**과목명: 시스템프로그래밍**

**CSE4110-02**

**<<Project #2>>**

**서강대학교 [공학부 컴퓨터공학과]**

**[20171662]**

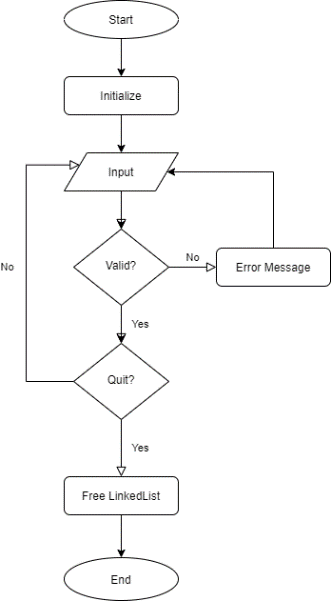
**[이나연]**

목 차

1. **프로그램 개요**
2. **프로그램 설명**
   1. 프로그램 흐름도
   2. assembler 구현 알고리즘
   3. pass1구현 알고리즘
   4. pass2구현 알고리즘
   5. object code 알고리즘
3. **모듈 정의**
   1. shellcmd.c
      1. type()
   2. opcmd.c
      1. searchformat()
   3. sicxecmd.c
      1. insertSymbol()
      2. SearchSymbol()
      3. assemble()
      4. pass\_1()
      5. pass\_2()
      6. findObj()
      7. insertObj()
      8. FreeObj()
      9. ShowSymbol()
      10. FreeSymbol()
4. **전역 변수 정의**
   1. SYMTAB\* symhead
   2. ObjectCode\* obhead
   3. int prog\_len
   4. int start\_addr
   5. int base\_addr
   6. char helpprint[]
5. **코드 설명**
   1. 20171662.c
      1. main()
   2. shellcmd.c
      1. type()
   3. opcmd.c
      1. searchformat()
   4. sicxecmd.c
      1. insertSymbol()
      2. SearchSymbol()
      3. assemble()
      4. pass\_1()
      5. pass\_2()
      6. findObj()
      7. insertObj()
      8. FreeObj()
      9. ShowSymbol()
      10. FreeSymbol()
6. **기타 코드**
   1. 20171662.c
   2. 20171662.h
   3. makefile
7. 프로그램 개요

해당 프로그램은 프로젝트 1에서 구현한 셸(shell)에 assemble 기능을 추가하는 프로그램이다. SIC/XE machine의 assembly program source 파일을 입력 받아서 .obj 파일 및 .lst 파일을 생성하고, 어셈블리 과정 중 생성된 symbol table과 결과물인 object 파일 및 .lst 파일을 볼 수 있는 type filename 기능을 제공한다.

1. 프로그램 설명
   1. 프로그램 흐름도

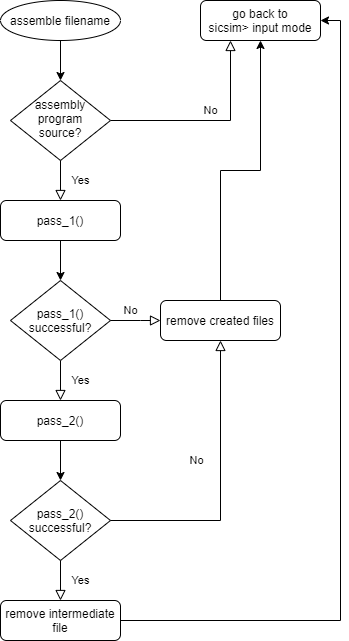


전체적인 프로그램의 흐름도는 다음과 같다. 먼저 프로그램이 실행되면, 추후 command들이 정상적으로 실행되기 위하여 필요한 여러 변수들에 메모리를 할당하여 주는 등의 Initialization을 실행한다. 그런 다음,

>>> sicsim> “input”

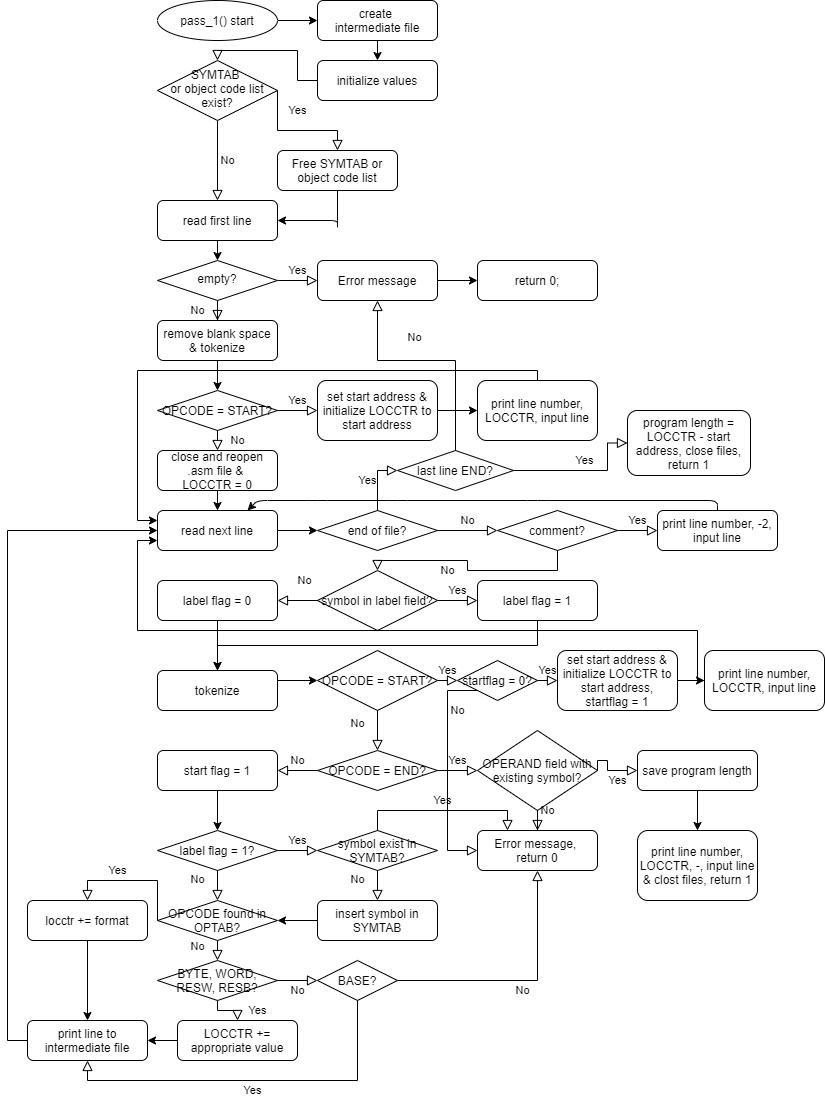
의 형태로 input을 받아 만약 해당 input이 valid하다면 해당 command의 기능을 수행하도록 하고, 만약 그렇지 않다면 적절한 에러 처리와 함께 다시 input을 새로 받는다. 만약 quit 이나 q에 해당하는 프로그램 종료에 관한 command가 들어오면, 할당된 메모리를 해제한 후 프로그램을 정상적으로 종료한다.

* 1. assembler 구현 알고리즘



전체적인 어셈블러의 흐름도는 위와 같다. 만약 input으로 assemble filename의 형식이 들어왔다면, 우선 파일이 assembly program source인지를 확인한다. 만약 파일명이 .asm파일이 아니라면, 에러 문구를 출력 후 다시 sicsim> 의 input을 받는 단계로 돌아간다. object code를 생성하기까지 크게 2가지의 알고리즘을 사용하였는데, 교재에 나와있는 pass1과 pass2를 사용하였다. 만약 pass 1 수행 도중 에러가 나왔다면, 에러가 난 줄을 먼저 출력한 뒤 해당 알고리즘 도중에 생성된 intermediate file 을 지운 후 다시 input을 받게끔 한다. 만약 pass 2 수행 도중 에러가 나왔다면, 에러가 난 줄을 먼저 출력한 뒤 해당 알고리즘 도중 생성된 .lst file 및 .obj file과 이전 pass 1에서 생성된 intermediate file을 지운 후 다시 input을 받게끔 한다.

* 1. pass1구현 알고리즘



pass 1의 흐름도는 위와 같다. 해당 프로그램 중 0이 반환되면 pass 1 의 실패를 의미하므로, 그 이후의 과정은 2.2의 흐름도를 참조하면 된다. 우선 pass 1이 시작하자마자 intermediate file을 생성한다. 그런 다음 필요한 변수들을 초기화한다. 만약 SYMTAB이나 Object Code list가 비어있지 않다면, 이전의 assemble 과정에서 생성된 것들이므로 메모리 해제를 시켜준 후 다시 새롭게 head에 메모리를 할당한다. 그런 다음 어셈블리 프로그램의 첫 줄을 읽는다. 만약 첫 줄부터 없으면 적절한 에러 문구 출력 후 0을 반환한다.

첫 줄이 존재한다면 해당 줄을 tokenize하여 각각의 field에 담긴 symbol/ opcode/ operand를 분리한다. 만약 START가 OPCODE로 들어왔다면 operand field의 16진수 수를 start address로 설정하고 location counter를 start address로 초기화한다. 그런 다음 intermediate file에 line number, location counter, input된 줄을 print한다. 그런 다음 다음 줄을 입력 받는다. START가 들어오지 않았다면 다시 파일을 닫고 다시 연다. 이 때 location counter는 0으로 초기화한다.

다음 줄을 입력 받는다. 만약 다음 줄이 없다면 (파일이 끝났다면) 마지막으로 입력 받은 OPCODE가 END인지 확인한다. 만약 END가 아니라면 END없이 파일이 끝났다는 뜻이므로 에러 문구를 출력해준 후 0을 반환한다. 마지막 줄로 END가 들어왔다면 program length를 location counter – start address로 설정해준 후 intermediate file 및 어셈블리 파일을 닫고, 1을 반환한다.

파일이 끝나지 않았다면, comment인지 확인한다. 만약 comment가 맞다면, intermediate file에 line number, location counter 대신 -2, 그리고 입력 받은 줄을 출력한다. -2는 추후 pass 2에서 해당 줄이 comment임을 보여주는 flag로 작용한다. comment가 아니라면 label이 존재하는지 확인해준다. 만약 symbol이 label field에 존재한다면, label flag를 1로 설정해주고, 없다면 0으로 설정해준다.

입력 받은 한 줄을 strtok을 이용하여 빈 칸을 기준으로 tokenize한다.

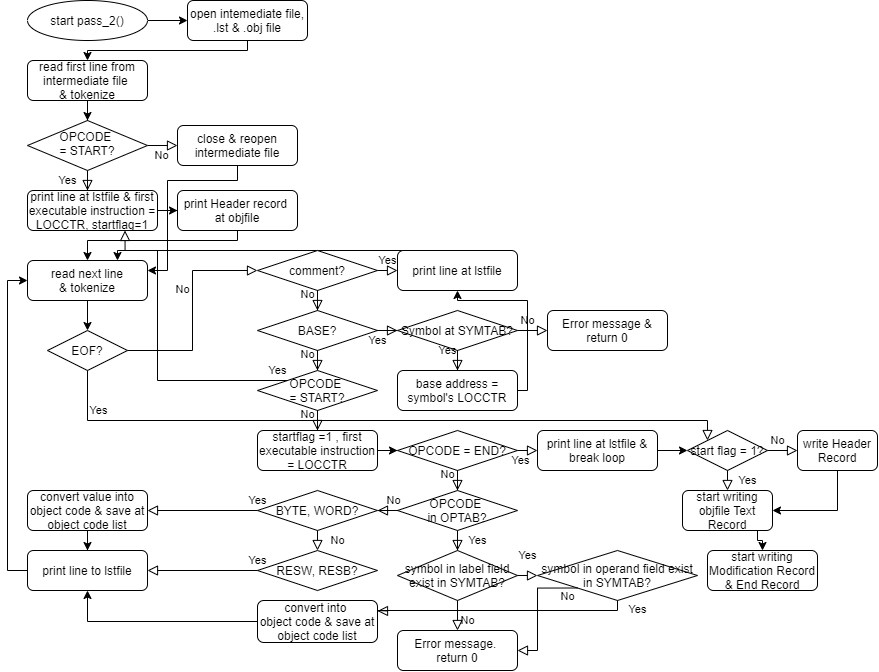
만약 START가 이제 들어왔다면, start flag를 확인한다. 만약 해당 프로그램이 START 없이 유효한 OPCODE가 들어와 이미 프로그램이 시작해 있거나, 이미 START가 들어왔었다면 start flag가 1로 초기화되어있었을 것이다. 만약 start flag가 1이라면 START가 중간에 들어온 것이므로 에러 문구 출력 후 0을 반환한다. start flag가 0이라면 처음 START가 들어온 유효한 줄이기 때문에 start address를 해당 줄의 operand field에 있는 16진수로 저장하고, location counter를 start address로 설정한 후 start flag를 1로 설정한다. 그런 다음 intermediate file 에 line number, location counter, input 받은 line을 출력한다. 다시 다음 줄을 입력 받는다.

만약 START가 아니라면 OPCODE에 END가 들어왔는지 확인한다. 만다면 operand field에 이미 존재하는 symbol이 들어있는지 확인한다. 맞다면 program length를 location counter – start address로 저장하고, intermediate file에 line number, location counter, -, input 받은 줄을 출력한다. ‘-‘는 이후 pass 2에서 label field가 비었음을 나타내는 flag로 작용한다. 그런 다음 성공적으로 pass1이 끝났음을 의미하므로 intermediate file과 어셈블리 파일을 닫은 후 1을 반환한다.

START와 END가 모두 아니라면 다른 OPCODE로 파일이 시작한다는 뜻이므로 start flag를 1로 설정한다. 그런 다음 앞서 설정한 label flag를 확인한다. 1이라면 해당 필드에 저장되어있는 symbol이 현재까지 존재하는 SYMTAB에 존재하는지를 확인한다. 만약 존재한다면 이미 존재하는 symbol이 두 번 정의된 것이므로 에러 문구를 출력한 후 0을 반환한다. 만약 존재하지 않는다면 SYMTAB에 해당 줄의 location counter와 함께 저장해준다.

label flag가 0이거나 symbol을 SYMTAB에 삽입한 후엔 OPCODE를 확인한다. 만약 해당 OPCODE가 존재하는 OPTAB에 저장되어 있는 opcode라면 location counter에 해당 opcode의 format number 만큼을 더한다. 만약 opcode앞에 +가 붙어있다면 format이 4라는 것이므로 4를 더한다. 만약 저장되어 있지 않은 opcode라면, BYTE, WORD, RESW, RESB인지 확인한다. 만약 WORD라면, location counter에 3을 더한다. 만약 RESW라면 operand field에 있는 10진수 숫자에 3을 곱한 값을 location counter에 더한다. 만약 RESB라면 operand field에 있는 10진수 숫자를 location counter에 더한다. 만약 BYTE라면 operand field에 저장되어 있는 문자열이나 16진수의 길이를 더한다. 다 아니라면 BASE인지 확인한다. BASE라면 location counter를 증가시키지 않고 넘어간다. 만약 위의 경우에 모두 포함되어있지 않은 opcode가 들어왔다면 유효하지 않은 opcode라는 것이므로 에러 문구 출력 후 0을 반환한다. 위의 과정을 수행한 후 0이 반환되지 않았다면 유효한 경우이므로 intermediate file에 line number, location counter, input 받은 줄을 저장한다. 만약 baseflag가 1이라면 location counter 자리에 -1을 쓴다. 이후 -1은 pass 2에서 BASE임을 나타내는 flag로 작용한다. 또한 label flag가 0이라면 symbol자리에 –를 대신 출력하여 pass2 에서 label이 존재하지 않는다는 flag로 작용한다. 해당 과정 이후 다음 줄을 입력 받아 파일이 끝날 때까지 위의 과정을 반복한다. 다음 줄로 넘어갈 때마다 line number를 5씩 증가시킨다.

* 1. pass2구현 알고리즘



pass 2의 흐름도는 위와 같다. 우선 pass1에서 생성된 intermediate file을 열고, .lst file과 .obj file을 생성한다. intermediate file에서 한 줄을 읽어오고 line number, location counter, label, opcode, operand 5가지의 부분으로 나눈다. 만약 opcode가 START라면 lst file에 해당 줄을 출력하고 first executable instruction을 LOCCTR로 설정한다. start flag도 1로 설정한다. 그런 다음 object file에 Header record를 출력한 후 다음 줄을 읽게끔 한다. 만약 START 가 아니라면 intermediate file을 닫고 다시 연다. 그 후 다시 해당 줄을 읽게끔 한다.

만약 파일을 다 읽었다면 start flag가 1인지를 확인한다. 0이라면 아직 header record가 쓰여지지 않은 상태일 것이므로 header record 를 object file에 출력 후 modification record와 end record 마저 출력한다. 만약 1이라면 바로 text record, modification record, end record를 출력한다.

파일이 끝나지 않았을 땐 우선 comment line인지 확인한다. comment라면 해당 줄을 linenumber와 함께 list file에 출력하게끔 한다. 그런 다음 그 다음 줄을 읽는 것으로 해당 과정을 다시 반복한다.

comment가 아니라면 BASE line인지 확인한다. 맞다면 operand field에 있는 symbol이 SYMTAB에 존재하는지 확인한다. 존재하지 않는다면 에러 문구 출력 후 0을 반환한다. 존재한다면 base address를 해당 symbol의 LOCCTR로 저장한다. 그런 다음 해당 줄을 list file에 출력한 후 그 다음 줄을 읽는 것으로 위의 과정을 반복한다.

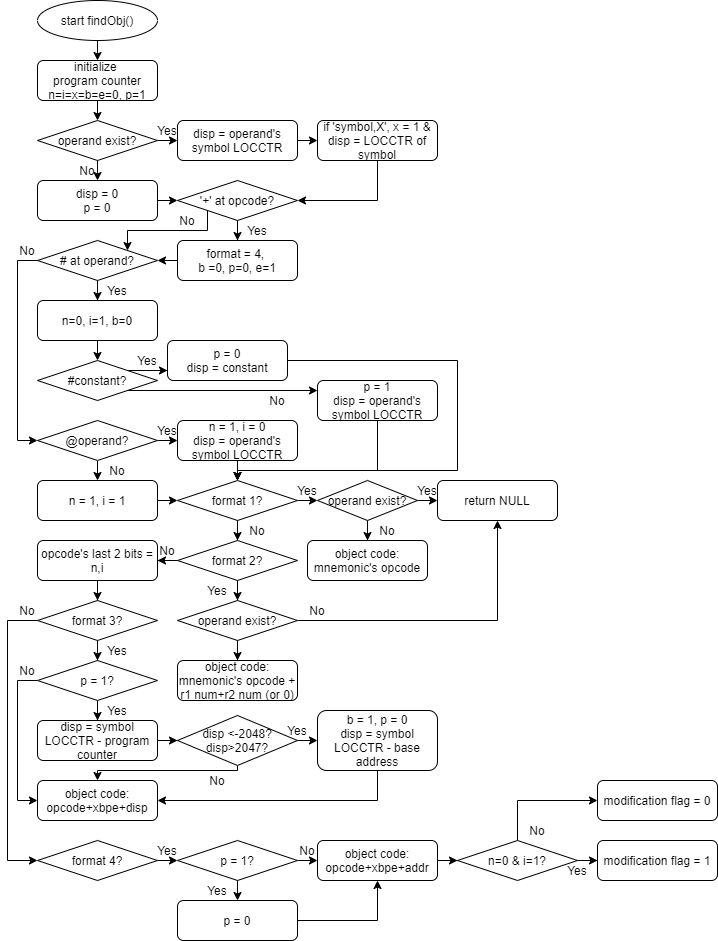
BASE가 아니라면 다시 OPCODE가 START인지 확인한다. START가 맞다면 위와 같이 lst file에 해당 줄을 출력하고 first executable instruction을 LOCCTR로 설정한다. start flag도 1로 설정한다. 그런 다음 object file에 Header record를 출력한 후 다음 줄을 읽게끔 한다. START도 아니라면 start flag를 1로 설정한다. 그런 다음 first executable instruction을 LOCCTR로 설정한다.

