과목 명: 시스템프로그래밍

담당 교수 명: 소 정 민

<<Assignment 2>>

**서강대학교 컴퓨터공학과**

**[학번] 20171665**

**[이름] 이선호**

목 차

1. 프로그램 개요 3

2. 프로그램 설명 3

2.1 프로그램 흐름도 3

3. 모듈 정의 5

3.1 모듈 이름: main 5

3.1.1 기능

3.1.2 사용 변수

3.2 모듈 이름: InitializeSymbolTable 6

3.2.1 기능

3.2.2 사용 변수

3.3 모듈이름: InsertSymbolTable 6

3.3.1 기능

3.3.2 사용변수

3.4 모듈이름: DeleteSymbolTable 6

3.4.1 기능

3.4.2 사용변수

3.5 모듈이름: SearchSymbolTable 6

3.5.1 기능

3.5.2 사용변수

3.6 모듈 이름: PrintSymbolTable 7

3.1.1 기능

3.1.2 사용 변수

3.7 모듈 이름: TypeCommand 7

3.2.1 기능

3.2.2 사용 변수

3.8 모듈이름: AssemblePassOne 7

3.3.1 기능

3.3.2 사용변수

3.9 모듈이름: AssemblePassTwo 8

3.5.1 기능

3.5.2 사용변수

4. 전역 변수 정의 9

4.1 typedef struct Symbol SYMBOL 9

4.2 int StartAddress 9

4.3 int ProgramLength 9

4.4 SYMBOL SymbolTable[20] 9

5. 코드 10

5.1 20171665.h 10

5.2 20171665.c 12

5.3 AssemblePassOne.c 33

5.4 AssemblePassTwo.c 41

# 프로그램 개요

1주차에서 생성한 SIC(Simplified Instruction Computer) machine의 기본적인 기능에서 나아가 assembly program source file을 input으로 읽어서 label에 관한 table을 생성하는 기능이 추가된다. 또한 앞서 제작한 opcode table를 사용하여 전반적으로 input file을 두번 읽어서 list와 object code file로 변환하여 작성하는 작업을수행한다.

# 프로그램 설명

## 프로그램 흐름도

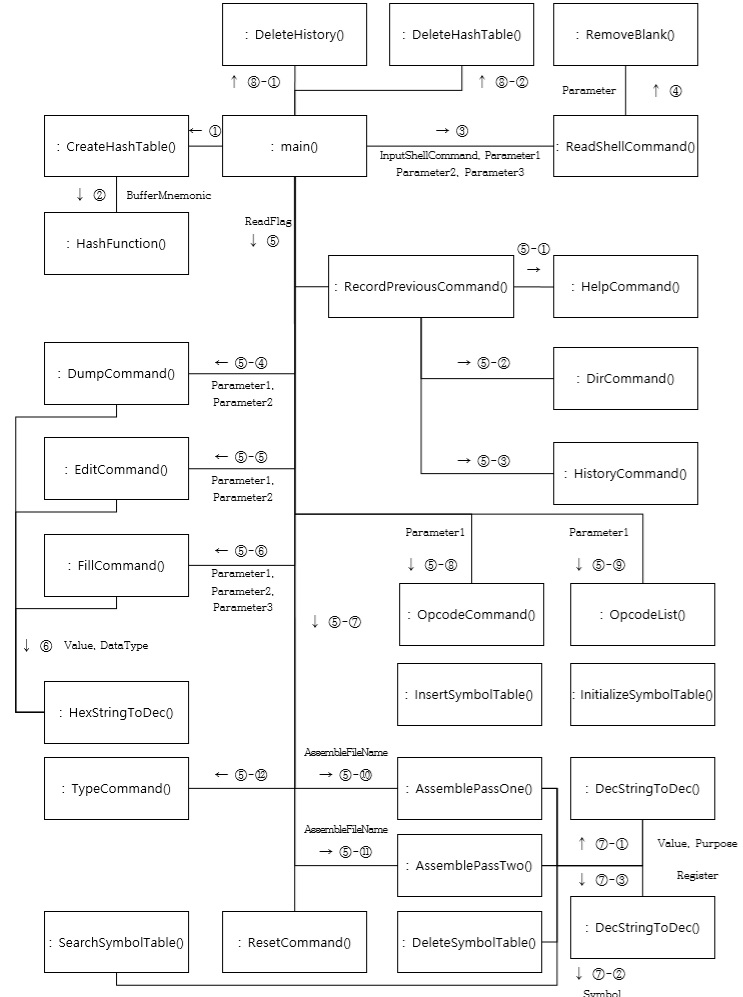


그림 1> 프로그램 흐름도

이 프로그램에서 1주차에서 제작했던 각 sub-routine들의 상호작용에다가 2주차에서 요구하는 기능을 수행하기 위해 Symbol에 관한 함수인 InitializeSymbolTable, InsertSymbolTable, DeleteSymbolTable, SearchSymbolTable, PrintSymbolTable, FindRegisterNumber와, 어셈블리어로 작성된 코드를 machine code로 변환하는 AssemblePassOne, AssemblePassTwo, 그리고 TypeCommand 함수를 추가하여 프로그램 흐름도를 그렸다.

1주차와 마찬가지로 프로그램 처음 실행할 때 main 함수에서 CreateHashTable 함수를 호출한다. 이 함수에서는 opcode.txt 파일을 읽어서 opcode mnemonic을 인수로 하여 HashFunction 함수에서 hash value를 구하여 hash table을 완성한다. 이후 사용자로부터 quit command를 입력받기 전까지 command를 매번 입력받아 그에 맞는 sub-routine을 실행하기 위해 while loop을 돈다. 사용자로부터 commad line을 입력받아서 command와 parameter들의 공백을 제거하여 분리하는 작업을 ReadShellCommand와 RemoveBlank 함수에서 실행한다. 이를 통해 얻은 command 에 일대일 대응되는 값인 ReadFlag에 따라switch문을 통해 어떠한 sub-routine을 실행할지 결정한다. ReadFlag가 HelpCommand, DirCommand, HistoryCommand 함수에 대응될 경우 우선 RecordPreviousCommand 함수를 실행하여 이전 command 내역을 저장하는 linked list에 저장하고 나서 실행한다. 이 외의DumpCommand, EditCommand, FillCommand, ResetCommand, OpcodeCommand, OpcodeList, TypeCommand 함수는 필요한 개수의 Parameter들을 인수로 전달받아서 실행한다.

