 4  
 self.pFlag = 2  
 self.eFlag = 1  
 self.tokenList = []  
 self.symTab = symTab  
 self.literalTab = literalTab  
 self.instTab = instTab  
 self.Listlength = 0  
  
 def putToken(self, line):  
 self.tokenList.append(Token(line))  
  
 def getToken(self, index):  
 return self.tokenList[index]  
  
 def makeObjectCode(self, index):  
 objectcode = 0#type int objectcode  
 opcode = 0#type int opcode  
 address = 0  
 if self.instTab.hash\_get(self.tokenList[index].operator) > 0:#Instruction  
 if self.tokenList[index].operator[0] == '+':#format 4  
 newline = self.tokenList[index].operator[1:]  
 opcode = int(self.instTab.instMap[newline].opcode, 16)  
 else:#format 3  
 opcode = int(self.instTab.instMap[self.tokenList[index].operator].opcode, 16)  
 if self.instTab.hash\_get(self.tokenList[index].operator) == 3:#format 3, 4  
 self.tokenList[index].setFlag(self.nFlag, 1)#basically n, i, p = 1  
 self.tokenList[index].setFlag(self.iFlag, 1)  
 self.tokenList[index].setFlag(self.pFlag, 1)  
 self.tokenList[index].byteSize = 3  
 if self.tokenList[index].operator[0] == '+':#format4 p = 0, e = 1  
 self.tokenList[index].setFlag(self.pFlag, 0)  
 self.tokenList[index].setFlag(self.eFlag, 1)  
 self.tokenList[index].byteSize = 4  
 if self.tokenList[index].operator == "RSUB":#RSUB exception  
 self.tokenList[index].setFlag(self.pFlag, 0)  
 objectcode = opcode \* 65536 + self.tokenList[index].nixbpe \* 4096  
 self.tokenList[index].objectCode = hex(objectcode)[2:].upper()  
 return  
 if self.tokenList[index].operand[0][0] == '#':#immediate  
 self.tokenList[index].setFlag(self.pFlag, 0)  
 self.tokenList[index].setFlag(self.nFlag, 0)  
 elif self.tokenList[index].operand[0][0] == '@':#indirect  
 self.tokenList[index].setFlag(self.iFlag, 0)  
 if len(self.tokenList[index].operand) == 2 and self.tokenList[index].operand[1] == "X":  
 self.tokenList[index].setFlag(self.xFlag, 1)  
  
 if self.tokenList[index].operand[0][0] == '#' or self.tokenList[index].operand[0][0] == '@':  
 str1 = self.tokenList[index].operand[0][1:]  
 else:  
 str1 = self.tokenList[index].operand[0]  
  
 check = -1  
 for i in range(len(self.symTab.symbolList)):  
 if str1 == self.symTab.symbolList[i]:  
 check = i#if it is same section symbol, check = i else, check = -1  
 break  
 if check != -1:#symbol  
 address = self.symTab.locationList[i] - self.tokenList[index+1].location  
 if address < 0:  
 address = 4096 + self.symTab.locationList[i] - self.tokenList[index+1].location  
 else:#immediate number  
 if str1[0] >= '0' and str1[0] <= '9':  
 address = int(str1, 16)  
 else:  
 if self.tokenList[index].operand[0][0] == '=':#literal  
 address = self.literalTab.locationList[0] - self.tokenList[index+1].location  
 else:#another section symbol  
 address = 0  
 if self.tokenList[index].byteSize == 3:  
 objectcode = opcode \* 65536 + self.tokenList[index].nixbpe \* 4096 + address  
 elif self.tokenList[index].byteSize == 4:  
 objectcode = opcode \* 16777216 + self.tokenList[index].nixbpe \* 1048576 + address  
 else:#format 2  
 self.tokenList[index].byteSize = 2  
 if len(self.tokenList[index].operand) >= 1:  
 if self.tokenList[index].operand[0] == "X":  
 address = 1  
 elif self.tokenList[index].operand[0] == "A":  
 address = 0  
 elif self.tokenList[index].operand[0] == "S":  
 address = 4  
 elif self.tokenList[index].operand[0] == "T":  
 address = 5  
 objectcode = opcode \* 256 + address \* 16  
 if len(self.tokenList[index].operand) == 2:  
 if self.tokenList[index].operand[1] == "X":  
 address = 1  
 elif self.tokenList[index].operand[1] == "A":  
 address = 0  
 elif self.tokenList[index].operand[1] == "S":  
 address = 4  
 elif self.tokenList[index].operand[1] == "T":  
 address = 5  
 objectcode = objectcode + address  
  
 self.tokenList[index].objectCode = hex(objectcode)[2:].upper()  
 if len(self.tokenList[index].objectCode)==5:#if first number is 0, put it in  
 self.tokenList[index].objectCode = "0"+self.tokenList[index].objectCode  
  
 else:#not instruction  
 if self.tokenList[index].operator == "LTORG":  
 self.tokenList[index].objectCode = self.literalTab.literalList[0]  
 self.tokenList[index].byteSize = 3  
 elif self.tokenList[index].operator == "BYTE":  
 arr = self.tokenList[index].operand[0].split('\'')  
 self.tokenList[index].objectCode = arr[1]  
 self.tokenList[index].byteSize = 1  
 elif self.tokenList[index].operator == "WORD":  
 self.tokenList[index].objectCode = "000000"  
 self.tokenList[index].byteSize = 3  
 elif self.tokenList[index].operator == "END":  
 self.tokenList[index].objectCode = self.literalTab.literalList[0]  
 self.tokenList[index].byteSize = 1  
  
  
 def getObjectCode(self, index):  
 return self.tokenList[index].objectCode  
  
  
class Token:  
 def \_\_init\_\_(self, line):  
 self.location = 0  
 self.label = ""  
 self.operator = ""  
 self.operand = []  
 self.comment = ""  
 self.nixbpe = 0  
 self.objectCode = ""  
 self.byteSize = 0  
 self.parsing(line)  
  
 def parsing(self, line):  
 arr = line.split('\t')  
 self.label = arr[0]  
 self.operator = arr[1]  
 if len(arr) > 3:  
 self.comment = arr[3]  
 if len(arr) > 2:  
 op = arr[2].split(",")  
 for i in range(len(op)):  
 self.operand.append(op[i])  
  
 def setFlag(self, flag, value):  
 if value == 1:  
 self.nixbpe += flag  
 elif value == 0:  
 self.nixbpe -= flag  
  
 '''def getFlag(self, flag):  
 return self.nixbpe & flag'''