그런 다음 OPCODE가 END인지 확인한다. 맞다면 해당 줄을 lst file에 출력한 후 start flag가 1인지 확인한다. 맞다면 object file에 text record, modification record, end record를 쓰게끔 한다. 아니라면 header record를 먼저 출력 후 나머지 record들을 출력한다.

END도 아니라면 OPCODE가 OPTAB안에 존재하는지 확인한다. 존재한다면 label field가 비어있지 않을 때 label field안의 symbol이 SYMTAB에 존재하는지 확인한다. 존재하지 않는다면 에러 문구 출력 후 0을 반환한다. 존재한다면 operand field에 있는 symbol도 SYMTAB에 존재하는지 확인한다. 존재하지 않는다면 역시 에러 문구 반환 후 0을 반환한다. 존재한다면 해당 줄의 object code를 생성한다. object code 생성 방법은 2.5를 참고한다.

OPCODE가 OPTAB안에 존재하지 않는다면, BYTE/WORD인지 확인한다. 맞다면 operand field안의 상수를 그대로 object code로 변환한 후 해당 object code를 LOCCTR과 함께 object code list에 저장한다. 그런 다음 lst file에 해당 줄을 object code까지 함께 출력 후 그 다음 줄을 읽게끔 하여 위의 과정을 반복한다. 아니라면 RESW/RESB인지 확인한다. 맞다면 해당 줄을 object code없이 lst file에 출력한 후 그 다음 줄을 읽게끔 하여 위의 과정을 반복한다. 위의 경우 말고의 나머지 경우는 pass 1에서 이미 에러 처리를 하여 pass 2까지 넘어오지 않으므로 고려하지 않는다.

* 1. object code 알고리즘



object code 결정의 흐름도는 위와 같다. program counter를 현재 줄의 LOCCTR에 format의 번호만큼 더하여 결정한다. 이는 그 다음 LOCCTR의 값과 같다. format변수에 해당 opcode의 format 번호도 저장한다. 그런 다음 nixpbe를 각각 000100으로 초기화한다. p는 base relative를 제외하고는 보통 1이기 때문에 1로 초기화하였다. 만약 operand가 존재한다면 우선 disp를 해당 operand field의 symbol의 LOCCTR로 저장해준다. 만약 symbol,X의 형태로 존재한다면 x = 1로 초기화해준 후 ,X 이전의 symbol의 LOCCTR을 disp에 저장해준다. 만약 operand field가 비어있다면 disp = 0, p = 0으로 초기화해준다.

만약 opcode의 첫 문자가 +이라면 format 4임을 의미한다. 따라서 format을 4로 초기화시켜준 후 b=0, p=0, e=1로 설정해준다.

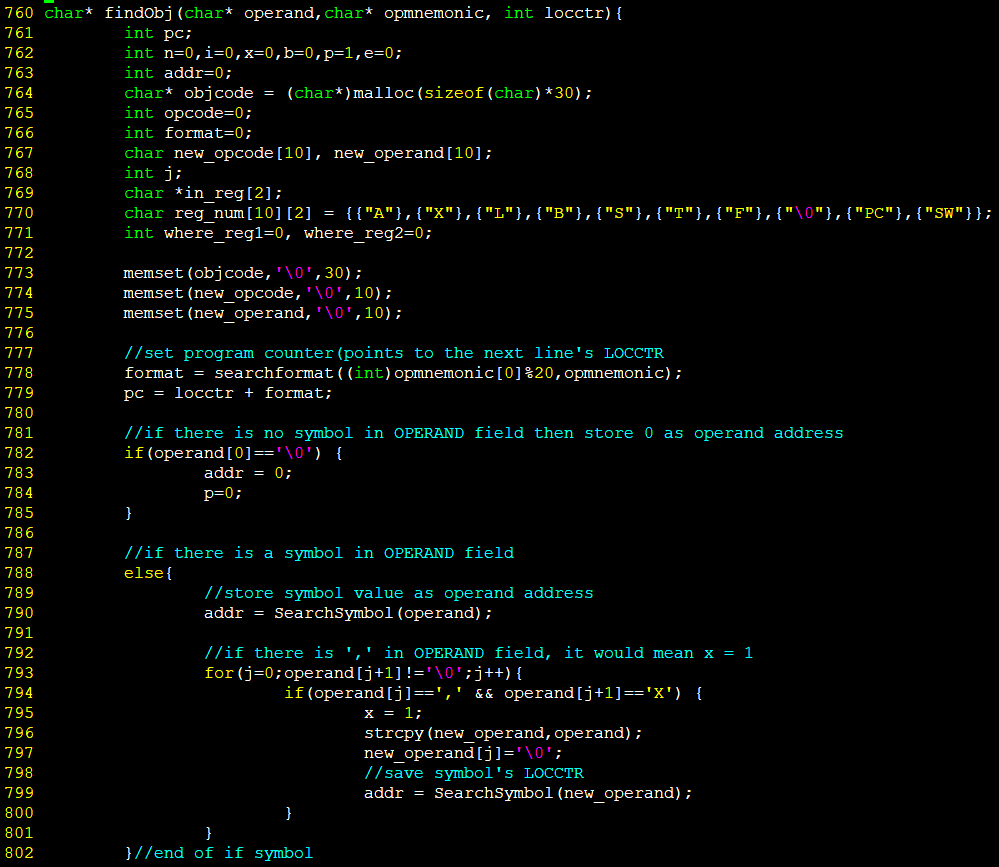
만약 opcode의 첫 문자가 #이라면 immediate addressing 이라는 뜻이므로 n=0, i=1, b=0으로 설정해준다. 만약 #다음 상수가 온다면 pc relative도 아니므로 p=0으로 초기화해준 후 disp를 해당 상수의 값으로 한다. 만약 #다음 symbol이 온다면 p=1, disp=symbol의 LOCCTR로 설정해준다.

만약 opcode의 첫 문자가 @라면 indirect addressing mode이므로 n=1, i=0으로 설정해준 후 disp=@다음의 symbol의 LOCCTR을 저장해준다.

만약 opcode의 첫 문자가 #이나 @이 아니라면 simple addressing이므로 n = 1, i = 1로 설정해준다.

만약 format이 1이라면, operand가 존재하는지 확인해준 후 존재한다면 틀린 형식이므로 NULL을 반환해준다. 존재하지 않는다면 object code에 해당 mnemonic의 opcode를 그대로 두 half byte형식으로 (%02X) 저장해준다.

만약 format이 2라면, opcode r1(,r2)의 형태여야 한다. operand가 존재하는지 확인해준 후 존재하지 않는다면 틀린 형식이므로 NULL을 반환해준다. 존재한다면 object code에 mnemonic의 opcode 2 half byte, r1의 레지스터 넘버 1half byte, r2가 존재한다면 r2의 레지스터 넘버 1half byte, 존재하지 않는다면 0을 저장해준다. (총 4 half byte)



format 3이나 4라면 opcode의 맨 뒷 두 비트를 n과 i로 변환해준다.

만약 format이 3이라면 p=1인지 확인한다. 맞다면 disp=symbol의 LOCCTR – program counter로 변환해준다. 만약 disp가 -2048보다 작거나 2047보다 큰 경우 base relative이어야 하므로 p=0, b=1로 설정해준 후 disp를 symbol의 LOCCTR – base address로 설정해준다. p=0 인 경우나 위의 과정을 다 거친 다음 object code에 opcode(2)+xbpe(1)+disp(3)(6 half byte)로 설정해준다.

format이 4라면 p=1인지 확인하고 맞다면 format 4는 항상 b=p=0이므로 p=1로 맞추어준다. 그런 다음 object code를 opcode(2)+xbpe(1)+addr(5)로 저장해준다. 위의 과정에서 immediate addressing이라면 #옆의 상수의 값 또는 symbol의 LOCCTR이 들어가 있을 것이고, 그 외의 경우에도 symbol의 LOCCTR이 들어가있을 것이기 때문에 별다른 과정을 거치지 않아도 된다. 그런 다음 만약 immediate addressing이라면 modification record에 쓰여져야 하므로 modification flag를 1로 한다.

생성된 object code를 object code linked list에 modification flag과 LOCCTR과 함께 저장한다.

1. 모듈 정의
   1. shellcmd.c

해당 파일은 shell 관련 명령어들을 처리하는 파일이다. 프로젝트 1을 구현하기 위해 주로 쓰였지만, 프로젝트 2의 type filename 명령어를 처리하기 위해 해당 파일에 새로운 모듈을 추가하였다.

* + 1. int type(char\* input)

1. 기능

input으로 들어온 file을 현재 디렉토리에서 찾고 안의 내용을 출력해준다. 만약 해당 디렉토리에 파일이 없다면 에러 문구를 출력하고 -1을 반환한다. 출력이 성공적으로 되었다면 1을 반환한다.

1. 사용 변수

* FILE\* fp: FILE 포인터로 주어진 file을 의 fopen한 포인터를 저장하는 데에 사용된다.
* char\* input\_str: 200만큼의 메모리를 할당해주어 파일의 한 줄을 입력 받아 저장하는 데에 사용된다.
  1. opcmd.c

해당 파일은 opcode 관련 명령어들을 처리하는 파일이다. 프로젝트 1을 구현하기 위하여 주로 사용되지만, 프로젝트 2 중에도 OPCODE와 관련하여 수행하여야 하기 때문에 해당 파일에 모듈을 추가하였다.

* + 1. int searchformat(int key, char\* instruction)

1. 기능

hash table에서 주어진 key로 주어진 mnemonic의 format을 반환한다. format이 1이면 1, 2이면 2, 3/4이면 3을 반환한다. format이 4인지는 finObj() 모듈에서 확인하여 적절히 사용된다.

1. 사용 변수

* Opcode\* cur: Opcode hash Table의 [key]번째 list의 구조체 포인터
  1. sicxecmd.c

해당 파일은 프로젝트 2에서 새롭게 구현해야하는 명령어들을 처리하기 위하여 만들어졌다. assemble filename이 입력되면 해당 파일 내의 함수들이 사용되어 .asm파일로부터 .lst 파일과 .obj 파일을 생성하게끔 한다. symbol이 입력되면 가장 최근에 assemble된 파일에서 생성된 SYMTAB을 출력하게끔 한다.

* + 1. void insertSymbol(char\* label, int locctr)

1. 기능

SYMTAB linked list에 새로운 symbol과 그에 해당하는 location counter를 SYMTAB 노드에 담아 삽입한다. 노드 삽입은 symbol이 alphabetic한 order로 삽입되게끔 한다.

1. 사용 변수

* SYMTAB\* node: SYMTAB 구조체 포인터로 새로운 노드를 의미한다. parameter로 받아진 label과 locctr를 담아 저장한다.
* SYMTAB\* cur: SYMTAB 구조체 포인터로 SYMTAB의 노드들을 돌며 새로 만들어진 노드가 어디에 들어갈지를 돕는다.
  + 1. int SearchSymbol(char\* label)

1. 기능

parameter로 받아진 symbol이 존재하는 SYMTAB에 존재하는지 확인해준다. 존재한다면 해당 symbol의 location counter를, 존재하지 않는다면 -1을 반환한다.

1. 사용 변수

* SYMTAB\* cur: SYMTAB 구조체 포인터로 SYMTAB의 노드들을 돌며 주어진 symbol을 찾는 데에 쓰인다.
  + 1. int assemble(char\* filename)

1. 기능

input으로 들어온 file이 asm file이 맞는지 확인한다. 아니라면 에러 문구를 출력 후 0을 반환한다. 만약 파일이 존재하지 않는다면 에러 문구를 출력한 후 0을 반환한다. 만약 pass1에서 에러가 나왔다면 해당 파일에서 생성된 intermediate file을 삭제한 후 0을 반환한다. 만약 pass2에서 에러가 나왔다면 intermediate file, .lst file, .obj file을 삭제한 후 0을 반환한다. 만약 두 pass 모두 성공적으로 종료되었다면 intermediate file을 삭제한 후 성공하였다는 문구를 출력한 다음 1을 반환한다.

1. 사용 변수

* FILE\* fp: input으로 들어온 file의 fopen된 파일 포인터를 저장한다.
* unsigned int i: for문을 위한 임시 변수이다.
  + 1. int pass\_1(char\* filename)

1. 기능

input으로 들어온 file을 열어 assembly program source에 코드 상의 오류가 있는지 확인하며 intermediate file에 들어온 filename및 다음 줄부터 line number, location counter, label, opcode, operand에 대한 정보를 줄마다 출력한다. 에러가 나면 line number와 함께 에러 문구를 출력한 후 0을 반환하고, 성공적으로 프로그램이 끝나면 1을 반환한다.

1. 사용 변수

* FILE\* fp: input으로 들어온 file의 fopen된 파일 포인터를 저장한다.
* FILE\* itmd: intermediate file의 fopen된 파일 포인터를 저장한다.
* char asminput[101]: input으로 들어온 file의 한 줄을 저장한다.
* char input\_save[101]: strtok로 asminput의 원형을 해치기 전 사본의 역할을 한다.
* char\* tok\_input[10]: asminput을 strtok한 token의 포인터들을 각각의 char\*에 저장한다.
* char opcode[30]: token들 중 opcode의 역할을 하는 token을 저장한다.
* int size: 생성된 token의 개수를 저장한다.
* int i: for문을 위한 임시 변수이다.
* int locctr: 해당 줄의 다음 줄의 location counter를 저장한다.
* int locnow: 해당 줄의 location counter를 저장한다.
* int err\_line: 해당 줄의 line number를 저장한다. 5씩 증가한다.
* int labelflag: 해당 줄에 label field에 symbol이 존재하는지를 나타낸다.
* int baseflag: 해당 줄의 OPCODE가 BASE인지를 나타낸다.
* int startflag: 프로그램이 시작했는지를 나타낸다.
  + 1. int pass\_2(char\* filename)

1. 기능

intermediate file을 열어 해당 파일에 저장되어있는 정보를 통해 object code를 만들어 .lst file의 각 줄에 line number, (location counter), (label), opcode, (operand), (object code)를 출력한다. object code들은 object code linked list에 location counter및 modification flag와 함께 저장한다. 또한 .obj file에 object code list에 저장된 object code들을 형식에 맞게 출력한다. 중간에 에러가 발생하면 line number와 함께 에러 문구를 출력 후 0을 반환하고, 모듈이 성공적으로 마무리되면 1을 반환한다.

1. 사용 변수

* FILE\* fp: intermediate file의 fopen 파일 포인터를 저장한다.
* FILE\* lstfile: .lst file의 fopen 파일 포인터를 저장한다.
* FILE\* objfile: .obj file의 fopen 파일 포인터를 저장한다.
* char lstfilename[30]: .lst file의 file name을 저장한다.
* char objfilename[30]: .obj file의 file name을 저장한다.
* char intline[150]: intermediate file의 한 줄을 입력 받아 저장한다.
* int linenum: 현재 줄의 line number를 저장한다.
* int locctr: 현재 줄의 location counter를 저장한다.
* char symbol[30]: 현재 줄의 label field에 존재하는 symbol을 저장한다.
* char opcode[30]: 현재 줄의 opcode field에 존재하는 opcode를 저장한다.
* char operand[30]: 현재 줄의 operand field에 존재하는 operand를 저장한다.
* char\* temp[5]: 입력 받은 줄을 tokenize한 후 token들의 포인터를 저장한다.
* int startflag: 해당 프로그램이 시작되었는지를 나타낸다.
* unsigned int i: for문을 위한 임시 변수이다.
* int first\_executable\_instruction: first executable instruction을 저장한다.
* int objcode: 정수형으로 된 object code를 저장한다.
* char char\_objcode: 형식에 맞춰진 object code를 문자열 형식으로 저장한다.
* ObjectCode\* cur: object code list의 노드를 가리킨다. 노드를 돌며 text record 및 modification record를 작성하는 데에 각 노드의 object code와 location counter를 제공한다.
* char record[61]: text record의 각 줄에 출력될 object code들을 출력 전 연속으로 저장한다.
* int record\_len: text record의 각 줄에 출력될 object code의 길이를 저장한다.
  + 1. char\* findObj(char\* operand, char\* opmnemonic, int locctr)

1. 기능

한 줄에 대한 object code를 생성한다. 들어온 operand와 opcode를 바탕으로 format을 결정한 후 각각의 format의 방법에 따라 object code를 문자열로 저장한다. 그런 다음 insertObj()를 통하여 object code를 linked list에 저장한다.

1. 사용 변수

* int pc: program counter의 위치를 저장한다.
* int n,i,x,b,p,e: SIC/XE machine에서 사용되는 레지스터로 0 또는 1을 저장한다.
* int addr: format 3에서는 disp를, format 4에서는 address를 저장한다.
* char\* objcode: 생성된 object code를 저장한다.
* int opcode: 주어진 mnemonic의 opcode를 저장한다.
* int format: 해당 줄의 format을 저장한다. (1~4)
* char new\_opcode[10]: opcode의 앞에 +가 붙어있을 시 +를 제외한 나머지 부분을 저장한다.
* char new\_operand[10]: operand 앞에 특수 문자가 붙어있을 시, 또는 symbol,X로 되어있을 시 해당 부분을 제외한 나머지 부분을 저장한다.
* int j: for문을 위한 임시 변수이다.
* char \*in\_reg[2]: format 2에서 r1과 r2의 포인터를 각각 저장한다.
* char reg\_num[10][2]: SIC/XE machine 에서의 레지스터 이름과 해당 레지스터의 번호를 저장한다.
* int where\_reg1, int where\_reg2: format 2에서 r1과 r2의 레지스터 번호를 저장한다. r2가 없다면 where\_reg2에는 0이 저장된다.
  + 1. void insertObj(char\* res, int locctr, int mod)

1. 기능

주어진 object code와 location counter, modification flag를 하나의 Object Code node에 저장하여 존재하는 object code list의 맨 뒤에 저장한다.

1. 사용 변수

* ObjectCode\* node: 새로 생성할 노드를 가리키는 구조체 포인터이다.
* ObjectCode\* cur: object code list에 존재하는 각 노드를 돌다가 list의 끝을 가리킨다.
  + 1. FreeObj()

1. 기능

object code list에 할당된 메모리를 해제한다. 만약 비어있다면 head node에 할당된 메모리만 해제하고, 그렇지 않으면 하위 노드들을 먼저 메모리 해제 후 head node의 메모리를 해제한다.

1. 사용 변수

* ObjectCode\* cur: list의 하위 노드를 가리키는 포인터이다. list의 끝까지 돌며 해당 노드의 메모리를 할당 해제시킬 수 있게끔 한다.
* ObjectCode\* next: cur의 next node를 가리킨다. cur이 가리키고 있는 노드가 free된 이후에도 다음 노드로 넘어갈 수 있게끔 한다.
  + 1. void ShowSymbol()

1. 기능

존재하는 SYMTAB안의 symbol들을 순서대로 출력해준다. 만약 비어 있다면 비어있다는 문구를 출력해준다.