어셈블리어로 짠 코드를 Pass 1과 Pass 2로 두 번 읽기 위해서 각각 AssemblePassOne과 AssemblePassTwo라는 독립적인 모듈을 제작했다. Command 중에서 ‘assemble’와 Parameter1을 통해 file name을 입력받으면 AssemblePassOne에 의해 해당 name의 파일을 fopen으로 연다. 만일, 열기 과정에서 오류가 있을 경우 error를 출력하고 해당 함수를 종료한다. 파일을 용이하게 읽어서 lst 또는 obj 파일로 변환하기 위해 Pass 1에서 각 line마다 instruction의 주소에 관한 정보를 추가한immediate file을 생성하여 이 파일을 가지고 Pass 2에서 다시 읽도록 제작했다. 그래서 ‘(filename).imm’라는 파일을 Pass 1에서 새로 생성한다.

이후 assembly language로 작성된 파일로부터 EOF 문자를 만날 때까지 문장들을 차례대로 읽는다. 한 문장을 읽을 때마다 label과 opcode mnemonic, 그리고 operand를 분리하여 각각 string으로 저장한다. 첫 번째 opcode mnemonic이 ‘START’ directive가 아니라면 해당 줄을 오류로 출력하고, 그렇지 않으면 프로그램의 시작 주소를 저장한다. 만약 operand에서 프로그래머가 지정한 시작 주소가 존재하면 이를 시작 주소로 설정하고, 그렇지 않으면 0으로 정한다. Label이 존재하면 해당 label이 Symbol Table에 존재하는지 SearchSymbolTable을 통해 확인하고, 만일 존재하지 않으면 새로 Symbol을 InsertSymbolTable 함수를 통해 추가한다. 읽은 opcode mnemonic을 opcode table에서 탐색했는데 존재한다면 해당 opcode에서 가능한 형식을 파악하여 해당 길이만큼 현재 instruction의 주소에서 더하여 program counter를 설정한다. 그러나 가능한 형식이 없다면 오류를 출력한다. 만약 opcode mnemonic을 opcode table에서 찾지 못하면 END, BASE, NOBASE, RESW, RESB, WORD, BYTE 등 가능한 directive와 같은지 대조한다. END directive는 프로그램 끝에 반드시 있어야 하고 한 번만 나와야 하기 때문에 이를 EndFlag를 사용하여 조건을 만족하는지 확인한다. BYTE는 30 byte 이하의 원하는 hexadecimal digit 또는 character를 작성할 수 있도록 지원해야 하므로 작성한 문자열 길이만큼 instruction 주소에 더하여 program counter 값을 반영한다. 모든 문장에 관해서 instruction 주소와 함께 문장들을 imm 파일에 기록하고 오류가 존재하지 않으면 Pass 2로 넘어간다.

AssemblePassTwo에서는 앞에서 작성한 immediate file을 읽어서 각 instruction 문장들에 대응되는 hexadeicmal code를 제작해서 lst와 obj 파일을 작성한다. 각 opcode mnemonic마다 opcode table를 탐색하여 존재하는지 확인하는데, 존재하면 해당 opcode mnemonic에 대응되는 opcode number를 opcode format에 작성하고, 그렇지 않으면 가능한 directive에 속하는지 확인한다. Opcode format은 총 4가지가 존재하는데, format 1은 1 byte, format 2는 2 byte, format 3는 3 byte, format 4는 4 byte를 저장한다. Format 1과 2는 opcode 영역이 1byte에 저장되도록 하고, 나머지 3와 4는 6 bit에 저장되도록 한다. 이를 위해 opcode 영역에 해당하는 hexadecimal digit을 저장할 별도의 배열인 HexCodeOpcodeAndNI를 사용한다. Format 3와 4에서는 operand앞에 어떠한 문자가 어 있는지를 확인하여 immediate addressing 또는 indirect addressing을 사용할지 결정한다. Operand 뒤에 ‘,X’ 문자가 존재하면 index addressing을 사용할 것으로 결정하고, 해당 addressing 방법에 맞춰서 HexCodeXIBP 변수에 값으로 저장한다. Format 3에서는 relative addressing을 수행하기 위해immediate file에서 instruction 줄마다 저장한 주소를 가지고 program counter를 계산하여 PC relative를 수행해야 하는지 Base relative를 수행해야 하는지 결정한다. 만일 둘 다 가능한 범위를 초과하여 displacement 영역에 저장하지 못할 경우 오류를 출력한다. Format 4에서는 opcode mnemonic 앞에 ‘+’ 문자가 존재하는지 확인하고, direct addressing만 수행 가능하다는 점을 고려하여 format 3에서와 마찬가지로 HexCodeXIBP에 유효한 값을 저장한다. 단, direct addressing을 수행하는 문장이면 프로그램의 재배치를 고려하여 ModifiedRecord 배열에 이를 저장한다. Operand가 Label인지 SearchSymbolTable 함수를 통해 확인하는데 존재하는 label이면 해당 label에 대응되는 주소 값을 사용하고, 그렇지 않으면 오류를 출력한다. 한 줄에 대하여Addressing에 관한 모든 처리를 완료하면 HexCode 배열에 HexCodeOpcodeAndNI와 HexCodeXIBP, 그리고 operand를 16진수로 변환한 string을 합쳐서 저장하여 이를 lst파일과 obj 파일에 작성한다. 모든 instruction을 다 읽고 hex code를 object 파일에 작성하면 마찬가지로 프로그램 재배치를 고려하여 앞에서 저장한Modification Record를 작성하고 End Record와 함께 프로그램의 시작 주소를 작성하여 완료한다. 만약 도중에 오류가 발생하는 case가 생기면 ErrorFlag를 1로 설정하고 이제까지 작성한 lst와 obj, 그리고 imm 파일을 삭제한다.

# 모듈 정의

## 모듈 이름 : InitializeSymbolTable()

### 기능

SymbolTable을 사용하기 전에 기존에 저장되어 있던 symbol table 내용을 0으로 초기화한다.

### 사용 변수

Index – 20개의 기본적인 원소를 가지고 있는 SymbolTable의 각 배열의 첫 번째 원소가 비어 있는지를 loop을 돌며 확인하기 위해 사용한다.

LabelNamePointer – Symbol Table의 기본적인 20개 원소에 저장된 character 배열인 label name의 첫 번째 원소의 주소이다.

## 모듈 이름: InsertSymbolTable(char \*Label, int LocationCounter)

### 기능

HashFunction을 사용하여 Symbol에 대응되는 Hash 값을 구하여 해당 번째 linked list 뒤에 삽입하는 기능을 수행한다.

### 사용 변수

Buffer – HashFunction을 통해 구한 해당 Label에 대응되는 Symbol의 Hash값을 갖는다.

NewSymbol – Linked List의 맨 뒤쪽에 삽입하고자 하는 Symbol Node이다.

## 모듈이름: DeleteSymbolTable()

### 기능

Symbol Table을 사용하기 전에 기존에 저장되어 있던 symbol table의 내용을 지운다.

### 사용변수

Index – 20개의 기본적인 원소를 가지고 있는 SymbolTable의 각 배열의 첫 번째 원소를 loop를 돌며 접근하기 위해 사용한다.

TempSymbol – Linked List를 따라 가면서 node를 삭제하기 위해 사용한다.

DeleteSymbol – Linked List를 따라 가면서 삭제할 node를 의미한다.

## 모듈이름: SearchSymbolTable(char \*Label, SYMBOL \*\*TempPointer)

### 기능

찾고자 하는 Label이 Symbol Table에 존재하는지 HashFunction 함수를 사용하여 hash 값을 구한 후 Symbol 배열의 해당 원소로 접근하여 Linked list를 탐색하는 용도로 쓰인다..

### 사용변수

HashCode - HashFunction으로 찾은 label의 Hash값을 의미한다.

TempSymbol – Symbol 배열의 linked list를 따라가면서 찾고자 하는 label을 탐색하기 위해 사용한다.

## 모듈이름: PrintSymbolTable()

### 기능

Symbol Table에 저장된 symbol들을 역방향 alphabetical 순으로 창에 출력하는 역할을 한다.

### 사용변수

Index - 20개의 기본적인 원소를 가지고 있는 SymbolTable의 각 배열의 첫 번째 원소를 loop를 돌며 접근하기 위해 사용한다.

SymbolNumber – Symbol Table에 저장된 symbol의 개수가 총 몇 개인지 계산한다.

TempLabelAddress – Symbol이 갖고 있는 instruction의 주소 값을 저장하기 위해 사용한다.

TempSymbol – Symbol Table의 linked list에 접근하여 symbol의 개수가 몇 개인지 계산할 때 사용한다.

SortedSymbolList – Symbol을 역방향 alphabetical 순으로 정렬한 list를 저장한다.

TempLabelName – Symbol(Label)의 이름 배열의 첫 원소의 주소를 갖는다.

## 모듈이름: TypeCommand(char \*FileName)

### 기능

출력하고자 하는 파일을 열어서 안의 데이터를 character 단위로 읽으며 내용을 출력한다.

### 사용변수

FilePointer – FileName을 파일명으로 갖는 파일을 fopen으로 열기 위한 File Pointer 변수이다.

BufferChar – 파일의 데이터를 character 단위로 읽은 byte 값을 갖는다.

## 모듈이름: FindRegisterNumber(char \*Register)

### 기능

Operand와 같은 이름을 갖는 유효한 Register가 존재하는지 알려준다. -1을 반환하면 대응되는 Register가 없는 것이고, 0이면 찾은 것이다.

### 사용변수

ReturnValue – 함수의 반환값을 갖는다. -1이면 대응되는 register를 찾기를 실패한 것이고, 0이면 대응되는 register를 찾은 것이다.

## 모듈이름: AssemblePassOne(char\* AssembleFileName)

### 기능

Assembly Language로 작성된 코드를 list 또는 object 파일의 코드로 변환하여 작성하기 위해서는 두 번의 읽기가 필요한데, 그 중 첫 번째 읽기인 Pass 1을 수행하는 함수이다. 여기서는 각 instruction의 주소를 계산하고, 필요한 symbol을 Symbol Table에 저장함으로써 다음 Pass 2에서 Operand에 대응되는 label의 주소를 찾을 수 있게끔 한다. 계산한 instruction들의 주소는 immediate file에 같이 작성되고, 이 immediate file이 Pass 2에서 사용된다.

### 사용변수

FilePointer – 변환하고자 하는 파일의 File Pointer 변수이다.

ImmediatePointer – Immediate 파일의 File Pointer 변수이다.

FileName –변환하고자 하는 파일의 이름이다.

ImmediateFileName – Immediate 파일의 이름이다.

ListFileName – lst 확장자를 갖는 파일의 이름이다.

ObjectFileName – obj 확장자를 갖는 파일의 이름이다.

FileNameLength – 파일명의 길이를 계산하여 저장한다.

ErrorFlag – Pass 1 중 error가 발생했을 때 0에서 1로 설정된다. 이는 Pass 2를 수행하지 않고 기존에 수행했던 모든 결과 데이터를 초기화하기 위해 필요하다.

LabelCounter – Label을 문자열로 저장할 때 해당 문자열의 길이를 구하기 위해 사용한다.

OpcodeMnemonicCounter – OpcodeMnemonic을 문자열로 저장할 때 해당 문자열의 길이를 구하기 위해 사용한다.

OperandCounter – Operand를 문자열로 저장할 때 해당 문자열의 길이를 구하기 위해 사용한다.

AddLocationCounter – Program Counter가 현재 instruction 주소에서 몇 바이트 더한 값이어야 하는지를 저장한다.

StartingAddress – 프로그램의 시작 주소를 저장한다.

LocationCounter – 현재 instruction의 주소를 갖는다.