1. 사용 변수

* SYMTAB\* cur: SYMTAB 구조체 포인터로 SYMTAB의 노드들을 돌며 각각의 노드 안의 location counter와 symbol의 정보를 가져와 출력하는 데에 쓰인다.
  + 1. void FreeSymbol()

1. 기능

존재하는 SYMTAB에 할당된 메모리를 해제해준다. 만약 비어 있다면 head node에 할당된 메모리만 해제해주고, 비어있지 않다면 우선 안의 노드들의 메모리를 할당 해제 시켜준 뒤 head node의 메모리를 할당 해제한다.

1. 사용 변수

* SYMTAB\* cur: SYMTAB 구조체 포인터로 SYMTAB의 노드들을 돌며 해당 노드의 메모리를 할당 해제하게끔 돕는다.
* SYMTAB\* next: SYMTAB 구조체 포인터로 cur의 next node의 포인터를 저장하여 정상적으로 모든 노드의 메모리를 할당 해제하는 데에 쓰인다.

1. 전역 변수 정의
   1. SYMTAB\* symhead

가장 최근에 assemble한 파일에서 intermediate file을 만들며 생성될 symbol과 location counter를 저장하는 SYMTAB의 head node이다. main()에서 해당 node에 메모리를 할당해준 후 next값을 NULL로 초기화 시켜준다.

* 1. ObjectCode\* obhead

가장 최근에 assemble한 파일에서 pass 2를 거치며 생성되는 각 줄의 object code, location counter, modification flag가 저장되는 ObjectCode list의 head node이다. main()에서 해당 node에 메모리를 할당해준 후 next값을 NULL로 초기화시켜준다.

* 1. int prog\_len

가장 최근에 assemble한 파일에서의 program length를 저장한다. program length는 가장 마지막 location counter – start address로 저장한다. 이 과정은 pass 1을 마무리하며 진행된다. 추후 pass 2에서 object file을 작성할 때에 필요하다.

* 1. int start\_addr

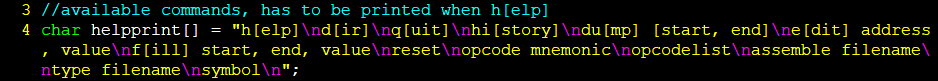
가장 최근에 assemble한 파일에서의 start address를 저장한다. 만약 첫 유효한 opcode가 START라면, 해당 줄의 operand field에 저장된 16진수의 숫자가 start\_addr에 저장된다. 아니라면 start\_addr는 0으로 저장한다. 이는 pass 2에서 object file을 작성할 때 필요하다.

* 1. int base\_addr

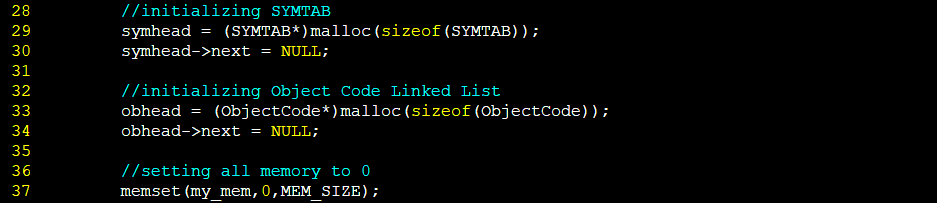
가장 최근에 assemble한 파일에서의 base address를 저장한다. opcode field에 BASE가 왔다면 operand field에 써있는 symbol의 location counter를 SYMTAB에서 찾은 후 base\_addr에 저장한다. 이는 pass 2에서 base relative mode일 때 object code 생성에 필요하다.

* 1. char helpprint[]

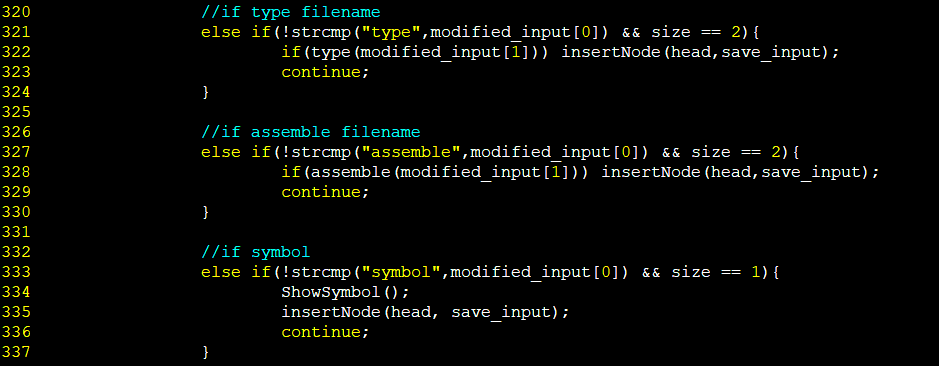
20171662.c의 전역 변수로 h[elp] 명령어가 input으로 들어왔을 시 출력되어야 하는 문자열을 저장한다. 프로젝트 1보다 구현해야 할 명령어가 늘어났으므로 해당 내용 역시 변경되었다.



1. 코드 설명
   1. 20171662.c
      1. main()



전역 변수로 선언된 symhead와 obhead에 메모리를 할당해준 후 next를 NULL로 초기화해준다.

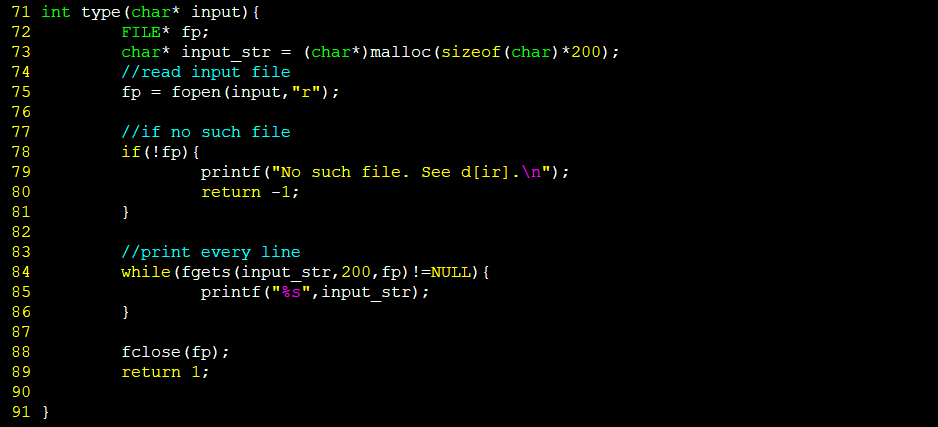


input으로 type filename 형식이 들어왔을 시 type()모듈에 modified\_input[1]을 전달한다. modified\_input[1]에는 두번째 token인 filename이 저장된다. 만약 성공적으로 type()모듈(5.2.1 참고)이 실행되었을 시 1을 반환하므로 그 때엔 insertNode()모듈로 history linked list에 들어온 input을 저장한다.

input으로 assemble filename 형식이 들어왔을 시 assemble()모듈에 modified\_input[1]을 전달한다. 역시 여기엔 filename이 저장되어 있다. 만약 assemble 과정이 성공적으로 진행되었다면 1을 반환하므로 이 때엔 insertNode()모듈로 history linked list에 들어온 input을 저장한다.

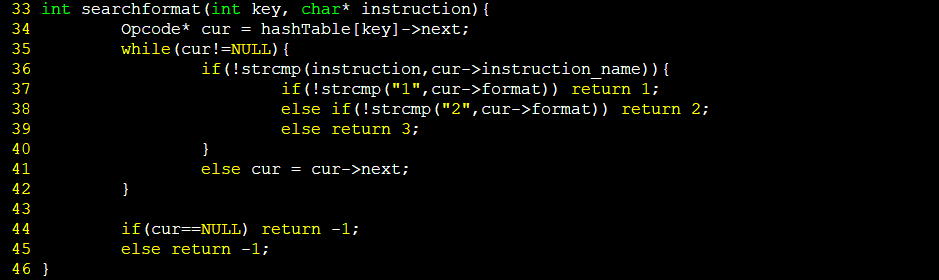
input으로 symbol이 들어왔을 시 ShowSymbol() 모듈을 호출하여 존재하는 SYMTAB의 symbol들을 location counter와 함께 출력한다. 그런 다음 insertNode()모듈로 history linked list에 들어온 input을 저장한다.

* 1. shellcmd.c
     1. type()



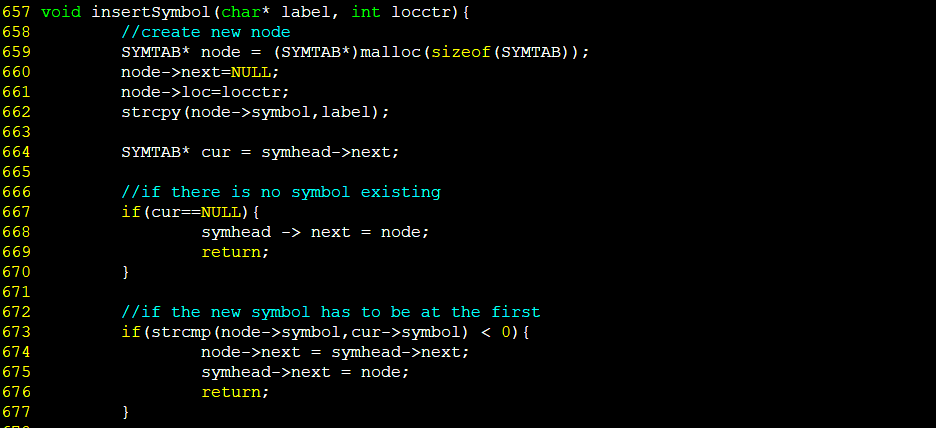
input으로 들어온 file을 연 후 file이 존재하지 않는다면 에러 출력을 한 후 -1을 반환한다. 존재한다면 fgets()를 사용하여 파일의 한 줄을 입력 받은 후 그대로 출력한다. 해당 과정은 파일이 끝날 때까지 반복한다. 다 출력하였으면 파일을 닫고 1을 반환한다.

* 1. opcmd.c
     1. searchformat()

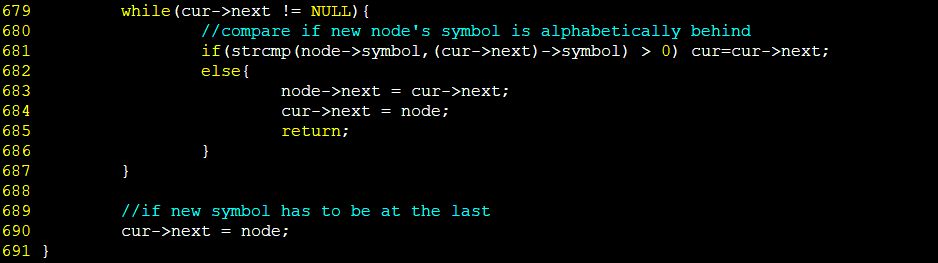


주어진 key값을 index로 하는 hashTable의 linked list의 head node 다음 노드부터 검토한다. 주어진 instruction을 각 노드의 instruction\_name과 비교하며 찾고자 하는 노드를 찾은 뒤 format 부분을 탐색한다. 만약 format 부분이 1이라면 1을, 2라면 2를, 3/4라면 3을 반환한다. format이 4일 경우는 opcode앞에 +가 붙은 경우로 findObj() 모듈에서 찾아 다른 조치를 취한다. 만약 list의 끝까지 탐색하였는데도 instruction이 담긴 노드를 찾지 못하였다면 -1을 반환한다.

* 1. sicxecmd.c
     1. insertSymbol()

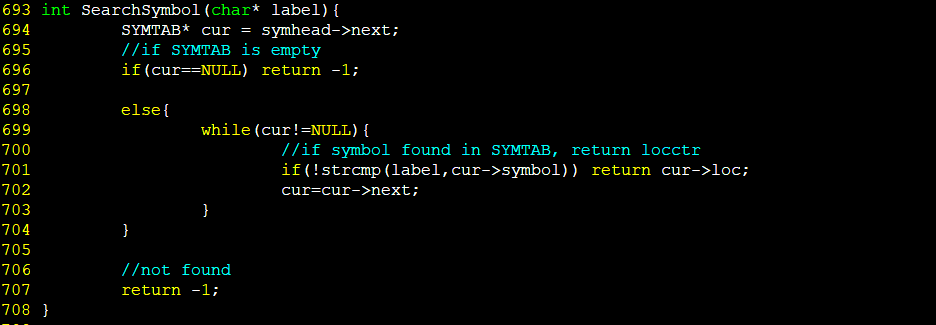


우선 SYMTAB\* node를 새로 생성하여 메모리를 할당해준 후 next, loc, symbol을 모두 주어진 정보로 초기화한다. 그런 다음 SYMTAB을 탐색하며 현재 노드와 주어진 label을 비교한다. 만약 SYMTAB이 비어있다면 head node의 next를 새로 생성된 노드로 저장한다. strcmp(str1,str2)모듈은 두 문자열이 같으면 0을, str1이 str2보다 사전적으로 앞서면 음수를, 사전적으로 뒤에 있으면 양수를 반환한다. 이러한 성질을 이용하여 주어진 label과 head node의 다음 노드를 비교한다. 만약 음수가 나왔다면 주어진 label이 head node의 다음 노드보다 사전적으로 앞선다는 뜻이므로 새로운 노드의 next를 head node의 next노드의 next로, head node의 next를 생성된 node로 설정한다.



그렇지 않다면 다음 노드들을 탐색하며 strcmp()하여 현재 가리키고 있는 노드의 다음 노드보다 주어진 label이 앞서고 현재 가리키고 있는 노드보다는 뒤에 있다면 현재 가리키고 있는 노드의 다음에 새로 생성된 node를 끼워 넣어준다. 만약 노드의 끝까지 다다랐다면 리스트의 맨 마지막에 새로 생성된 노드를 넣어준다.

* + 1. SearchSymbol()

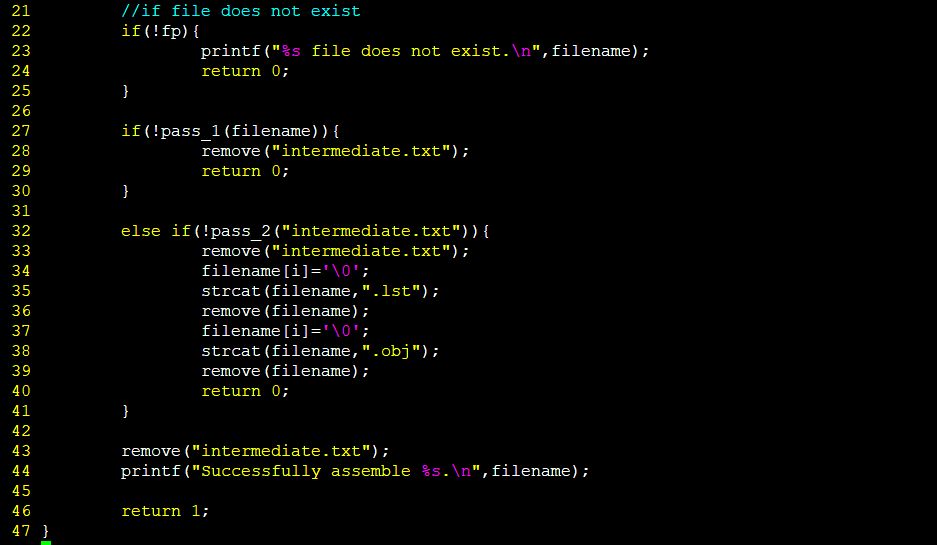


SYMTAB의 head node의 next를 cur로 설정한다. 만약 SYMTAB이 비어 있다면, -1을 반환한다. 그렇지 않다면, 리스트의 끝까지 돌며 label이 SYMTAB안에 저장되어 있는지 각 노드를 살핀다. 만약 주어진 label을 찾았다면, 해당 symbol의 location counter를 반환한다. 만약 리스트의 끝까지 탐색했는데도 찾지 못하였다면 -1을 반환한다.

* + 1. assemble()

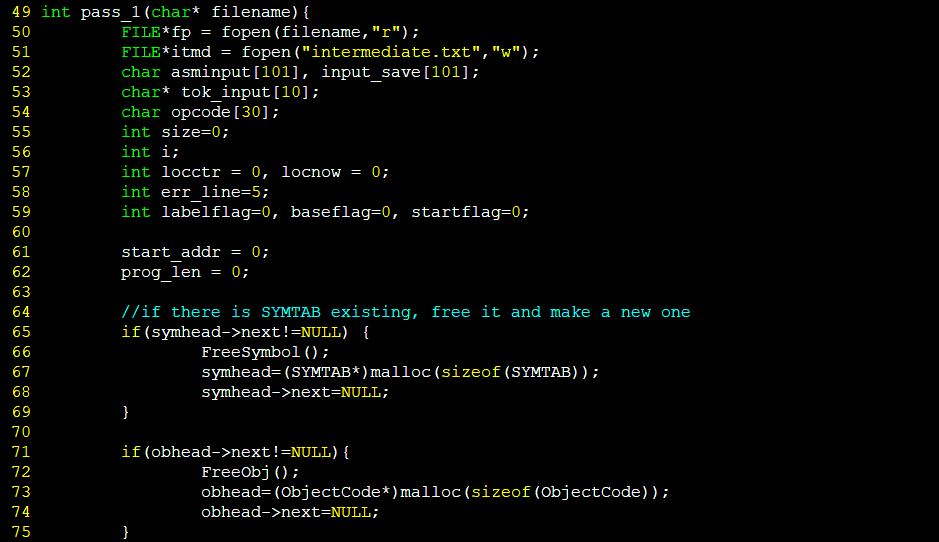


우선 주어진 filename을 가진 파일을 연다. 만약 주어진 파일명이 .asm이 아니라면 에러 문구 출력 후 0을 반환한다.

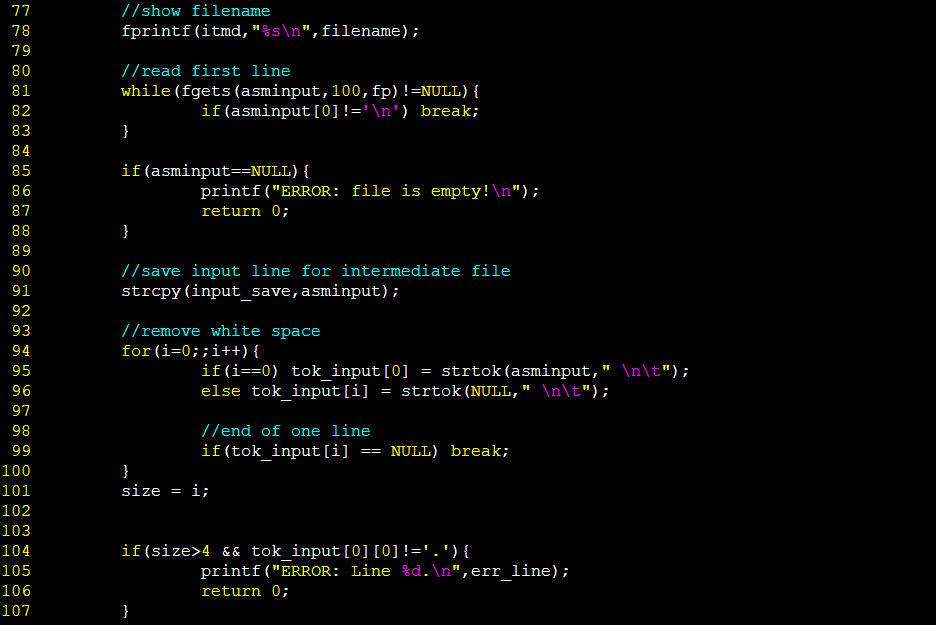


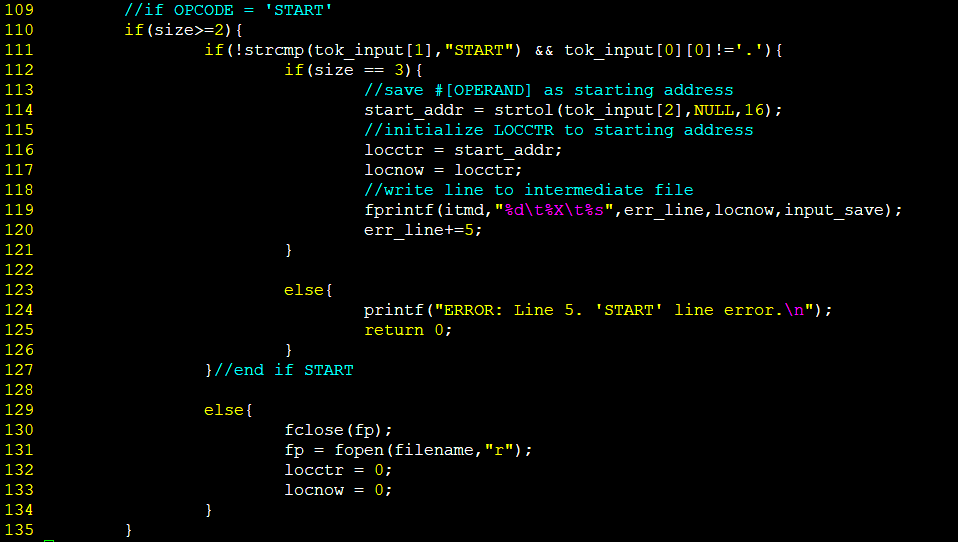
만약 파일이 존재하지 않는다면 에러 문구 출력 후 0을 반환한다. 그런 다음 pass\_1()을 호출한다. 0이 반환되었다면 pass\_1이 성공적이지 못했다는 뜻이므로 intermediate.txt 파일을 지운 후 0을 반환한다. 1이 반환되었다면 pass\_2()를 호출한다. 0이 반환되었다면 pass\_2가 성공적이지 못했다는 뜻이므로 intermediate.txt 파일과 부수적으로 주어진 filename을 기반으로 한 .lst file과 .obj file을 모두 지운 후 0을 반환한다. 만약 1이 반환되었다면 부수적으로 생성된 intermediate.txt를 삭제한 후 성공적으로 진행되었다는 문구를 출력한 다음 1을 반환한다.

* + 1. pass\_1()



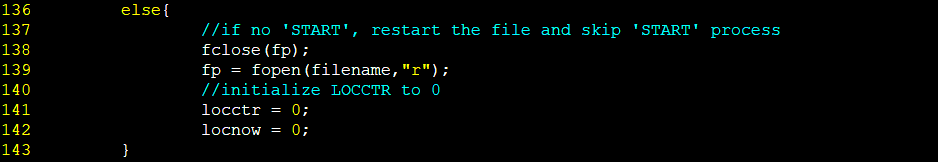
우선 input으로 주어진 file과 intermediate.txt파일을 열고 전역변수로 정의된 start\_addr와 prog\_len을 0으로 초기화해준다. 그런 다음 SYMTAB과 object code list가 존재하는지 확인한다. 만약 해당 파일 이전에 assemble이 이미 진행되었다면 비어있지 않을 확률이 높기 때문이다. 만약 비어있지 않다면 FreeSymbol()과 FreeObj() 모듈을 호출하여 두 리스트 모두 메모리 할당 해제를 한 후 head node만 다시 메모리 할당을 해준 후 next를 NULL로 초기화해준다.



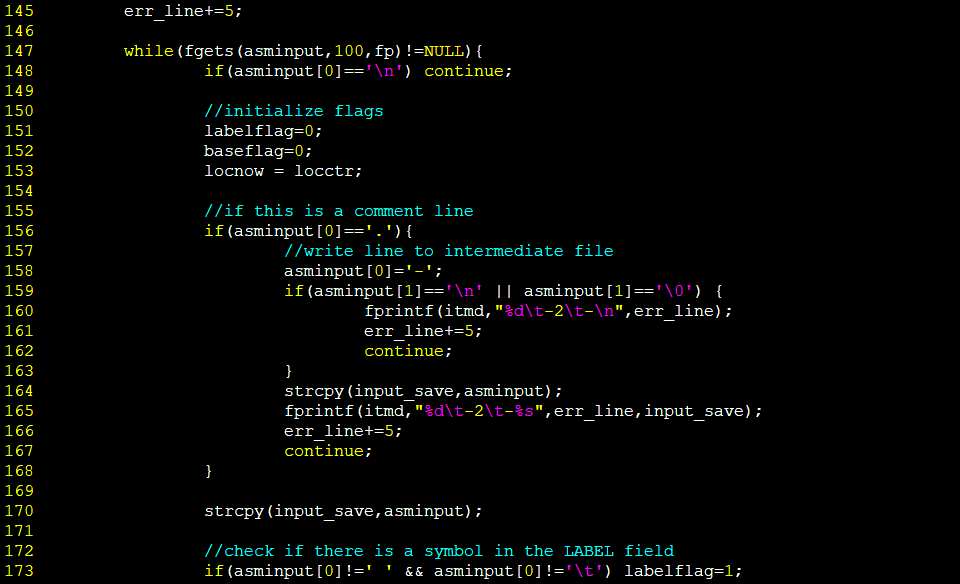


pass 2에서 주어진 파일의 이름에 맞추어 lst file과 obj file의 이름을 생성할 수 있도록 intermediate file 첫 줄에 file name을 출력한다. 그런 다음 첫 줄을 읽는다. 만약 ‘\n’만 들어왔다면 loop에서 나온다. 만약 NULL이 입력 받아졌다면 첫 줄이 없다는 뜻이므로 해당 파일은 비었다는 뜻이 된다. 에러 문구를 출력한 후 0을 반환한다. 그런 다음 tokenize되기 전 원형 보존을 위해 input\_save 변수에 asminput변수의 문자열을 복사하여 저장한다. 그런 다음 strtok를 이용하여 tok\_input[i]에 입력 받은 줄의 token을 저장한다. token의 개수를 size에 저장한다. 만약 size가 4보다 크고 첫 토큰의 첫 문자가 ‘.’이 아니라면 comment line도 아닌데 너무 많은 token이 생성된 것이므로 오류가 생겼다는 것을 의미한다. 에러 문구를 출력 후 0을 반환한다.

만약 size가 2 이상일 때 OPCODE 가 START이고 comment line이 아니라면 size를 확인한다. size가 3이라면 operand field에 저장된 16진수를 start address에 저장한다. 그런 다음 location counter를 start address로 초기화한다. 해당 줄을 intermediate file에 line number, location counter와 함께 출력한다. 그런 다음 line number를 5 증가시킨다. size가 3이 아니라면 에러 문구 출력 후 0을 반환한다. 만약 첫 줄이 comment도 아니고 START도 아니라면 파일을 처음부터 다시 읽게끔 파일을 닫고 다시 연다. 그런 다음 location counter를 0으로 초기화한다.

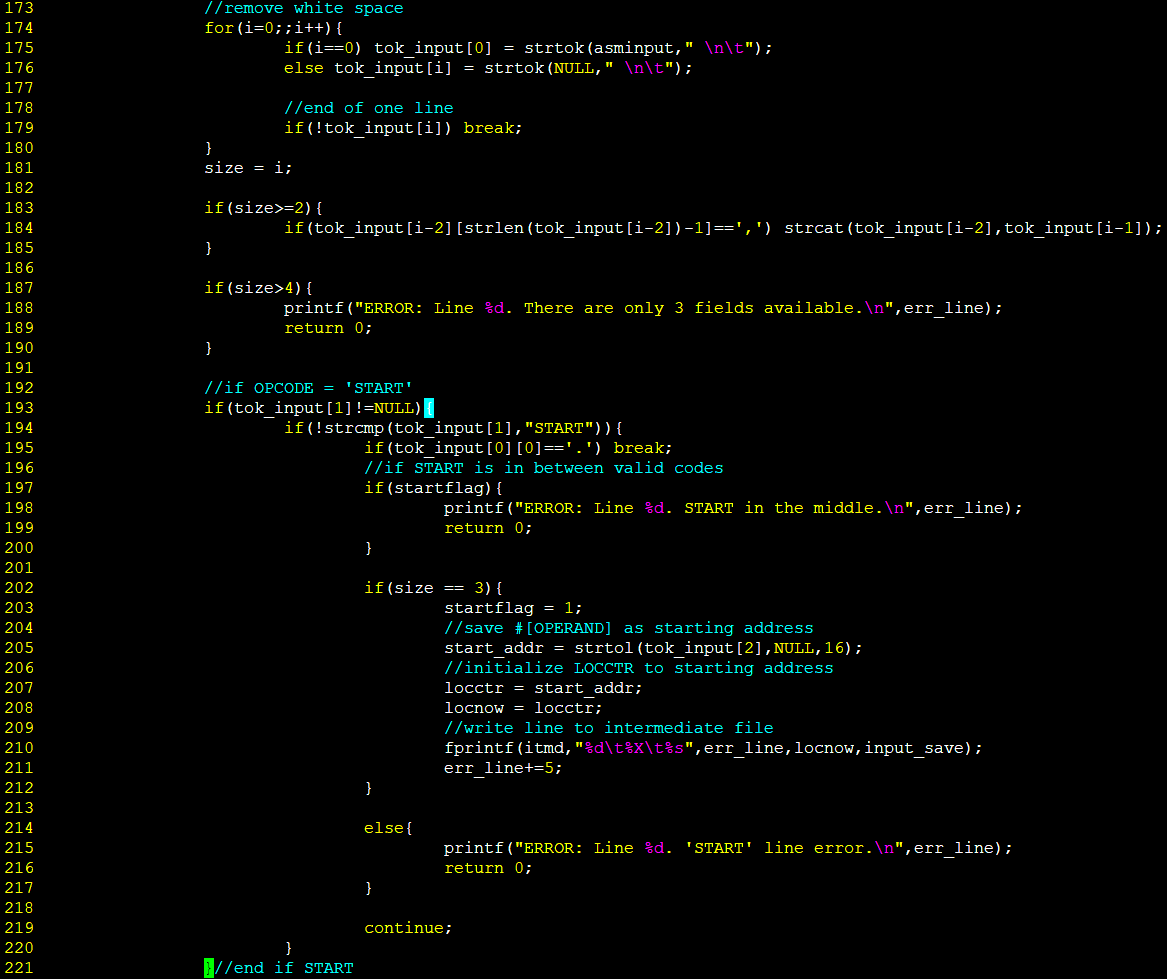


size가 2 이상이 아니라면 역시 START 가 아니므로 파일을 닫고 연 후 location counter를 0으로 초기화해준다.



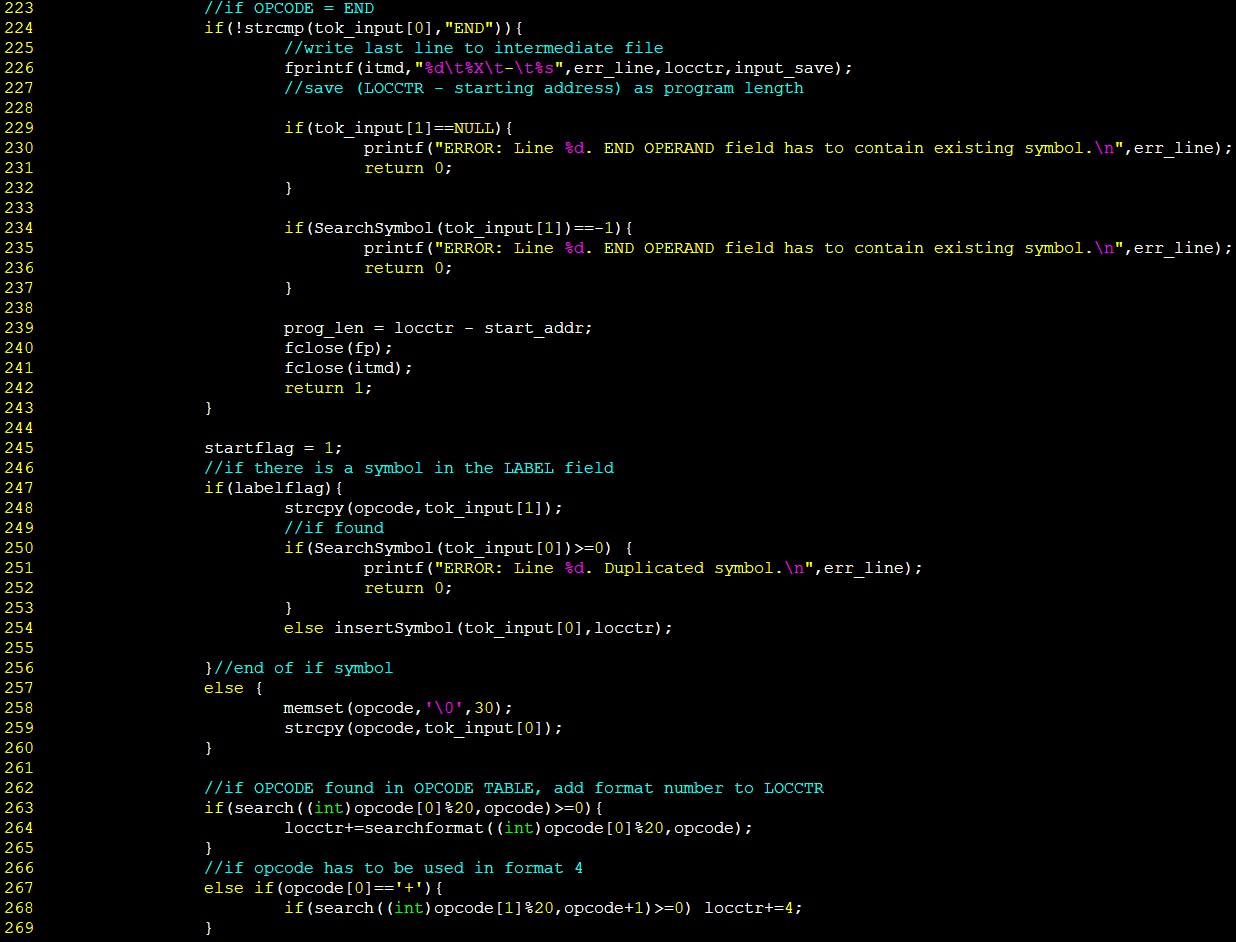
line number를 5 증가시킨 후 다시 파일을 첫 줄부터 읽는다. 만약 첫 문자가 ‘\n’라면 바로 다음 줄을 읽게끔 한다. 그런 다음 labelflag, baseflag를 0으로 초기화해준 후 다음 줄의 location counter를 저장하고 있던 locctr의 값을 현재 줄의 location counter를 저장해야 할 locnow에 옮긴다. 그런 다음 만약 해당 줄이 comment line이라면 해당 줄의 첫 문자를 ‘-‘로 변환해준다. 이는 pass2에서 intermediate file을 읽을 때 label field가 비어 있음을 나타내는 flag로 작용한다. 만약 comment line에 아무 내용이 없다면 intermediate file에 line number, -2, -를 출력한다. location counter 부분을 -2로 출력해줌으로써 해당 줄은 comment line임을 나타내는 것이다. 현재 line number를 저장하는 err\_line의 값을 5만큼 증가시켜준 후 다시 다음 줄을 읽게끔 한다. comment line에 내용이 있다면 input\_save에 asminput을 저장해준 뒤 intermediate file에 line number, -2, -, comment line을 출력한다. -2와 –의 역할은 위와 같다. err\_line의 값을 5만큼 증가시킨 후 다음 줄을 읽는다.

comment가 아니라면 asminput의 원형을 보존하기 위해 input\_save에 asminput을 복사한다. 그런 다음 label field가 비어 있는지 확인한다. 비어있지 않다면 label flag를 1로 설정한다.

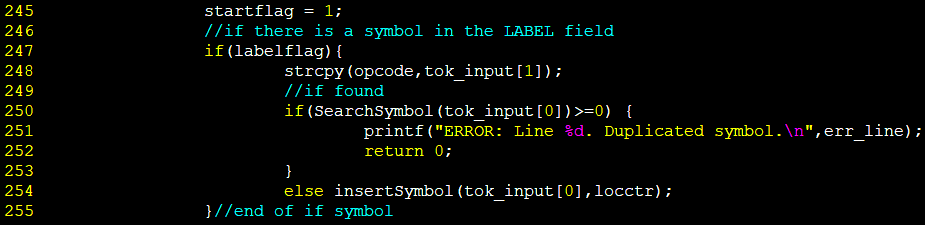


asminput을 빈칸을 기준으로 tokenize 한다. token의 개수는 size에 저장한다. 만약 size가 2보다 크거나 같고, 마지막에서 두번째의 token의 마지막 부분이 ‘,’라면 ,를 포함하는 operand 가 2개로 나뉘었다는 뜻이므로 맨 마지막 token과 마지막에서 두번째의 token을 붙인다. 만약 size가 4보다 크면 코드 상의 오류이므로 에러 문구를 출력 후 0을 반환한다.

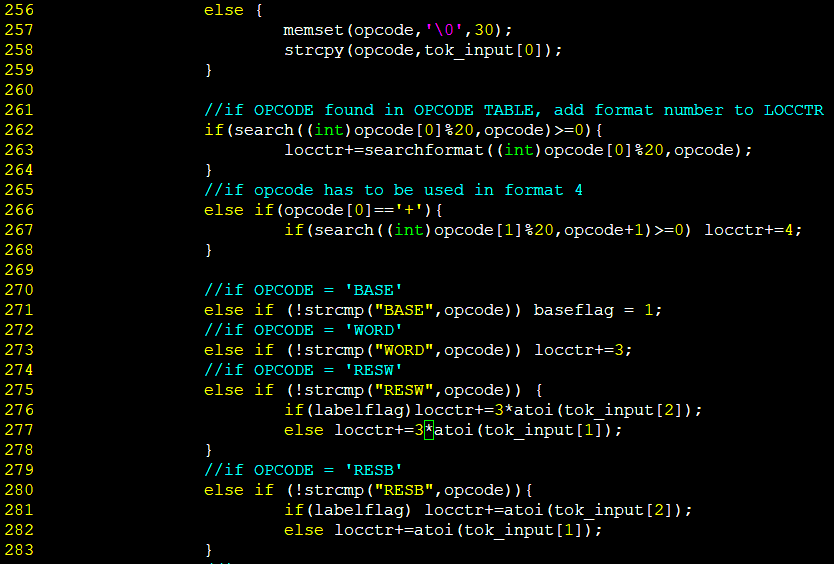
만약 OPCODE가 START라이고 comment line이 아니라면 start flag를 확인한다. 만약 start flag가 1이라면 START가 중간에 들어왔다는 뜻이므로 에러 문구 출력 후 0을 반환한다. 그렇지 않고 만약 size가 3이라면 유효한 줄이라는 뜻이므로 start flag를 1로 설정해준 후 operand field의 16진수 값을 start address에 저장한다. 그런 다음 location counter를 start address로 초기화한다. intermediate file에 line number, location counter, 해당 줄을 출력한 후 err\_line을 5 증가시킨다. 그 다음 줄을 다시 입력 받는다. 만약 size가 3이 아니라면 유효하지 않은 줄이라는 뜻이므로 에러 문구를 출력한 후 0을 반환한다.