LineNumber – Assembly Language 코드의 줄이 몇 번째인지 그 값을 갖는다.

Flag – 한 줄을 읽을 때 Label, Opcode, Operand를 구분지어 문자열에 각각 저장하기 위해 사용한다. 이는 줄을 읽을 때 문자 단위로 읽기 때문에 어디서 저장되는 문자열이 바뀌는지 알기 위해서다.

NotAddressFlag – Immediate 파일에 주소를 저장해야 하는지 그 여부를 알기 위해 사용한다.

StartFlag – ‘START’ directive가 파일의 첫 번째 줄에 존재하는지 확인하기 위해 사용한다.

EndFlag – ‘END’ directive가 파일에 한 개만 존재하는지 확인하기 위해 사용한다.

NowStartFlag - ‘START’ directive가 있는 줄에서는 label을 symbol로 저장하지 않기 위해 사용한다.

CharacterFlag – BYTE에 의해 Character가 입력될 때 공백도 같이 받고자 사용한다.

CommentFlag – 현재 읽은 줄이 주석에 해당되는지 확인하기 위해 사용한다.

BufferChar – 파일에서 데이터를 읽을 때 character 단위로 읽은 byte 값을 저장한다.

BeforeChar – 이전에 바로 읽은 byte 값을 저장한다.

Label – Label에 해당하는 문자열을 의미한다.

OpcodeMnemonic – Opcode의 Mnemonic에 해당하는 문자열을 의미한다.

Operand – Operand에 해당하는 문자열을 의미한다.

## 모듈이름: AssemblePassTwo(char\* AssembleFileName)

### 기능

사용자로부터 ‘quit’ command를 입력받아서 프로그램을 종료할 때, 이전에 저장된 command들을 모두 free하여 메모리를 해제한다.

### 사용변수

FilePointer – 변환하고자 하는 파일의 File Pointer 변수이다.

ImmediatePointer – Immediate 파일의 File Pointer 변수이다.

FileName –변환하고자 하는 파일의 이름이다.

ImmediateFileName – Immediate 파일의 이름이다.

ListFileName – lst 확장자를 갖는 파일의 이름이다.

ObjectFileName – obj 확장자를 갖는 파일의 이름이다.

FileNameLength – 파일명의 길이를 계산하여 저장한다.

ErrorFlag – Pass 2 중 error가 발생했을 때 0에서 1로 설정된다. 이는 lst 또는 obj 파일을 작성하지 않고 기존에 수행했던 모든 결과 데이터를 초기화하기 위해 필요하다.

LabelCounter – Label을 문자열로 저장할 때 해당 문자열의 길이를 구하기 위해 사용한다.

OpcodeMnemonicCounter – OpcodeMnemonic을 문자열로 저장할 때 해당 문자열의 길이를 구하기 위해 사용한다.

OperandCounter – Operand를 문자열로 저장할 때 해당 문자열의 길이를 구하기 위해 사용한다.

AddLocationCounter – Program Counter가 현재 instruction 주소에서 몇 바이트 더한 값이어야 하는지를 저장한다.

StartingAddress – 프로그램의 시작 주소를 저장한다.

LocationCounter – 현재 instruction의 주소를 갖는다.

LineNumber – Assembly Language 코드의 줄이 몇 번째인지 그 값을 갖는다.

Flag – 한 줄을 읽을 때 Label, Opcode, Operand를 구분지어 문자열에 각각 저장하기 위해 사용한다. 이는 줄을 읽을 때 문자 단위로 읽기 때문에 어디서 저장되는 문자열이 바뀌는지 알기 위해서다.

NotAddressFlag – Immediate 파일에 주소를 저장해야 하는지 그 여부를 알기 위해 사용한다.

StartFlag – ‘START’ directive가 파일의 첫 번째 줄에 존재하는지 확인하기 위해 사용한다.

CharacterFlag – BYTE에 의해 Character가 입력될 때 공백도 같이 받고자 사용한다.

CommentFlag – 현재 읽은 줄이 주석에 해당되는지 확인하기 위해 사용한다.

BufferChar – 파일에서 데이터를 읽을 때 character 단위로 읽은 byte 값을 저장한다.

BeforeChar – 이전에 바로 읽은 byte 값을 저장한다.

Label – Label에 해당하는 문자열을 의미한다.

OpcodeMnemonic – Opcode의 Mnemonic에 해당하는 문자열을 의미한다.

Operand – Operand에 해당하는 문자열을 의미한다.

HexAddressCounter – 현재 instruction의 주소 문자열의 길이를 갖는다.

BaseRegister – Base relative addressing을 사용하기 위해 base register에 값을 저장하고자 사용한다.

ProgramCounter – Program counter relative addressing을 사용하기 위해 program counter에 값을 저장하고자 사용한다.

RelocationCounter – 프로그램 재배치 시 몇 개의 instruction의 operand 주소를 수정해야 할지 그 개수를 저장한다.

RealOperand – Operand를 실제 10진수로 바꿨을 때 그 값을 갖는다.

NIFlag – Indirect 또는 Immediate addressing을 사용해야 하는지 여부를 저장한다.

BaseFlag – Base relative addressing을 사용해야 하는지 그 여부를 저장한다.

IndexFlag – Index addressing을 사용해야 하는지 그 여부를 저장한다.

MemoryFlag – 메모리 할당 관련 BYTE, WORD directive에 의해 메모리를 저장해야 하는지 그 여부를 저장한다.

ReserveFlag – RESW, RESB directive에 의해 operand에 해당하는 크기만큼 메모리를 할당해야 하는지 그 여부를 저장한다.

HexAddress – Immediate 파일로부터 읽은 현재 instruction의 주소 문자열을 저장한다.

TempOperand – Operand에서 필요한 데이터 문자열을 일부 복사하여 저장하기 위해 사용한다.

HexCode – Insruction을 hexadecimal object code로 변환했을 때의 문자열을 저장한다.

HexCodeOpcodeAndNI – Opcode format에서 Opcode과 indirect, immediate addressing flag 값을 문자열로 저장한다.