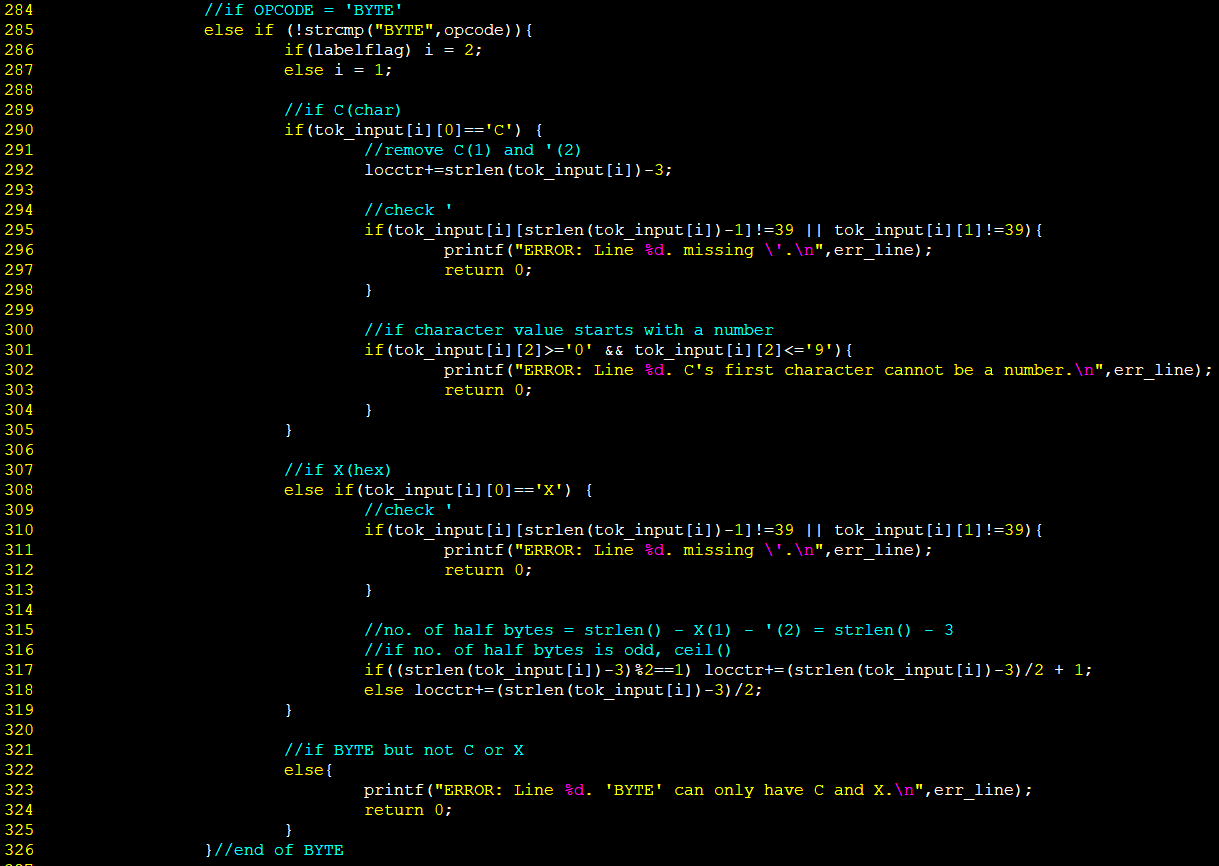
만약 OPCODE가 END라면 intermediate file에 line number, location counter, -, 입력받은 줄을 출력한다. –의 역할은 위와 같이 label field가 비어 있음을 나타낸다. 만약 operand field가 비어있다면 또는 operand field안의 symbol이 SYMTAB에 존재하지 않는 symbol이라면 에러문구를 출력한 후 0을 반환한다. 모두 다 정상이라면 program length를 location counter – starting address로 초기화한 후 열린 파일들을 다 닫는다. 그런 다음 1을 반환한다.



START도 아니고 END도 아니고 comment line도 아니라면 파일이 시작한 것이므로 start flag를 1로 설정해준다. 만약 label field가 비어있지 않았다면 두번째 token이 opcode라는 것이므로 opcode에 두번째 token을 복사한다. 그런 다음 label field에 있는 symbol이 SYMTAB에 있는지 확인한다. 만약 존재한다면 같은 symbol이 두 번 label field에 정의되었다는 뜻이므로 에러 문구를 출력한 후 0을 반환한다. 만약 존재하지 않는다면 insertSymbol()로 SYMTAB에 해당 symbol을 location counter와 함께 저장한다.



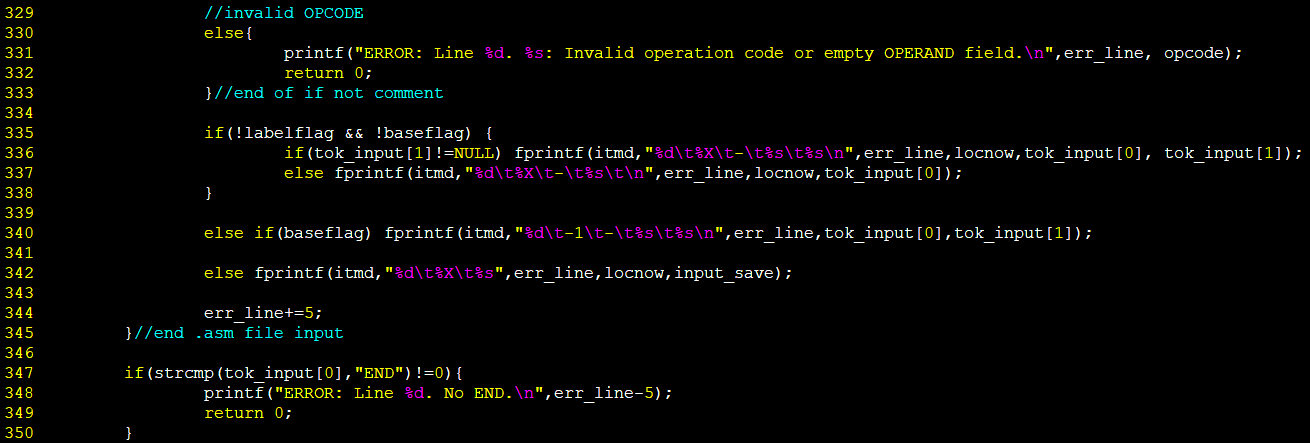
만약 label이 없다면 opcode에 첫번째 token을 복사한다. 그런 다음 opcode가 OPTAB에 존재하는지 search()모듈을 통해 알아본다. 만약 존재한다면 해당 opcode의 format 수만큼 location counter를 증가시킨다. 만약 존재하지 않는다면 첫 문자가 +로 시작한다면 +이후의 opcode를 search()한다. 만약 존재한다면 format 4라는 뜻이므로 location counter를 4만큼 증가시킨다. 만약 OPCODE 가 BASE라면 baseflag를 1로 설정해준다. 만약 OPCODE가 WORD라면 location counter를 3 증가시킨다. OPCODE가 RESW라면 operand field의 10진수 값에 3을 곱한 값을 location counter에 더한다. 만약 OPCODE가 RESB라면 operand field의 10진수 값을 그대로 location counter에 더한다.



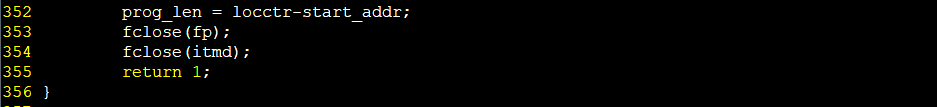
만약 OPCODE가 BYTE라면 operand field의 값의 첫 글자가 C인지 X인지를 확인한다. C라면 character 변수라는 뜻이다. 따라서 문자열의 길이를 location counter에 더한다. 그런 다음 따옴표가 앞 뒤로 잘 있는지 확인하고 없으면 에러 문구를 출력 후 0을 반환한다. 따옴표가 잘 존재한다면 첫 글자가 숫자인지 확인한다. 문자열의 첫 글자는 숫자가 오면 안되므로 숫자가 왔다면 에러 문구 출력 후 0을 반환한다.

X라면 16진수 변수라는 뜻이다. 역시 따옴표가 앞 뒤로 잘 있는지 확인하고 없으면 에러 문구 출력 후 0을 반환한다. 16진수 변수는 한 자리당 half byte를 차지한다. 반드시 한 번에 1byte씩 출력되어야 하기 때문에 만약 16진수 숫자의 길이가 홀수이면 byte수는 자릿수/2에 +1을 location counter에 더해 주어야 한다. (X’FF1’이라면 object code는 0FF1이 되어야 함) 짝수라면 자릿수/2를 location counter에 더해 주어야 한다.

만약 C나 X가 아니라면 에러 문구를 출력한 후 0을 반환한다.

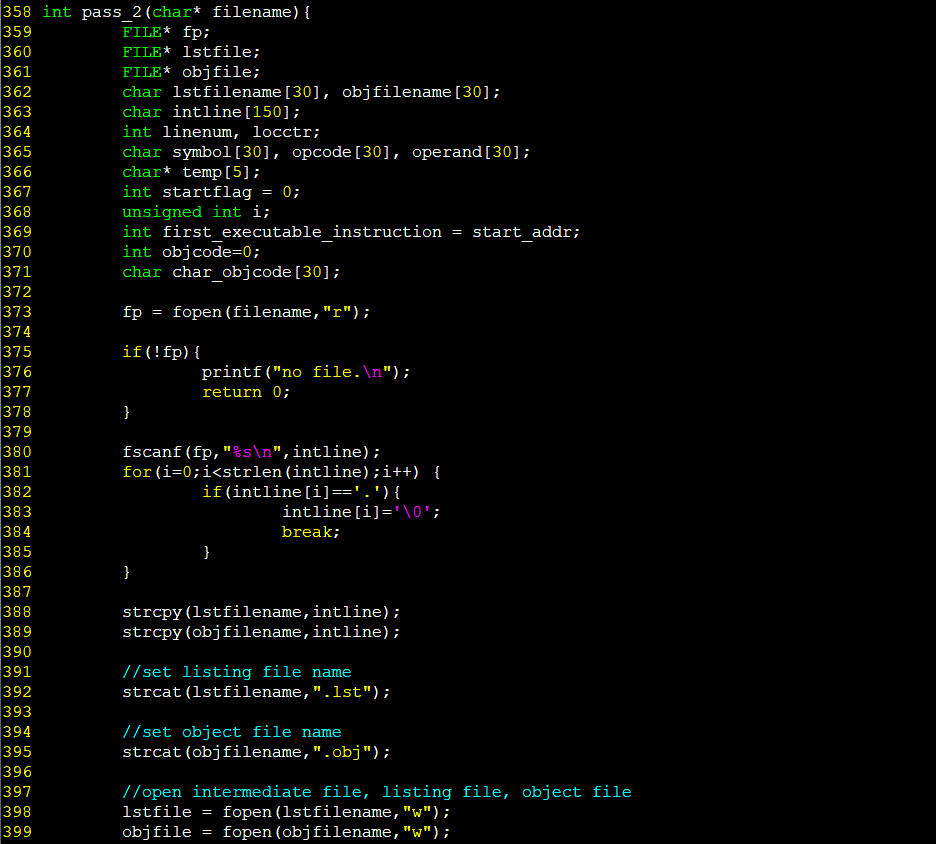


만약 유효하지 않은 OPCODE가 들어왔을 시엔 에러 문구를 출력 후 0을 반환한다. 만약 label field가 비어있고 base line이 아닌 줄이었을 땐 line number, location counter, -, opcode, operand를 intermediate file에 출력한다. –의 역할은 앞서 말한 것과 같다. operand field가 비어 있다면 빈 채로 출력한다. base flag가 1이라면 BASE line 이라는 것이다. 해당 줄은 location counter를 출력하지 않아야 한다. intermediate file에 line number, -1, -, opcode, operand를 출력한다. -1은 pass 2에서 해당 줄이 base line임을 알리는 flag와 같은 역할을 한다. 만약 label field도 존재하고 base도 아니라면 line number, location counter와 입력 받은 줄을 그대로 intermediate file에 출력한다. 그 다음 줄을 다시 입력 받기 전 line number를 5 증가시킨다. 만약 파일이 끝났는데 END가 마지막 opcode가 아니었다면, END로 파일이 종료된 것이 아니므로 에러 문구를 출력 후 0을 반환한다.

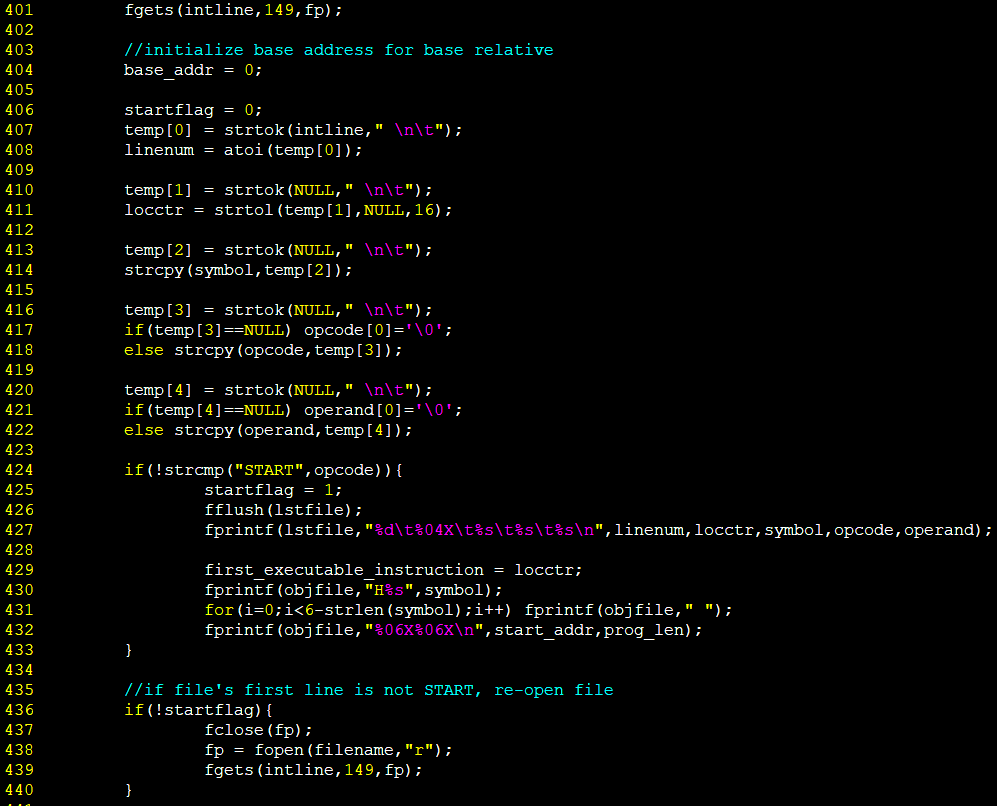


program length를 location counter – start address로 설정한 후 모든 파일을 닫아준다. pass 1이 성공적으로 마무리 되었으므로 1을 반환한다.

* + 1. pass\_2()

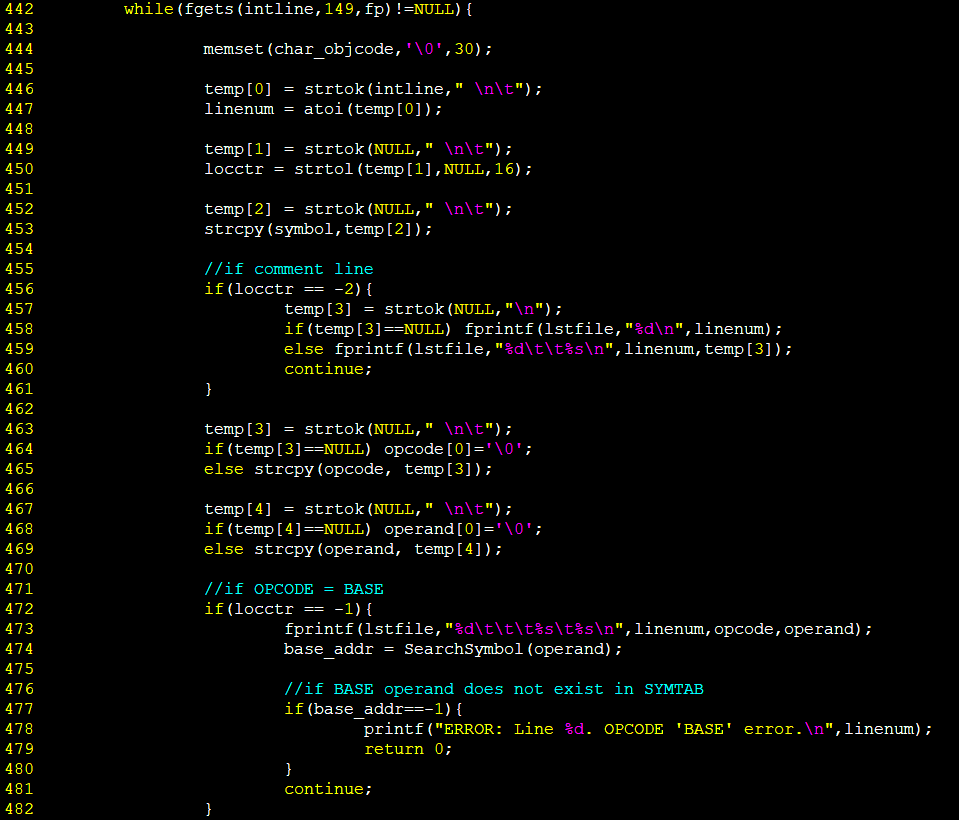


우선 intermediate file을 연다. 만약 파일이 없다면 에러 문구 출력 후 0을 반환한다. 그런 다음 첫 줄에 저장된 .asm file의 이름을 입력 받는다. .asm이전의 파일명을 lstfilename과 objfilename 변수에 복사한다. 그런 다음 각각에 .lst와 .obj를 붙여 파일명을 완성한다. 완성한 파일명을 가진 파일들을 쓰기 전용으로 연다.



그 다음 줄을 입력 받는다. base address 값과 start flag는 0으로 초기화해준다. intermediate file의 결과로는 comment line을 제외하고 5 부분으로 한 줄이 이루어졌을 것이므로 총 5개의 token을 생성한다. 첫 부분은 line number를 의미하므로 linenum변수에 10진수의 형태로 저장한다. 두번째 부분은 location counter를 의미하므로 locctr변수에 16진수의 형태로 저장한다. 세번째 부분은 label을 의미하므로 symbol에 세번째 부분을 복사한다. 네번째 부분은 opcode를 의미하므로 opcode변수에 네번째 부분을 복사한다. 다섯번 째 부분은 operand를 의미하므로 operand변수에 다섯번째 부분을 복사한다. 네번째와 다섯번 째 부분은 NULL값이 들어왔다면 opcode 변수와 operand 변수의 첫 문자를 ‘\0’으로 설정해준다.

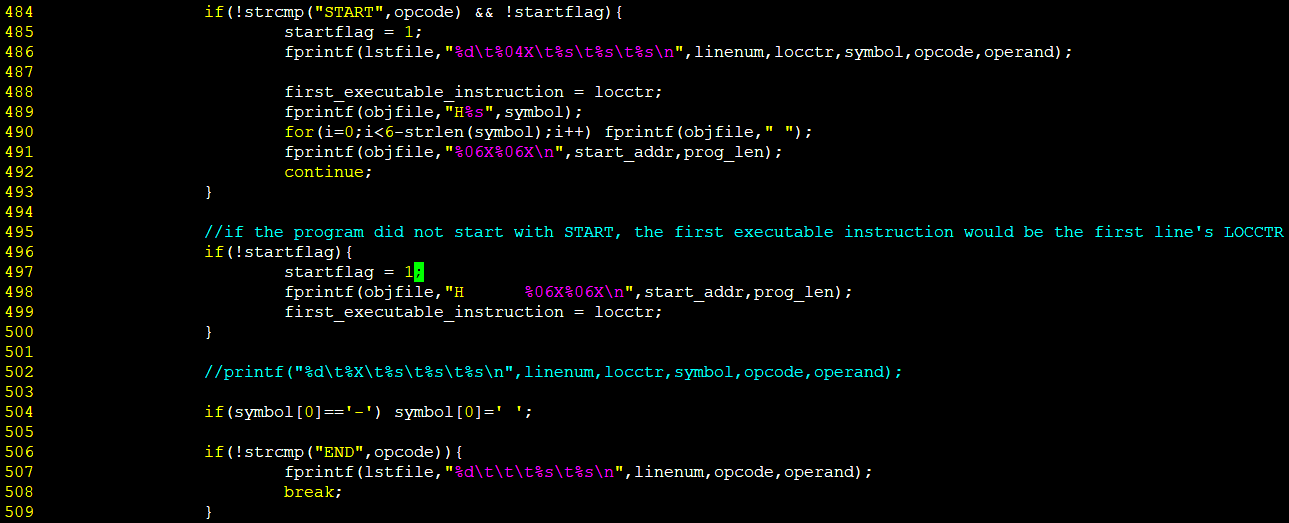
만약 첫 줄에 START line이 들어왔다면 start flag를 1로 설정해준 후 .lst file에 line number, location counter, symbol, opcode, operand를 모두 출력해준다. 그런 다음 first executable instruction code를 location counter로 설정해준다. 입력 받은 symbol이 프로그램의 이름이므로 object code의 header record에 H+symbol+start address+program length를 출력한다. 첫 줄에 START가 들어오지 않았다면 intermediate file을 닫고 다시 열어 파일명을 저장하고 있는 첫 줄을 다시 읽는다.



그런 다음 파일을 끝날 때까지 한 줄씩 읽는다. 우선 object code를 저장하는 char\_objcode를 초기화한 후 읽어온 한 줄을 3번째 부분까지 tokenize한다. 만약 location counter가 -2라면, 해당 줄은 comment line이라는 뜻이다. 만약 빈 줄이라면 .lst file에 line number만 출력하고 다음 줄로 넘어가고, 빈 줄이 아니라면 .lst file에 line number와 해당 줄을 출력한 후 다음 줄을 읽는다.

comment가 아니라면 다음 두 부분까지 tokenize한다. 위와 마찬가지로 첫 부분은 line number를 의미하므로 linenum변수에 10진수의 형태로 저장한다. 두번째 부분은 location counter를 의미하므로 locctr변수에 16진수의 형태로 저장한다. 세번째 부분은 label을 의미하므로 symbol에 세번째 부분을 복사한다. 네번째 부분은 opcode를 의미하므로 opcode변수에 네번째 부분을 복사한다. 다섯번 째 부분은 operand를 의미하므로 operand변수에 다섯번째 부분을 복사한다. 네번째와 다섯번 째 부분은 NULL값이 들어왔다면 opcode 변수와 operand 변수의 첫 문자를 ‘\0’으로 설정해준다.

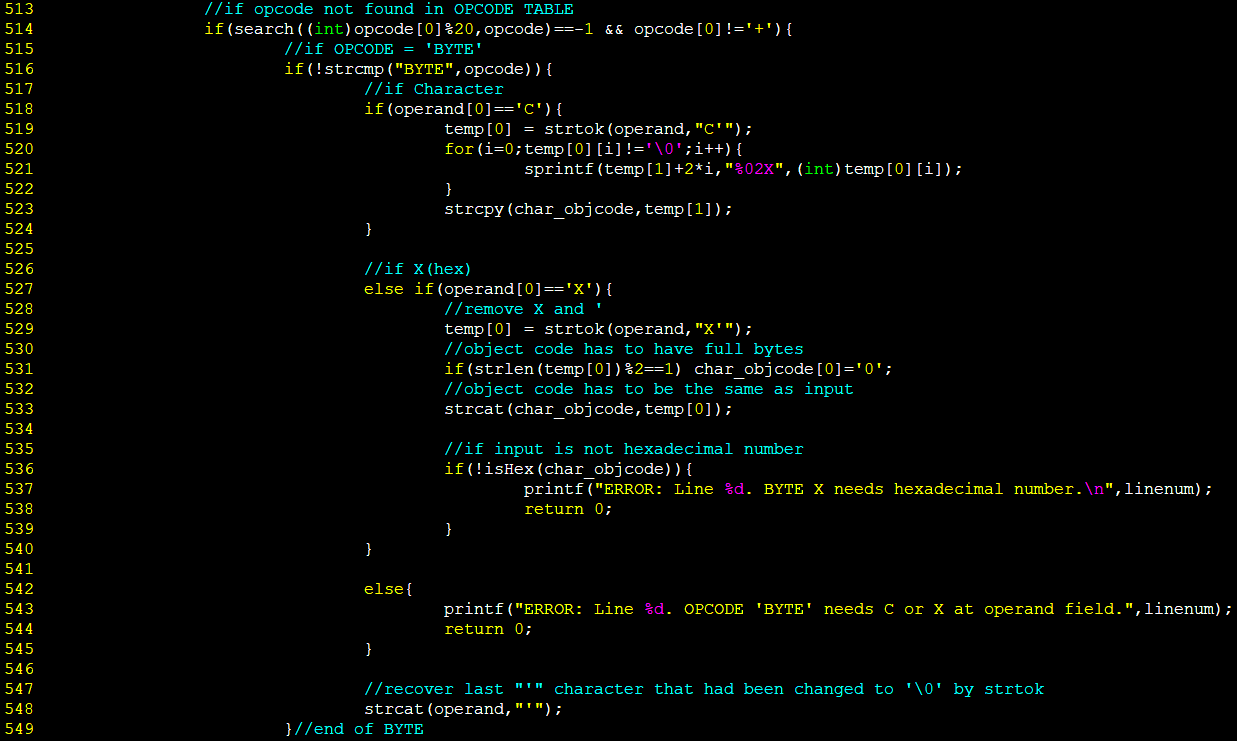
만약 location counter가 -1이라면 BASE 라는 뜻이므로 .lst file에 line number, location counter 부분은 건너뛰고 opcode, operand를 출력한다. 그런 다음 base address에 operand field에 있는 symbol의 location counter를 SearchSymbol()을 이용하여 찾아 저장한다. 만약 -1이 저장되었다면 존재하지 않는 symbol이라는 뜻이므로 에러 문구 출력 후 0을 반환한다. 성공적으로 저장되었다면 다음 줄을 읽는다.



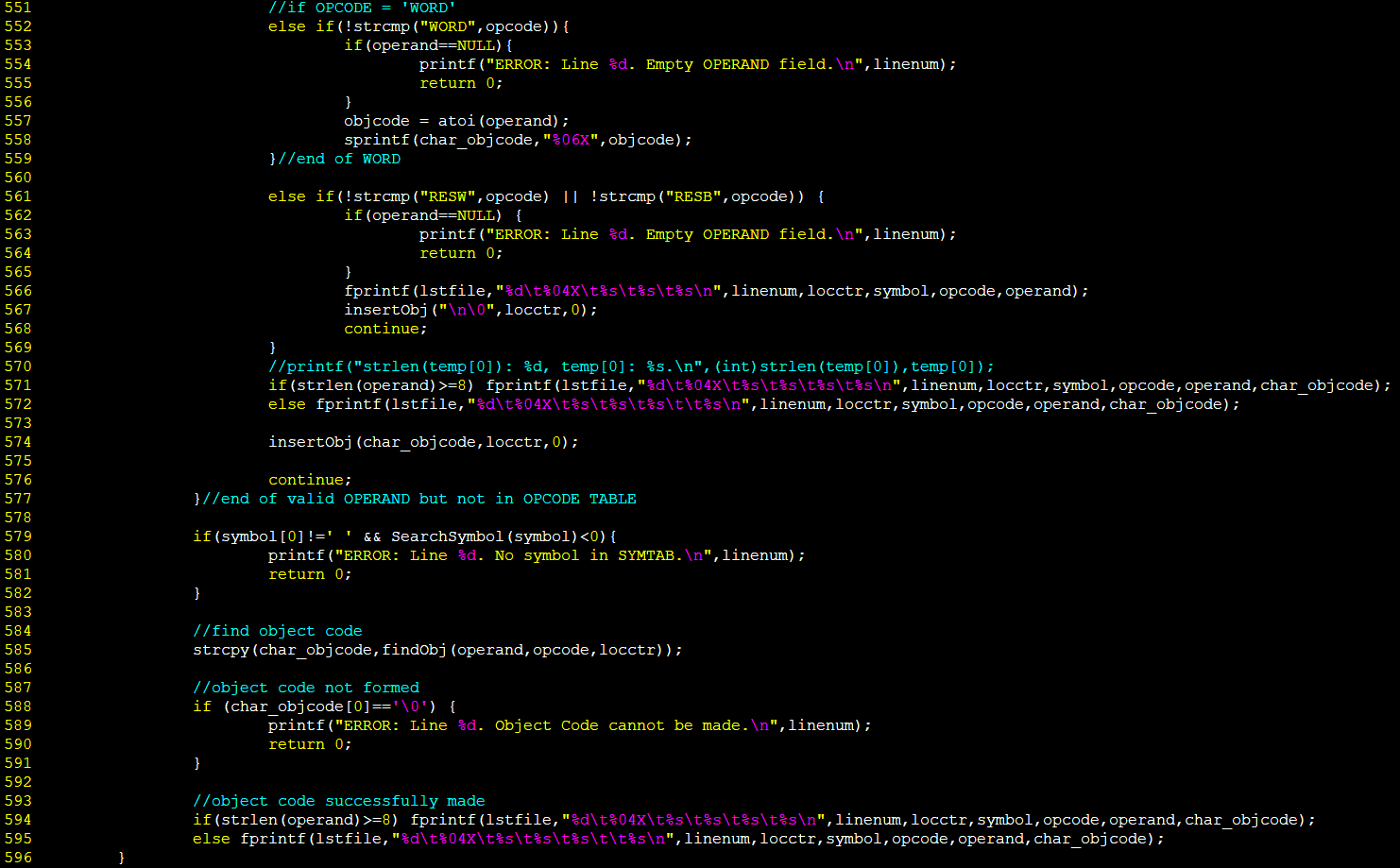
만약 startflag가 1이 아니고 START 가 opcode로 들어왔다면 start flag를 1로 설ㅈ렁해준 후 .lst file에 line number, location counter, symbol, opcode, operand를 출력한다. first executable instruction은 location counter로 설정해준다. 그런 다음 앞서 언급한 바와 같이 .obj file에 header record를 작성한 후 다음 줄을 읽는다. 만약 START가 들어오지 않았다면 start flag를 1로 설정한 후 first executable instruction 을 location counter로 설정해준다. header record도 프로그램명 부분을 빈칸으로 두고 작성한다.

symbol 부분에 –가 들어있다면 symbol 부분이 비었다는 것을 알고 이를 ‘ ‘로 변경해준다. lst file 출력할 때에는 빈 칸으로 출력해야하기 때문이다.

만약 END가 opcode로 들어왔다면 lst file에 line number, location counter와 symbol은 건너뛰고 opcode, operand를 출력한다. 그런 다음 파일을 그만 읽는다.



만약 opcode가 앞에 +가 붙어있지 않고 OPTAB에 존재하지 않는다면 우선 BYTE인지 확인해준다. BYTE에 operand 변수가 C로 시작한다면 문자열임을 의미한다. strtok로 문자열만 분리하여 temp[0]에 저장한다. object code에는 각각의 문자의 ASCII 코드 값을 2자리의 16진수 형태로 순서대로 저장하여야 한다. 이를 char\_objcode에 저장한다. 만약 X로 시작한다면 16진수임을 의미한다. 역시 strtok로 숫자만 분리하여 temp[0]에 저장한다. 만약 해당 16진수의 자릿수가 홀수라면, 앞에 0을 붙여 object code에 저장한다. 만약 분리된 숫자가 16진수의 형태가 아니라면, 에러 문구를 출력 후 0을 반환한다. 만약 첫 문자가 C나 X가 아니라면 에러 문구를 출력 후 0을 반환한다. 그런 다음 strtok로 인해 지워진 따옴표를 다시 operand에 붙여준다.



만약 WORD가 opcode로 들어왔다면 operand 부분을 확인한다. 비어 있다면 적절하지 못한 코드이므로 에러 문구 출력 후 0을 반환한다. 비어있지 않다면 operand 부분의 10진수 숫자를 objcode int형 변수에 저장 후 해당 수를 6자리 16진수로 변환하여 char\_objcode에 저장한다.

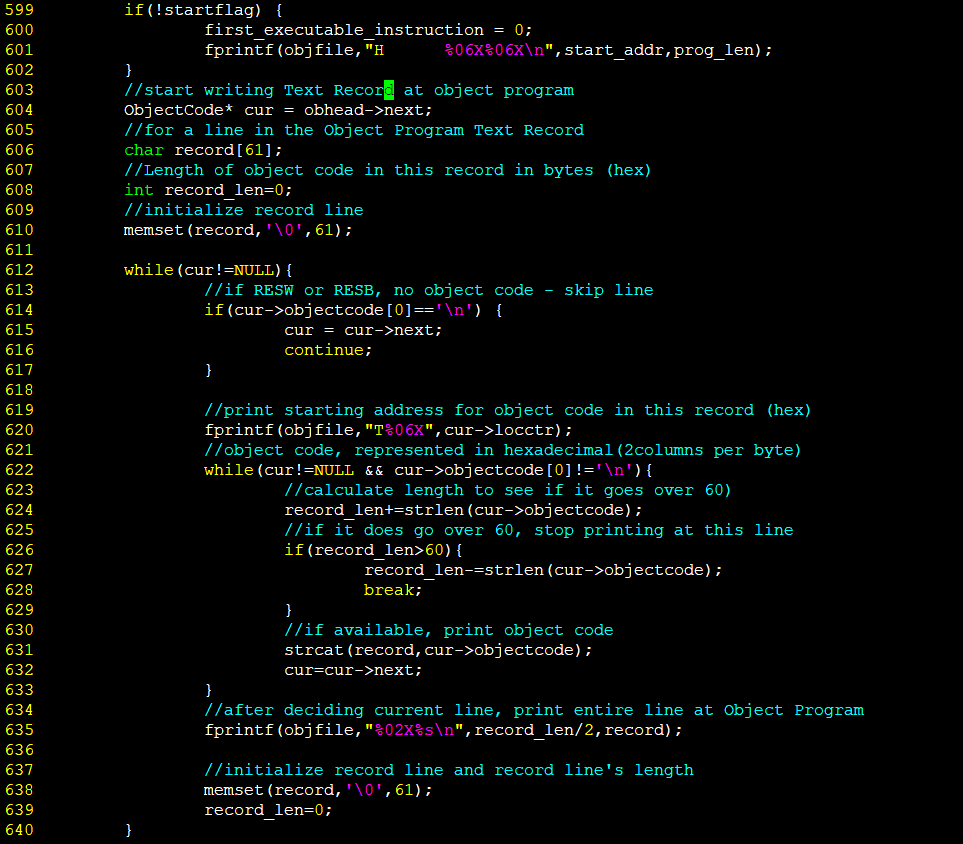
만약 RESW나 RESB가 opcode로 들어왔다면 operand 부분을 확인한다. 만약 비어 있다면 에러 문구를 출력한 후 0을 반환한다. 그런 다음 .lst file에 바로 line number, location counter, symbol, opcode, operand를 출력한 후 object code linked list에 “\n\0”을 저장하게끔 한다. 나중에 object file을 출력 시 변수가 나오면 text record에서 줄 바꿈을 해주어야 하기 때문이다. 위 과정 완료 후 다음 줄을 읽게끔 한다.

WORD나 BYTE의 경우일 땐 .lst file에 line number, location counter, opcode, operand, char\_objcode를 출력한 다음 char\_objcode를 insertObj()를 사용하여 object code linked list에 저장하게끔 한다. 그런 다음 그 다음 줄을 읽게끔 한다.

아예 유효하지 않은 OPCODE의 경우는 pass 1에서 이미 다뤘으므로 해당 과정에서는 생각하지 않는다.

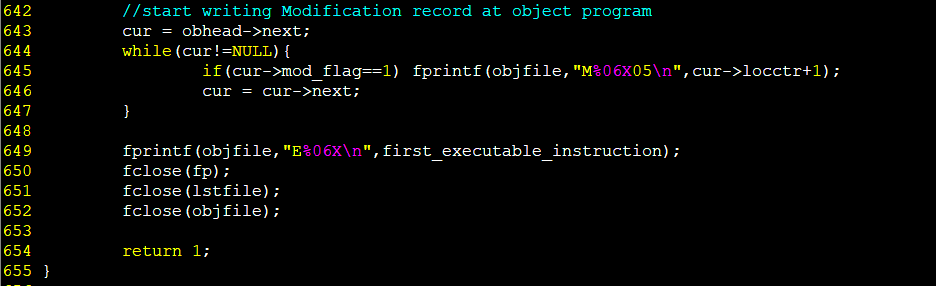
만약 label field가 비어있지 않는데 해당 field안의 symbol이 SYMTAB에 존재하지 않는 경우 역시 오류이므로 에러 문구를 출력한 후 0을 반환한다.

유효한 OPCODE가 들어왔을 경우 해당 과정이 수행된다. 현재 줄의 operand, opcode, location counter를 findObj()모듈에 전달 후 결과 값을 char\_objcode에 저장한다. 만약 NULL이 반환되었다면 object code 생성에 문제가 있었다는 뜻이므로 에러 문구 출력 후 0을 반환한다. object code가 성공적으로 만들어졌다면 .lst file에 line number, location counter, symbol, opcode, operand, object code를 모두 출력한다.