HexCodeXBPE – Index , Base relative, PC relative, Extension flag 값을 문자열로 저장한다.

HexCodeOperand – Hexadeicaml로 변환한 Operand 값을 문자열로 저장한다.

FirstRegister – Format 2에서 필요로 하는 두 개의 reigster에서 첫 번째 register를 의미한다.

SecondRegister – Format 2에서 필요로 하는 두 개의 register에서 두 번째 register를 의미한다.

MemoryContent – 메모리 관련 directive에 의해 저장된 메모리 문자열을 갖는다.

ProgramName – 프로그램 이름 문자열을 갖는다.

TextRecordStartAddress – Text Record를 작성할 때 각 줄마다 시작되는 instruction 주소를 저장하기 위해 사용한다.

TextRecordObjectCode – Text Record에서 한 줄당 입력되는 최대 30 byte의 길이를 갖는 코드이다.

ModifiedAddress – 프로그램 재배치 시 operand의 주소를 수정해야 하는 instruction의 주소를 배열로 저장한다.

TextRecordCounter – END directive를 읽었을 때 이제까지 Text Record의 한 줄에 저장해야 하는 코드가 있는지 확인하기 위해 사용한다.

# 전역 변수 정의

## typedef struct Symbol{ char \*LabelName, int LabelAddress, short LabelFlag, struct Symbol \*Next} SYMBOL

Assembly Language Code에서 읽은 데이터에 존재하는 label을 갖는 symbol table을 구성할 때 table 각각의 node의 데이터를 저장하기 위해 사용한다. SYMBOL은 symbol의 이름에 해당하는 LabelName, symbol이 존재하는 주소인 LabelAddress, 이미 존재하는 symbol인지 확인하는 LabelFlag, 그리고 다음 node를 linking하기 위한 자기 자신을 data type으로 갖는 포인터 변수를 멤버 변수로 갖는다.

## int StartAddress

처음 프로그램이 시작되는(‘START’ directive가 존재하는) instruction의 주소를 저장한다.

## int ProgramLength

프로그램의 길이가 얼마인지 그 값을 저장한다.

## SYMBOL SymbolTable[20]

Symbol들을 Pass 1에서 주소와 이름을 저장하는 크기가 20인 Symbol Table이다.

# 코드

## 20171665.h

#ifndef \_\_HEADER\_\_

#define \_\_HEADER\_\_

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <dirent.h>

//#include <unistd.h> // 리눅스에서 io.h 대신 사용

#include <io.h> // 비주얼스튜디오에서 unistd.h 대신 사용

#include <errno.h>

#define Help 1

#define Dir 2

#define Quit 3

#define History 4

#define Dump 5

#define Edit 6

#define Fill 7

#define Reset 8

#define Opcode 9

#define OpcodeList 10

#define Assemble 11

#define SymbolCommand 12

#define Type 13

#define Word 100

#define Character 101

#define Integer 102

#define Hexadecimal 103

#define ForHexAddress 104

#define ForProgramCounter 105

#define IndirectAddressing 200

#define ImmediateAddressing 201

typedef struct Node {

char \*Value;

struct Node\* Next;

}NODE;

typedef struct Hash {

char Number[5];

char Mnemonic[10];

char Format[10];

struct Hash\* Next;

}HASH;

typedef struct Symbol {

char \*LabelName;

int LabelAddress;

short LabelFlag;

struct Symbol\* Next;

}SYMBOL;

extern NODE \*FirstNode;

extern HASH HashTable[20]; // Hash Table 선언

extern SYMBOL SymbolTable[20]; // Symbol Table 선언

extern char DumpMemory[0x100000];

extern int LastAddress;

extern int StartAddress;

extern int ProgramLength;

extern void RemoveBlank(char \*\*Parameter);

extern int HexStringToDec(char \*Parameter, short DataType);

extern int DecStringToDec(char \*Parameter, short Purpose);

extern int ReadShellCommand(char \*\*InputShellCommand, char \*\*Parameter1, char \*\*Parameter2, char\*\* Parameter3);

extern void HelpCommand();

extern void DirCommand();

extern void RecordPreviousCommand(char\* InputShellCommand);

extern void HistoryCommand();

extern void DeleteHistory();

extern int DumpCommand(char \*Start, char \*End);

extern int EditCommand(char \*Address, char \*Value);

extern int FillCommand(char \*Start, char \*End, char \*Value);

extern int ResetCommand();

extern int HashFunction(char \*Value);

extern int OpcodeCommand(char \*Mnemonic);

extern int SearchOpcode(char \*Mnemonic, HASH\*\* TempPointer);

extern int OpcodeListCommand();

extern void DeleteHashTable();

extern int CreateHashTable();

extern int InitializeSymbolTable();

extern int InsertSymbolTable(char \*Label, int LocationCounter);

extern void DeleteSymbolTable();

extern int SearchSymbolTable(char \*Label, SYMBOL \*\*TempPointer);

extern int PrintSymbolTable();

extern int FindRegisterNumber(char \*Register);

extern int AssemblePassOne(char \*AssembleFileName);

extern int AssemblePassTwo(char \*AssembleFileName);

#endif

## 20171665.c

#include "20171665.h"

//#include <crtdbg.h> // 메모리 누수 확인

NODE \*FirstNode = NULL;

HASH HashTable[20]; // Hash Table 선언

SYMBOL SymbolTable[20]; // Symbol Table 선언

char DumpMemory[0x100000] = { 0 };

int LastAddress = -1;

int StartAddress = -1;

int ProgramLength = 0;

/\*------------------------------------------------------------------------------------\*/

/\*함수 : RemoveBlank(char \*\*Parameter)\*/

/\*목적 : 전달받은 Parameter에서 공백을 제거한다.