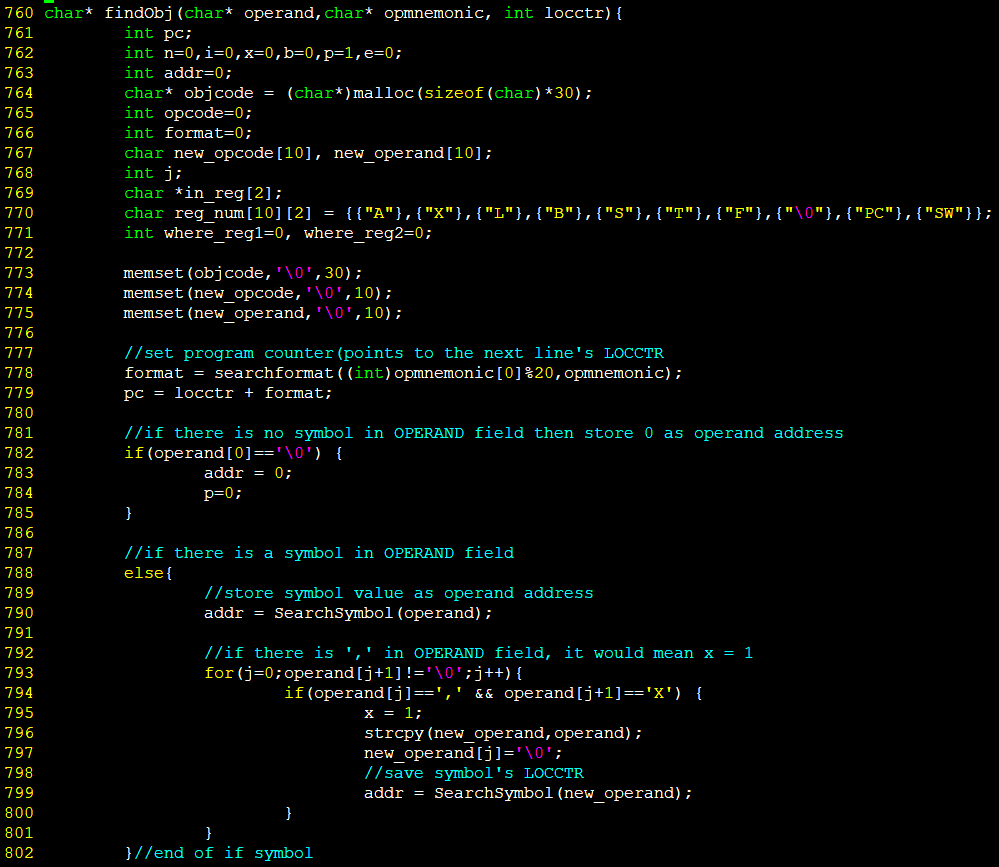
만약 파일 읽기가 끝났는데도 start flag가 0이라는 뜻은 프로그램이 시작을 하지 않고 파일이 끝났다는 뜻이므로 우선 프로그램 명을 빈칸으로 두고 header record를 작성한다.

그런 다음 text record를 작성한다. ObjectCode linked list의 끝까지 돌며 안의 node에 들어있는 object code를 출력한다. 우선 해당 줄이 RESW나 RESB라면 변수이므로 다음 노드로 넘어간다. 유효한 object code가 처음으로 나오면 T+location counter를 먼저 object file에 출력한다. 그런 다음 각 노드에 들어있는 object code를 탐색하며 record\_len에 object code의 길이를 더한다. 해당 길이가 60이 되기 전까지 record 문자열에 object code를 붙여준다. 60이 넘어가면 마지막 object code는 붙이지 않고 object file에 record\_len+record를 출력한다. 그런 다음 record\_len을 0으로 초기화하고 record를 ‘\0’로 초기화한 후 다시 다음 노드들을 탐색하며 위와 같은 과정을 반복한다.



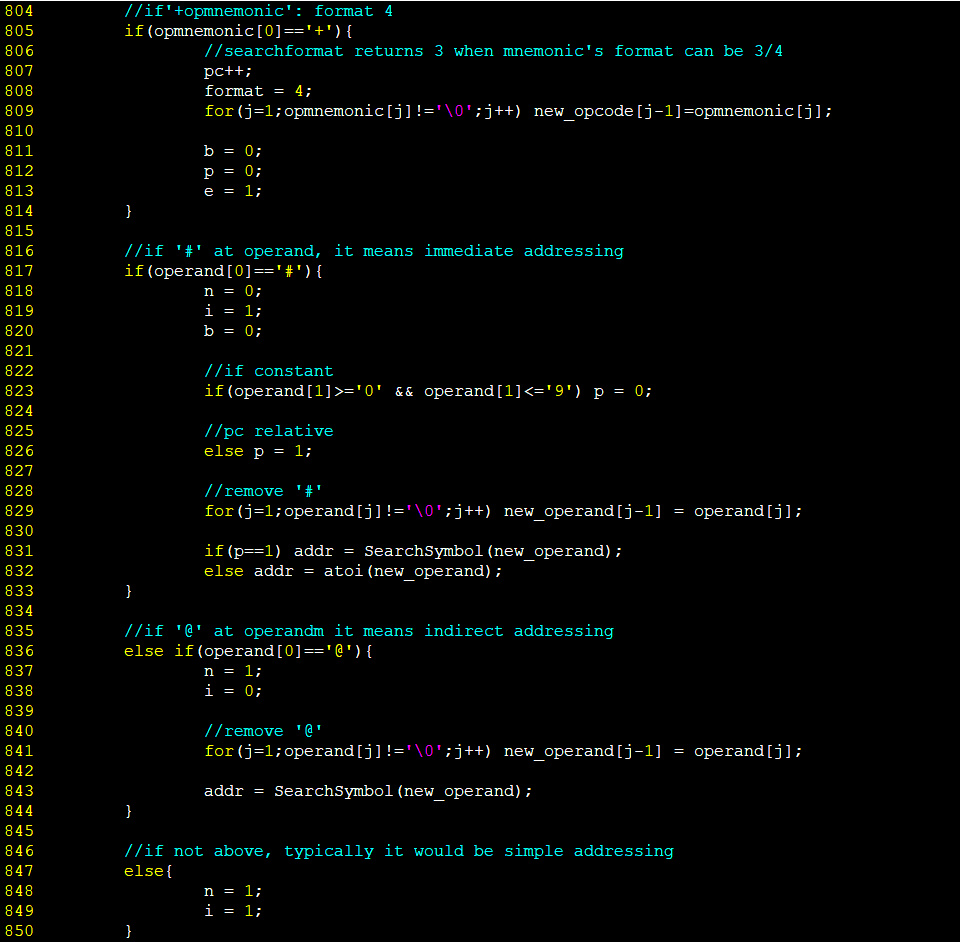
그런 다음 modification flag가 1로 설정되어 있는 object code들만 modification record에 출력해준다. 모두 출력 후 end record에 E+first executable instruction을 출력해준 후 모든 파일을 닫고 1을 반환해준다.

* + 1. findObj()



우선 n,i,x,b,p,e 레지스터 값을 0,0,0,0,1,0으로 초기화해준다. objcode, new\_opcode, new\_operand를 초기화해준다. 그런 다음 format 변수에 주어진 opcode mnemonic의 format을 searchformat()을 통하여 저장한다. pc변수는 program counter를 의미하며 현재 location counter에 format을 더한 값으로 설정해준다.

만약 operand field가 비어있으면 disp를 0으로 초기화하고 p=0으로 한다. 비어있지 않다면 해당 operand안의 symbol의 location counter를 disp값으로 저장해준다. 만약 ‘,X’가 operand 에 있다면 x = 1로 설정해준 후 new\_operand에 해당 부분 이전까지의 symbol을 저장해준다. 그런 다음 disp에 new\_operand의 location counter 값을 SearchSymbol()을 통하여 알아낸 후 저장한다.

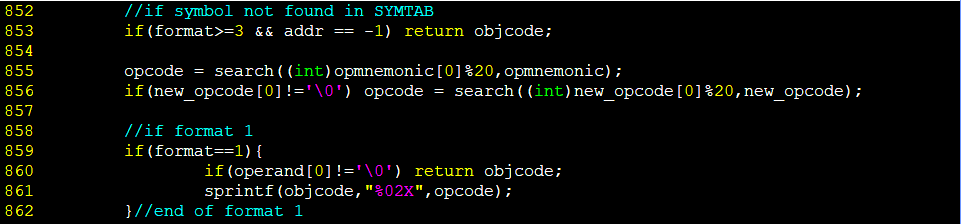


만약 opcode 부분이 +로 시작한다면, format 4임을 의미하므로 program counter를 1 증가시킨다. 원래 searchformat()모듈은 format 부분이 3/4라면 3을 반환하기 때문이다. format 역시 4로 설정해준다. 그런 다음 new\_opcode에 + 다음의 opcode를 저장한다. b=0, p=0, e=1로 설정해준다.

만약 operand 부분이 #으로 시작한다면, immediate addressing을 의미하므로 n=0, i=1, b=0으로 설정한다. #뒤에 상수가 온다면 p=0, 아니라면 p=1로 설정해준다. new\_operand에 #뒤의 값을 저장해준다. #뒤에 상수가 온다면 disp를 상수값 그대로 설정해주고, 아니라면 disp에 new\_operand의 location counter를 SearchSymbol() 모듈을 통하여 설정해준다.

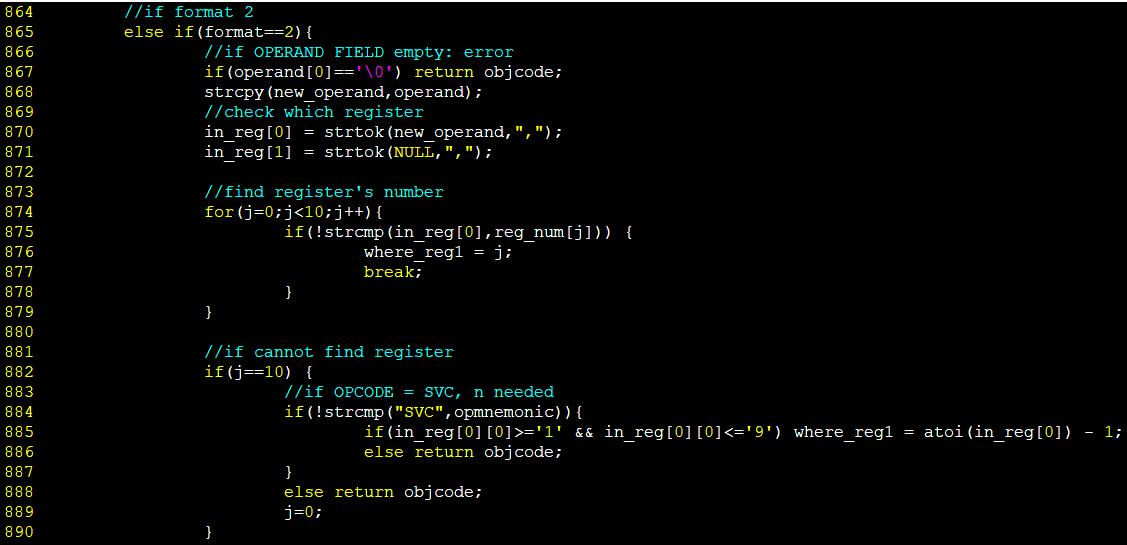
만약 operand 부분이 @으로 시작한다면, indirect addressing을 의미하므로 n=1, i=0으로 설정해준다. 그런 다음 new\_operand에 @뒤의 값을 저장해준다. disp에 new\_operand의 location counter를 SearchSymbol() 모듈을 통하여 설정해준다.

위의 경우가 모두 아니라면 보통 simple addressing이므로 n=1, i=1로 설정해준다.

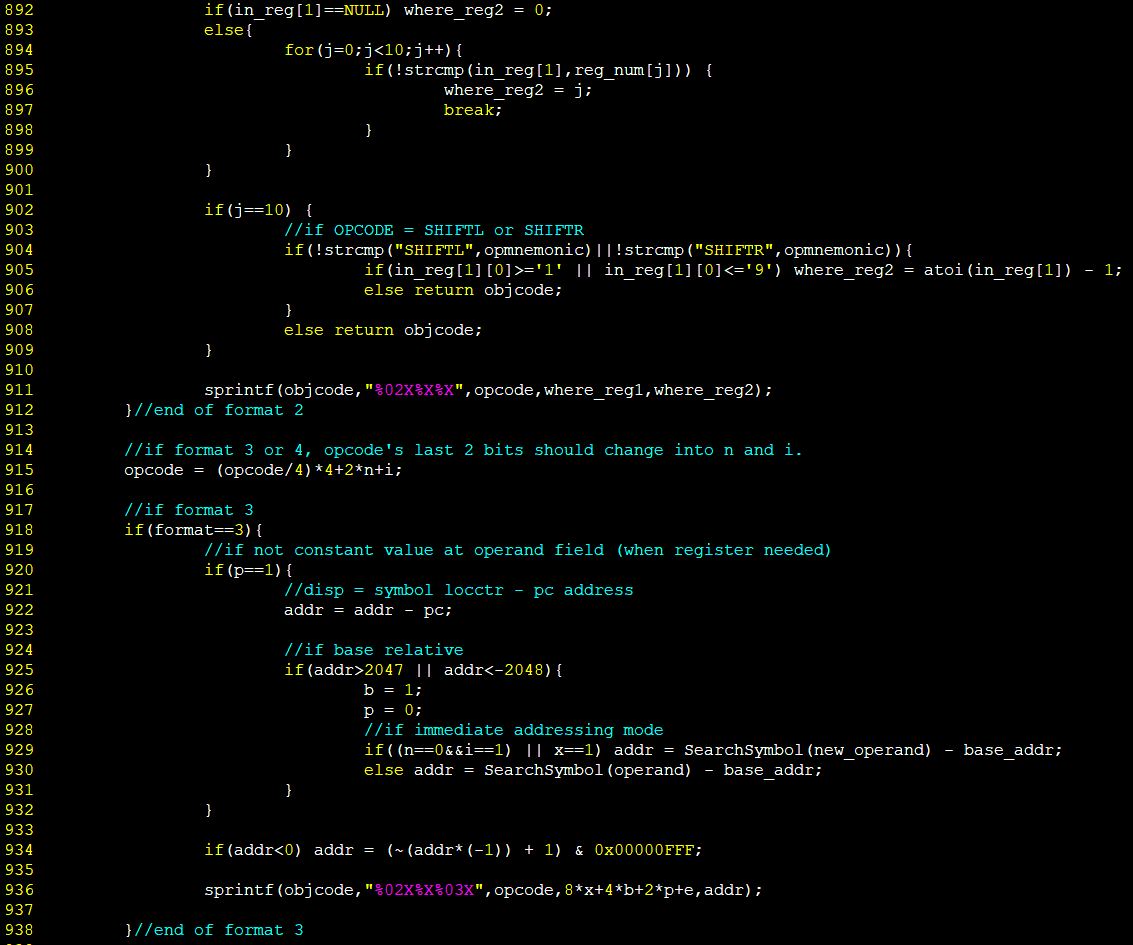


만약 format이 3 이상이라 disp를 써야하는데 disp가 -1이라면 symbol을 찾지 못했다는 것이므로 NULL이 저장되어있는 objcode를 반환한다. opcode에는 opmnemonic의 opcode를 저장한다. 만약 new\_opcode를 생성하였으면 new\_opcode의 opcode를 opcode에 저장한다.

만약 format 1이라면 operand field가 비어있는지 확인한다. 비어있지 않으면 옳지 않은 형식이므로 NULL이 저장되어있는 objcode를 반환한다. 비어있다면 objcode에 2자리 16진수로 opcode를 저장해준다.



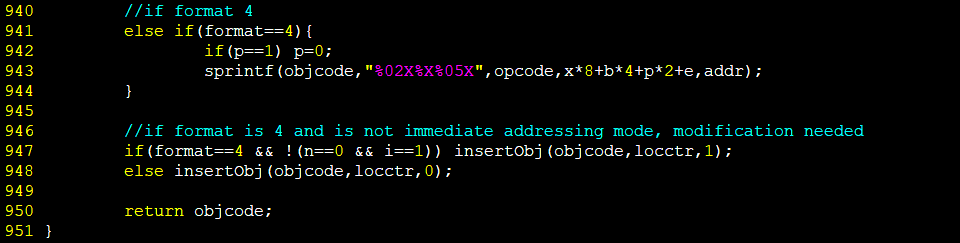
만약 format 2라면 operand field가 비어있는지 확인한다. 비어있다면 옳지 않은 형식이므로 NULL이 저장되어있는 objcode를 반환한다. 비어있지 않다면 operand의 원형을 헤치지 않기 위해 tokenize를 하기 위하여 new\_operand에 operand를 복사한다. 그런 다은 new\_operand를 ,를 기준으로 나눈다. operand field에는 r1또는 r1,r2의 형식이 들어가 있을 것이다. 각 레지스터의 이름을 저장하고 있는 인덱스를 레지스터의 번호로 저장하고 있는 reg\_num의 값들을 찾아보며 r1자리에 있는 register의 번호를 찾아온다. 그런 다음 해당 번호를 where\_reg1에 저장한다. 만약 찾지 못했다면 OPCODE가 SVC인지 확인한다. SVC는 r1대신 n을 필요로 하므로 만약 숫자가 들어왔을 경우 해당 숫자 -1을 where\_reg1에 저장한다. 만약 여기에도 해당이 안되면 옳지 못한 경우이므로 NULL이 저장되어있는 objcode를 반환한다.



만약 operand field에 딱 하나만의 레지스터가 들어왔다면, where\_reg2=0을 저장해준다. 두번째 레지스터도 들어왔다면, 역시 각 레지스터의 이름을 저장하고 있는 인덱스를 레지스터의 번호로 저장하고 있는 reg\_num의 값들을 찾아보며 r2자리에 있는 register의 번호를 찾아온다. 그런 다음 해당 번호를 where\_reg2에 저장한다. 찾지 못했다면 OPCODE가 SHIFTL인지 SHIFTR인지 확인한다. 해당 OPCODE는 r1, n을 요하기 때문에 두번째 register가 숫자라면 해당 숫자 -1을 where\_reg2에 저장한다. 그렇지 않으면 register가 잘못 들어온 것이므로 NULL이 저장되어있는 objcode를 반환한다. 모든 과정이 정상적으로 수행이 되었다면 objcode에 opcode(2) + where\_reg1(1) + where\_reg2(1) 로 총 4자리의 objcode를 생성한다.

format 3과 4는 opcode의 마지막 2 비트를 n과 i로 변환해주어 쓰기 때문에 opcode의 값을 변경해준다.

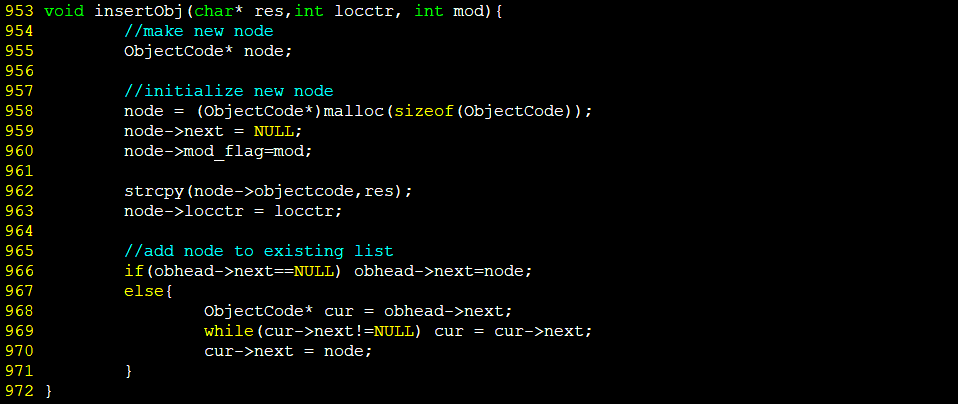
만약 format 3이라면 p = 1인지 확인한다. pc relative라면 disp를 operand의 location counter – pc로 다시 저장해준다. 만약 disp의 범위가 2047보다 크거나 -2048보다 작으면 base relative를 써야 하므로 b=1, p=0으로 설정해준 후 disp를 operand의 location counter – base address로 저장해준다. p가 1이거나 0이거나의 경우에서 만약 addr이 0보다 작은 수이면 2의 보수를 취해준다. 그런 다음 objcode에 opcode(2) + xbpe(1) + addr(3)으로 총 6자리의 objcode를 생성한다.



만약 format 4라면 p가 1로 변경되어 있으면 다시 0으로 맞추어 준다. #옆에 상수였다면 상수 그대로 addr에 저장되어있을 것이고, 그 외의 경우에는 symbol의 location counter로 addr에 저장되어있을 것이다. objcode에 opcode(2) + xbpe(1) + addr(5)로 총 8자리의 objcode를 생성한다. format 4일 때 immediate addressing mode를 제외하고는 modification flag를 1로 설정한다.

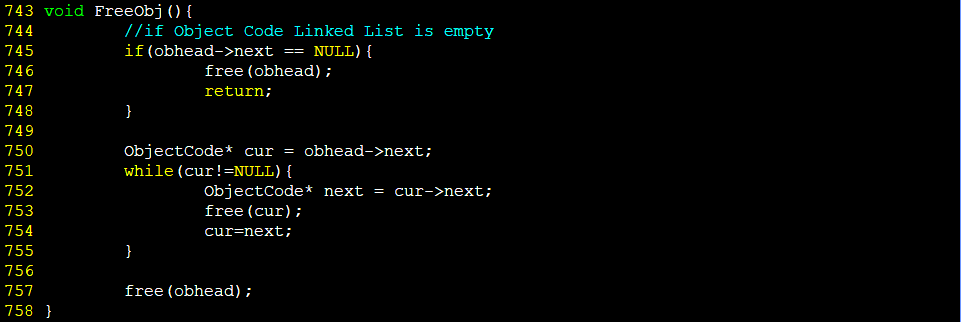
위에서 생성된 objcode를 ObjectCode list에 insertObj()를 통하여 저장한다. 그런 다음 objcode를 반환한다.

* + 1. insertObj()



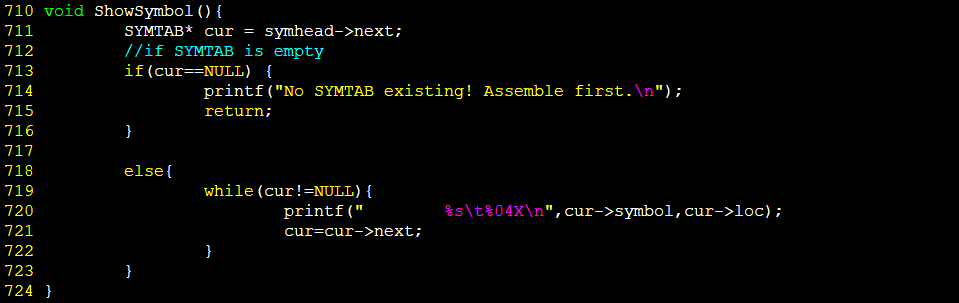
ObjectCode\* 변수 node를 새로 생성하여 주어진 object code, location counter, modification flag를 구조체 안에 저장한다. 그런 다음 head node의 next가 비어있으면 head node의 next에 새로 생성한 node를 저장하고, 그렇지 않으면 cur 변수를 통해 리스트의 맨 마지막에 새로 생성한 node를 저장한다.

* + 1. FreeObj()



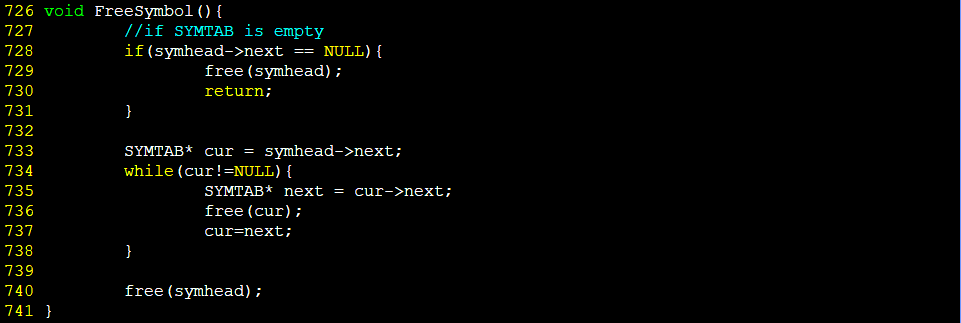
만약 head node의 next가 비어있으면 head node만 메모리 할당 해제한다. 그렇지 않으면 먼저 하위 노드들의 메모리를 해제한 다음 head node의 메모리를 할당 해제한다.

* + 1. ShowSymbol()



만약 SYMTAB이 비어있으면 비어있다는 문구를 출력한 후 프로그램을 종료한다. 만약 비어있지 않다면 리스트의 끝까지 탐색하며 각 노드에 저장되어있는 symbol과 location counter를 출력한다.

* + 1. FreeSymbol()



만약 head node의 next가 비어있으면 head node만 메모리 할당 해제한다. 그렇지 않으면 먼저 하위 노드들의 메모리를 해제한 다음 head node의 메모리를 할당 해제한다.

1. 기타 코드
   1. 20171662.c

#include "20171662.h"

//available commands, has to be printed when h[elp]

char helpprint[] = "h[elp]\nd[ir]\nq[uit]\nhi[story]\ndu[mp] [start, end]\ne[dit] address, value\nf[ill] start, end, value\nreset\nopcode mnemonic\nopcodelist\nassemble filename\ntype filename\nsymbol\n";

int main(){

int i;

//flag to check for too many arguments

int flag=0;

//size of command (no. of arguments, ex. dump 4, 2E: size=3)

int size = 1;

//for saving address from command dump

int startnum, endnum;

//for saving input itself for history (char\* input will be changed due to strtok)

char\* save\_input = (char\*)malloc(sizeof(char)\*100);

//to start with 0 when dump is first called

end\_index = 0xFFFFF;

//initializing head node for history

head = (Node\*)malloc(sizeof(Node));

head->next = NULL;

//initializing hashTable for opcode

hashTable = (Opcode\*\*)malloc(sizeof(Opcode\*)\*TABLE\_SIZE);

init();

//initializing SYMTAB

symhead = (SYMTAB\*)malloc(sizeof(SYMTAB));

symhead->next = NULL;

//initializing Object Code Linked List

obhead = (ObjectCode\*)malloc(sizeof(ObjectCode));

obhead->next = NULL;

//setting all memory to 0

memset(my\_mem,0,MEM\_SIZE);

//importing opcode.txt

FILE\* fp;

char opcode\_input[81];

char\* mod\_opinput[3];

fp = fopen(OPCODE,"r");

while(fgets(opcode\_input,80,fp)!=NULL){

//tokenize opcode

mod\_opinput[0]=strtok(opcode\_input," \n\t");

mod\_opinput[1]=strtok(NULL," \n\t");

mod\_opinput[2]=strtok(NULL," \n\t");

//generate new node & initialize

Opcode\* op\_mem = (Opcode\*)malloc(sizeof(Opcode));

op\_mem->opcodenum = strtol(mod\_opinput[0],NULL,16);

strcpy(op\_mem->instruction\_name,mod\_opinput[1]);

strcpy(op\_mem->format,mod\_opinput[2]);

//setting the index into mod 20

int index = (int)(op\_mem->instruction\_name[0])%20;

add(op\_mem,index);

}//end of file reading

fclose(fp);

while(1){

flag = 0;

//allocate memory for the input variable

input = (char\*)malloc(sizeof(char)\*100);

printf("sicsim> ");

fflush(stdin);

fgets(input,100,stdin);

//printf("%s",input);

if(strlen(input)>100) printf("ERROR: INPUT TOO LONG");

if(input[0]=='\n') continue;

//removing white space

size = 1;

for(i=0;;i++){

//returns the first token, adds '\0' at the end of the token at the string

if (i==0)modified\_input[i] = strtok(input," \n\t");

else modified\_input[i] = strtok(NULL," \n\t");

//if too many arguments

if (i>4) {

printf("ERROR: TOO MANY ARGUMENTS\n");

flag=1;

break;

}

//if end of input

if(!modified\_input[i]) break;

}

size = i;

if(flag) continue;

//for checking ','

if(size>=3){

for(i=1;i<size-1;i++){

if(modified\_input[i][strlen(modified\_input[i])-1]!=','){

printf("ERROR: Invalid Input. See h[elp].\n");

flag=1;

break;

}

else modified\_input[i][strlen(modified\_input[i])-1]='\0';

}

if(flag) continue;

}//end of checking for ','

//saving the input in the right form

strcpy(save\_input,right\_input\_form(size,modified\_input));

if(size==1){

//if q[uit]

if(!strcmp("quit",modified\_input[0]) || !strcmp("q",modified\_input[0])) {

FreeAll(head);

destructor();

FreeSymbol();

FreeObj();

break;

}

//if h[elp]

else if(!strcmp("help",modified\_input[0]) || !strcmp("h",modified\_input[0])) {

printf("%s",helpprint);

insertNode(head, save\_input);

continue;

}

//if d[ir]

else if(!strcmp("dir",modified\_input[0]) || !strcmp("d",modified\_input[0])) {

printdir();

insertNode(head, save\_input);

continue;

}

//if hi[story]

else if(!strcmp("history",modified\_input[0]) || !strcmp("hi",modified\_input[0])){

insertNode(head, save\_input);

ShowAll(head);

continue;

}

}//end of size=1

//if du[mp] [start, end]

if(!strcmp("dump",modified\_input[0]) || !strcmp("du",modified\_input[0])){

switch(size){

case 1:

//boundary check

if(end\_index > 0xFFFFF) end\_index=0xFFFFF;

onlydump();

insertNode(head,save\_input);

break;

case 2:

//check if address is in hex form.

if(isHex(modified\_input[1])==-1){

printf("ERROR: address not in hex form.\n");

continue;

}

//change the starting address from char to hex

startnum = strtol(modified\_input[1],NULL,16);

//boundary check

if(startnum > 0xFFFFF || startnum < 0){

printf("ERROR: Start Address out of range.\n");

continue;

}

dumpstart(startnum);

insertNode(head,save\_input);

break;

case 3:

//check if address is in hex form.

if(isHex(modified\_input[1])==-1 || isHex(modified\_input[2])==-1){

printf("ERROR: address not in hex form.\n");

continue;

}

//change start&end address from char to hex

startnum = strtol(modified\_input[1],NULL,16);

endnum = strtol (modified\_input[2],NULL,16);

//boundary check

if(startnum > 0xFFFFF || endnum > 0xFFFFF || startnum < 0 || endnum < 0){

printf("ERROR: Address out of range.\n");

continue;

}

else if(startnum > endnum){

printf("ERROR: Range ERROR\n");

continue;

}

dumpstartend(startnum,endnum);

insertNode(head,save\_input);

break;

default:

printf("ERROR! TOO MANY ARGUMENTS\n");

continue;

//flag = 1;

continue;

}//end of switch

}//end of du[mp] [start, end]

//if e[dit] address, value

else if (!strcmp("edit",modified\_input[0]) || !strcmp("e",modified\_input[0])){

if (size == 3){

//check if address is in hex form.

if(isHex(modified\_input[1])==-1 || isHex(modified\_input[2])==-1){

printf("ERROR: Address or Value not in hex form.\n");

continue;

}

//change address and value from char to hex

int add = strtol(modified\_input[1],NULL,16);

int val = strtol(modified\_input[2],NULL,16);

//boundary check

if (add<0 || add>0xFFFFF) {

printf("ERROR: Address out of range.\n");

continue;

}

//check value (00~FF)

else if(val<0x00 || val>0xFF){

printf("ERROR: Value out of range.\n");

continue;

}

else {

editvalue(add,val);

insertNode(head,save\_input);

continue;

}

}

else{

printf("ERROR: Input has to be e[dit] address, value. See h[elp].\n");

continue;

}

}//end of e[dit] address, value

//if f[ill] start, end, value

else if(!strcmp("f",modified\_input[0]) || !strcmp("fill",modified\_input[0])){

if(size==4){

//check if address or value is in hex form.

if(isHex(modified\_input[1])==-1 || isHex(modified\_input[2])==-1 || isHex(modified\_input[3])==-1){

printf("ERROR: Address or Value not in hex form.\n");

continue;

}

//change start, end address and value from char to hex

int start = strtol(modified\_input[1],NULL,16);

int end = strtol(modified\_input[2],NULL,16);

int value = strtol(modified\_input[3],NULL,16);

//boundary check

if(start<0 || start>0xFFFFF || end<0 || end>0xFFFFF || start>end){

printf("ERROR: Address out of range.\n");

continue;

}

//value boundary check

else if(value<0x00 || value>0xFF){

printf("ERROR: Value out of range.\n");

continue;

}

else{

fillvalue(start,end,value);

insertNode(head,save\_input);

continue;

}

}//end of size==4

else{

printf("ERROR: 4 arguments needed. See h[elp].\n");

continue;

}

}//end of f[ill] start, end, value

//if reset

else if(!strcmp("reset",modified\_input[0]) && size==1){

reset\_mem();

insertNode(head,save\_input);

continue;

}

//if opcode mnemonic

else if(!strcmp("opcode",modified\_input[0]) && size == 2){

//setting key to mod 20 of the first alphabet of the opcode

int key = (int)(modified\_input[1][0])%20;

int result\_opcode = search(key,modified\_input[1]);

if(result\_opcode == -1){

printf("ERROR: CANNOT FIND OPCODE. See opcodelist.\n");

continue;

}

else {

printf("opcode is %X\n",search(key,modified\_input[1]));

insertNode(head, save\_input);

continue;

}

}//end of opcode mnemonic

//if opcodelist

else if(!strcmp("opcodelist",modified\_input[0]) && size == 1){

insertNode(head,save\_input);

show();

continue;

}//end of opcodelist

//if type filename

else if(!strcmp("type",modified\_input[0]) && size == 2){

if(type(modified\_input[1])) insertNode(head,save\_input);

continue;

}

//if assemble filename

else if(!strcmp("assemble",modified\_input[0]) && size == 2){

if(assemble(modified\_input[1])) insertNode(head,save\_input);

continue;

}

//if symbol

else if(!strcmp("symbol",modified\_input[0]) && size == 1){

ShowSymbol();

insertNode(head, save\_input);

continue;

}

//if invalid input

else {

printf("ERROR! Not an available input. see h[elp].\n");

continue;

}

}

return 0;

}

int isHex(char\* hex\_input){

unsigned int i;

//flag to check for non-hex num.

int hex\_flag=1;

for(i=0;i<strlen(hex\_input);i++){

//if each character is 0~9 or a~f or A~F

if((hex\_input[i]>='0' && hex\_input[i]<='9') || (hex\_input[i]>='a' && hex\_input[i]<='f') || (hex\_input[i]>='A' && hex\_input[i]<='F')) hex\_flag=1;

//not a hex num

else{

hex\_flag=-1;

return -1;

}

}

return hex\_flag;

}

//to save the correct form for each command

char\* right\_input\_form(int size, char\*\* m\_input){

char\* ret\_input = (char\*)malloc(sizeof(char)\*100);

int i;

ret\_input[0]='\0';

for(i=0;i<size;i++){

strcat(ret\_input,m\_input[i]);

if(i==0 && size>1) strcat(ret\_input," ");

if(i>=1 && size>i+1){

strcat(ret\_input,", ");

}

}

return ret\_input;

}

* 1. 20171662.h

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <dirent.h>

#define MEM\_SIZE 1<<20//2^20, 1megabyte memory

#define OPCODE "opcode.txt"

#define TABLE\_SIZE 20//hashTable size

//for du[mp]. to save the last address printed

int end\_index;

//for main input. gets the whole line

char\* input;

//to save tokenized input

char\* modified\_input[100];

//to check if input is hex.

int isHex(char\*);

//to change the tokenized input into a single line with correct input form

char\* right\_input\_form(int,char\*\*);

//=========================================

//===========FOR SHELL COMMANDS============

//=========================================

//\*\*\*when d[ir]\*\*\*

//-----------------------------------------

void printdir();

//-----------------------------------------

//\*\*\*when hi[story]\*\*\*

//-----------------------------------------

//Node for saving input

typedef struct Node{

char\* data;

struct Node\* next;

}Node;

//head of Linked List for hi[story]

Node\* head;

//to add new node at the end of Linked List

void insertNode(Node\*, char\*);

//to print all nodes in the history list

void ShowAll(Node\*);

//to free all memory allocated for history Linked List when q[uit]

void FreeAll(Node\*);

//-----------------------------------------

//\*\*\*when type filename\*\*\*

//-----------------------------------------

int type(char\*);

//=========================================

//===========FOR MEMORY COMMANDS===========

//=========================================

//\*\*\*when du[mp] [start,end]\*\*\*

//-----------------------------------------

//memory with size 2^20

int my\_mem[MEM\_SIZE];

//for du[mp]

void onlydump();

//for du[mp] start

void dumpstart(int);

//for du[mp] start, end

void dumpstartend(int,int);

//for printing out memory values of the given address range

void print\_mem(int, int);

//-----------------------------------------

//\*\*\*when e[dit] address, value\*\*\*

//-----------------------------------------

//for e[dit] address, value. changes the value of the given memory address

void editvalue(int,int);

//-----------------------------------------

//\*\*\*when f[ill] start, end, value\*\*\*

//-----------------------------------------

//for f[ill] start, end, value. changes the memory value of given range to new value

void fillvalue(int, int, int);

//-----------------------------------------

//\*\*\*when reset\*\*\*

//-----------------------------------------

//for reset. changes all memory values to 0.

void reset\_mem();

//-----------------------------------------

//=========================================

//===========FOR OPCODE COMMANDS===========

//=========================================

//\*\*\*when opcode mnemonic\*\*\*

//-----------------------------------------

//structure for Opcode hashTable

typedef struct Opcode{

int opcodenum;//opcode

char instruction\_name[10];//instruction name

char format[5];//format number

struct Opcode\* next;//points to the next node (Linked List)

}Opcode;

//hashTable for OPCODE TABLE.

Opcode\*\* hashTable;

//for allocating memory for each hashTable[i]

void init();

//for freeing memory allocated for hashTable when q[uit]

void destructor();

//for searching for opcode of the given instruction

int search(int, char\*);

//for searching for opcode format number

int searchformat(int, char\*);

//for adding node to the hashTable[key]

void add(Opcode\*, int);

//for opcodelist. prints out all contents of the hashTable

void show();

//-----------------------------------------

//=========================================

//========FOR ASSEMBLER COMMANDS===========

//=========================================

//\*\*\*when assemble filename\*\*\*

//-----------------------------------------

//SYMTAB - Linked List

typedef struct SYMTAB{

char symbol[10];

int loc;

struct SYMTAB\* next;

}SYMTAB;

SYMTAB\* symhead;

typedef struct ObjectCode{

char objectcode[30];

int locctr;

int mod\_flag;

struct ObjectCode\* next;

}ObjectCode;

ObjectCode\* obhead;

//program length

int prog\_len;

//program starting address;

int start\_addr;

//base address

int base\_addr;

//insert LABEL, LOCCTR into SYMTAB

void insertSymbol(char\*, int);

//search for symbol

int SearchSymbol(char\*);

//main assembler

int assemble(char\*);

//for Pass 1 of Assembler

int pass\_1(char\*);

//for Pass 2 of Assembler

int pass\_2(char\*);

//for object code

char\* findObj(char\*,char\*,int);

//for inserting node at ObjectCode Linked List

void insertObj(char\*, int, int);

//for Freeing Object Code Linked List

void FreeObj();

//\*\*\*when symbol\*\*\*

//----------------------------------------

//SYMTAB print

void ShowSymbol();

//Free SYMTAB

void FreeSymbol();

* 1. makefile

CC = gcc

CFLAGS = -Wall -W -g

OBJECTS = 20171662.o shellcmd.o memcmd.o opcmd.o sicxecmd.o

TARGET = 20171662.out

all: $(TARGET)

$(TARGET) : $(OBJECTS)

$(CC) $(CFLAGS) -o $(TARGET) $(OBJECTS)

clean:

rm -rf $(OBJECTS) $(TARGET) $(TARGET).dSYM