/\*리턴값 : 없음\*/

/\*------------------------------------------------------------------------------------\*/

void RemoveBlank(char \*\*Parameter) {

int Index = 0, TempIndex = 0;

while ((\*Parameter)[TempIndex] == ' ' || (\*Parameter)[TempIndex] == '\t') {

TempIndex++;

}

while ((\*Parameter)[TempIndex] != '\0') {

(\*Parameter)[Index++] = (\*Parameter)[TempIndex++];

}

if ((!Index) && (\*Parameter != NULL)) {

free(\*Parameter);

\*Parameter = NULL;

}

else {

(\*Parameter)[Index] = '\0';

}

return;

}

/\*------------------------------------------------------------------------------------\*/

/\*함수 : HexStringToDec(char \*Parameter, short DataType)\*/

/\*목적 : 전달받은 Parameter에서 16진수로 된 string을 10진수의 integer 값으로 바꾼다.

/\*리턴값 : 16진수를 10진수의 integer로 변환한 값을 반환한다. 입력한 값이 16진수가 아니거나 변환하고자 하는 data type의 범위를 초과할 경우 -1을 반환한다.\*/

/\*------------------------------------------------------------------------------------\*/

int HexStringToDec(char \*Parameter, short DataType) {

long Buffer = -1;

char \*EndPointer = NULL;

errno = 0;

Buffer = strtol(Parameter, &EndPointer, 16);

if (DataType != ForHexAddress && DataType != ForProgramCounter) {

if (EndPointer == Parameter || \*EndPointer != '\0') {

printf("No digits were found.\n");

return -1;

}

if ((Buffer == 2147483647L || Buffer == -2147483647L - 1) && errno == ERANGE) {

printf("The value you insert is invalid. Please write the valid value.\n");

return -1;

}

if (!(Buffer >= 0x00 && Buffer <= 0xFF) && DataType == Character) {

printf("The value you insert is out of boundary. (Boundary: 0x00 ~ 0xFF)\n");

return -1;

}

}

return (int)Buffer;

}

/\*------------------------------------------------------------------------------------\*/

/\*함수 : \*/

/\*목적 :

/\*리턴값 : \*/

/\*------------------------------------------------------------------------------------\*/

int DecStringToDec(char \*Parameter, short Purpose) {

long Buffer = -1;

char \*EndPointer = NULL;

errno = 0;

Buffer = strtol(Parameter, &EndPointer, 10);

if (Purpose != ImmediateAddressing && Purpose != Word) {

if (EndPointer == Parameter || \*EndPointer != '\0') {

return -1;

}

if ((Buffer == 2147483647L || Buffer == -2147483647L - 1) && errno == ERANGE) {

return -1;

}

}

return (int)Buffer;

}

/\*------------------------------------------------------------------------------------\*/

/\*함수 : ReadShellCommand(char \*\*InputShellCommand, char \*\*Parameter1, char \*\*Parameter2, char\*\* Parameter3)\*/

/\*목적 : 전달받은 Command와 Parameter에 따라 유효한 명령을 수행할 수 있도록 한다.

/\*리턴값 : 유효하지 않을 경우 0을 반환하고, 유효할 경우 입력받은 명령에 따라서 Header 파일에 정의된 명령값들을 반환한다.\*/

/\*------------------------------------------------------------------------------------\*/

int ReadShellCommand(char \*\*InputShellCommand, char \*\*Parameter1, char \*\*Parameter2, char\*\* Parameter3) {

int BufferCounter = 0, ResultValue = 0;

short Flag = 0, Invalid = 0;

int CommandCounter = 0, ParameterCounter1 = 0, ParameterCounter2 = 0, ParameterCounter3 = 0;

char BufferChar = EOF, BeforeChar = EOF, \*BufferCommand = NULL;

printf("sicsim> ");

BufferCommand = (char\*)realloc(BufferCommand, sizeof(char) \* (CommandCounter + 2));

\*Parameter1 = (char\*)realloc(\*Parameter1, sizeof(char) \* (ParameterCounter1 + 2));

\*Parameter2 = (char\*)realloc(\*Parameter2, sizeof(char) \* (ParameterCounter2 + 2));

\*Parameter3 = (char\*)realloc(\*Parameter3, sizeof(char) \* (ParameterCounter3 + 2));

while ((BufferChar = getchar()) != '\n' && BufferChar != EOF) {

\*InputShellCommand = realloc(\*InputShellCommand, sizeof(char) \* (BufferCounter + 2));

(\*InputShellCommand)[BufferCounter++] = BufferChar;

if (BufferChar == ' ' || BufferChar == '\t') {

// 명령만 입력받은 상태에서 공백 또는 Tab이 입력된 경우

if (CommandCounter && Flag == 0) {

Flag = 1;

}

BeforeChar = BufferChar;

continue;

}

else if (BufferChar == ',') {

// Parameter를 한 개 입력받은 상태에서 ','가 입력된 경우

if (ParameterCounter1 && Flag == 1) {

Flag = 2;

}

// Parameter를 두 개 입력받은 상태에서 ','가 입력된 경우

else if (ParameterCounter2 && Flag == 2) {

Flag = 3;

}

// Parameter를 세 개 입력받은 후 또는 Command가 입력된 후 ','가 입력된 경우

else if ((ParameterCounter3 && Flag == 3) || (!ParameterCounter1 && Flag == 1) || (Flag == 0)) {

Invalid = 1;

}

BeforeChar = BufferChar;

continue;

}

else if (BeforeChar == ' ') {

// Parameter에서 띄어쓰기 처리가 된 경우

if ((ParameterCounter1 && Flag == 1) || (ParameterCounter2 && Flag == 2) || (ParameterCounter3 && Flag == 3)) {

Invalid = 1;

}

}

switch (Flag) {

case 0:

BufferCommand = (char\*)realloc(BufferCommand, sizeof(char) \* (CommandCounter + 2));

BufferCommand[CommandCounter++] = BufferChar;

break;

case 1:

\*Parameter1 = (char\*)realloc(\*Parameter1, sizeof(char) \* (ParameterCounter1 + 2));

(\*Parameter1)[ParameterCounter1++] = BufferChar;

break;

case 2:

\*Parameter2 = (char\*)realloc(\*Parameter2, sizeof(char) \* (ParameterCounter2 + 2));

(\*Parameter2)[ParameterCounter2++] = BufferChar;

break;

case 3:

\*Parameter3 = (char\*)realloc(\*Parameter3, sizeof(char) \* (ParameterCounter3 + 2));

(\*Parameter3)[ParameterCounter3++] = BufferChar;

break;

default:

break;

}

BeforeChar = BufferChar;

}

if ((\*InputShellCommand) != NULL) {

BufferCommand[CommandCounter] = (\*Parameter1)[ParameterCounter1] = (\*Parameter2)[ParameterCounter2] = (\*Parameter3)[ParameterCounter3] = (\*InputShellCommand)[BufferCounter] = '\0';

}

RemoveBlank(Parameter1);

RemoveBlank(Parameter2);

RemoveBlank(Parameter3);

if (Invalid != 1) {

// 입력받은 Command에서 Parameter가 하나도 없는 경우

if (\*Parameter1 == NULL) {

if (!strcmp(BufferCommand, "h") || !strcmp(BufferCommand, "help")) {

ResultValue = Help;

}

else if (!strcmp(BufferCommand, "d") || !strcmp(BufferCommand, "dir")) {

ResultValue = Dir;

}

else if (!strcmp(BufferCommand, "q") || !strcmp(BufferCommand, "quit")) {

ResultValue = Quit;

}

else if (!strcmp(BufferCommand, "hi") || !strcmp(BufferCommand, "history")) {

ResultValue = History;

}

else if (!strcmp(BufferCommand, "du") || !strcmp(BufferCommand, "dump")) {

ResultValue = Dump;

}

else if (!strcmp(BufferCommand, "reset")) {

ResultValue = Reset;

}

else if (!strcmp(BufferCommand, "opcodelist")) {

ResultValue = OpcodeList;

}

else if (!strcmp(BufferCommand, "symbol")) {

ResultValue = SymbolCommand;

}

}

//Parameter가 최소 한 개 이상인 경우

else {

if ((!strcmp(BufferCommand, "du") || !strcmp(BufferCommand, "dump")) && Flag < 3) {

ResultValue = Dump;

}

else if ((!strcmp(BufferCommand, "e") || !strcmp(BufferCommand, "edit")) && Flag < 3) {

ResultValue = Edit;

}

else if (!strcmp(BufferCommand, "f") || !strcmp(BufferCommand, "fill")) {

ResultValue = Fill;

}

else if (!strcmp(BufferCommand, "opcode") && Flag < 2) {

ResultValue = Opcode;

}

else if (!strcmp(BufferCommand, "assemble") && Flag < 2) {

ResultValue = Assemble;

}

else if (!strcmp(BufferCommand, "type") && Flag < 2) {

ResultValue = Type;

}

}

}

if (BufferCommand != NULL) {

free(BufferCommand);

}

return ResultValue;

}

/\*------------------------------------------------------------------------------------\*/

/\*함수 : HelpCommand()\*/

/\*목적 : 사용자가 어떠한 Command와 그에 맞는 형식으로 Input을 입력해야 하는지를 화면에 출력한다.

/\*리턴값 : 없음\*/

/\*------------------------------------------------------------------------------------\*/

void HelpCommand() {

printf("\n");

printf("h[elp]\n");

printf("d[ir]\n");

printf("q[uit]\n");

printf("hi[story]\n");

printf("du[mp][start, end]\n");

printf("e[dit] address, value\n");

printf("f[ill] start, end, value\n");

printf("reset\n");

printf("opcode mnemonic\n");

printf("opcodelist\n");

printf("assemble filename\n");

printf("type filename\n");

printf("symbol\n");

}

/\*------------------------------------------------------------------------------------\*/

/\*함수 : DirCommand()\*/

/\*목적 : 현재 실행 파일이 위치하고 있는 디렉토리 내의 모든 파일과 폴더를 출력한다.

/\*리턴값 : 없음\*/

/\*------------------------------------------------------------------------------------\*/

void DirCommand() {

DIR \*DirectoryInfo = opendir(".");

struct dirent \*DirectoryEntry;

struct stat BufferEntry;

if (DirectoryInfo != NULL) {

int Counter = 1;

while ((DirectoryEntry = readdir(DirectoryInfo)) != NULL) {

if (Counter % 3 == 1) {

printf("\n");

}

else {

printf("\t");

}

if (stat(DirectoryEntry->d\_name, &BufferEntry) != -1) {

if (S\_ISDIR(BufferEntry.st\_mode)) {

printf("%s/", DirectoryEntry->d\_name);

}

if (S\_ISREG(BufferEntry.st\_mode)) {

printf("%s", DirectoryEntry->d\_name);

if (BufferEntry.st\_mode & S\_IXUSR) {

printf("\*");

}

}

}

Counter++;

}

printf("\n");

closedir(DirectoryInfo);

}

else {

printf("An error occurs while reading directory.\n");

}

}

/\*------------------------------------------------------------------------------------\*/

/\*함수 : RecordPreviousCommand()\*/

/\*목적 : 사용자가 입력한 Valid Command들을 Linked List에 기록한다.

/\*리턴값 : 없음\*/

/\*------------------------------------------------------------------------------------\*/

void RecordPreviousCommand(char\* InputShellCommand) {

NODE \*TempNode = NULL, \*PreviousNode = NULL;

TempNode = (NODE\*)malloc(sizeof(NODE));

TempNode->Value = (char\*)malloc(sizeof(char) \* strlen(InputShellCommand) + 1);

TempNode->Next = NULL;

strcpy(TempNode->Value, InputShellCommand);

if (FirstNode == NULL) {

FirstNode = TempNode;

}

else {

PreviousNode = FirstNode;

while (PreviousNode->Next != NULL) {

PreviousNode = PreviousNode->Next;

}

PreviousNode->Next = TempNode;

}

}

/\*------------------------------------------------------------------------------------\*/

/\*함수 : HistoryCommand()\*/

/\*목적 : RecordPreviousCommand에 의해 저장된 사용자의 이전 Command들을 모두 출력한다.

/\*리턴값 : 없음\*/

/\*------------------------------------------------------------------------------------\*/

void HistoryCommand() {

NODE \*TempNode = FirstNode;

int NodeIndex = 0;

if (TempNode == NULL) {

printf("No Command Recorded!\n");

}

else {

while (TempNode != NULL) {

printf("%-5d %s\n", (++NodeIndex), TempNode->Value);

TempNode = TempNode->Next;

}

}

}

/\*------------------------------------------------------------------------------------\*/

/\*함수 : DeleteHistory()\*/

/\*목적 : Quit Command를 통해 프로그램을 종료할 때, 저장된 Command들을 모두 free하여 메모리 해제한다.

/\*리턴값 : 없음\*/

/\*------------------------------------------------------------------------------------\*/

void DeleteHistory() {

NODE \*TempNode = FirstNode, \*DeleteNode = NULL;

if (TempNode != NULL) {

while (TempNode != NULL) {

DeleteNode = TempNode;

TempNode = TempNode->Next;

free(DeleteNode->Value);

free(DeleteNode);

}

}

return;

}

/\*------------------------------------------------------------------------------------\*/

/\*함수 : DumpCommand(char \*Start, char \*End)\*/

/\*목적 : 사전에 설정된 메모리(1MB)에서 사용자가 입력한 Parameter에 대응되는 주소 범위의 1 Byte 메모리 value를 출력한다.

/\*리턴값 : 정상적인 Parameter가 입력이 들어오고 정확히 수행되면 0을 반환하고, 그렇지 않으면 -1을 반환한다.\*/

/\*------------------------------------------------------------------------------------\*/

int DumpCommand(char \*Start, char \*End) {

int i = 0, j;

int Counter, BufferStartParameter, BufferEndParameter;

char Buffer;

if (Start != NULL) {

if ((BufferStartParameter = HexStringToDec(Start, Integer)) == -1) {

printf("Please check the [start] address.\n");

return -1;

}

if (!(BufferStartParameter >= 0 && BufferStartParameter <= 0xFFFFF)) {

printf("[Start] address is out of the boundary.\n");

return -1;

}

LastAddress = BufferStartParameter - 1;

}

LastAddress = (LastAddress + 1 <= 0xFFFFF) ? LastAddress : -1;

if (End != NULL) {

if ((BufferEndParameter = HexStringToDec(End, Integer)) == -1) {

printf("Please check the [end] address.\n");

return -1;

}

if (BufferEndParameter < BufferStartParameter) {

printf("[End] address must be larger than [Start] Address.\n");

return -1;

}

if (!(BufferEndParameter >= 0 && BufferEndParameter <= 0xFFFFF)) {

printf("[End] address is out of the boundary.\n");

return -1;

}

Counter = BufferEndParameter - LastAddress;

}

else {

Counter = 160;

}

while (i < Counter) {

if ((LastAddress + i + 1) > 0xFFFFF) {

break;

}

if ((i++) == 0) {

printf("%05X ", ((LastAddress + i) / 16) \* 16);

j = 0;

for (j = ((LastAddress + i) / 16) \* 16; j < (LastAddress + i); j++) {

printf(" ");

}

}

else if ((LastAddress + i) % 16 == 0) {

printf(" ; ");

for (j = 0; j < 16; j++) {

if ((Buffer = DumpMemory[LastAddress + i + j - 16]) >= 32 && Buffer <= 126) {

printf("%c", Buffer);

}

else {

printf(".");

}

}

printf("\n%05X ", (LastAddress + i));

}

printf("%02X ", DumpMemory[LastAddress + i] & 0xFF);

}

if ((LastAddress + i + 1) % 16 == 0) {

printf(" ; ");

for (j = ((LastAddress + i) / 16) \* 16; j <= (LastAddress + i); j++) {

if ((Buffer = DumpMemory[j]) >= 32 && Buffer <= 126) {

printf("%c", Buffer);

}

else {

printf(".");

}

}

}

else {

for (j = (LastAddress + i + 1); j < ((LastAddress + i) / 16 + 1) \* 16; j++) {

printf(" ");

}

printf(" ; ");

for (j = ((LastAddress + i) / 16) \* 16; j <= (LastAddress + i); j++) {

if ((Buffer = DumpMemory[j]) >= 32 && Buffer <= 126) {

printf("%c", Buffer);

}

else {

printf(".");

}

}

for (j = (LastAddress + i + 1); j < ((LastAddress + i) / 16 + 1) \* 16; j++) {

printf(".");

}

}

printf("\n");

LastAddress = LastAddress + i;

return 0;

}

/\*------------------------------------------------------------------------------------\*/

/\*함수 : EditCommand(char \*Address, char \*Value)\*/

/\*목적 : 메모리 주소 1 Byte와 수정하고자 하는 Value 값을 입력받아서 주소에 맞는 메모리 byte의 Value를 수정한다.

/\*리턴값 : 정상적인 Parameter가 입력이 들어오고 정확히 수행되면 0을 반환하고, 그렇지 않으면 -1을 반환한다.\*/

/\*------------------------------------------------------------------------------------\*/

int EditCommand(char \*Address, char \*Value) {

int BufferAddress = 0, BufferValue = 0;

if (Address != NULL) {

if ((BufferAddress = HexStringToDec(Address, Integer)) == -1) {

printf("Please check the address.\n");

return -1;

}

if (!(BufferAddress >= 0 && BufferAddress <= 0xFFFFF)) {

printf("The address is out of the boundary. (Boundary: 0x00000 ~ 0xFFFFF)\n");

return -1;

}

if (Value == NULL) {

printf("Please write the valid value.\n");

return -1;

}

if ((BufferValue = HexStringToDec(Value, Character)) == -1) {

printf("Please check the value.\n");

return -1;

}

DumpMemory[BufferAddress] = (char)BufferValue;

}

else {

printf("Please write the valid address.\n");

return -1;

}

return 0;

}

/\*------------------------------------------------------------------------------------\*/

/\*함수 : FillCommand(char \*Start, char \*End, char \*Value)\*/

/\*목적 : 메모리 주소 범위와 수정하고자 하는 Value 값을 입력받아서 주소 범위에 속하는 메모리 byte의 Value를 수정한다.

/\*리턴값 : 정상적인 Parameter가 입력이 들어오고 정확히 수행되면 0을 반환하고, 그렇지 않으면 -1을 반환한다.\*/

/\*------------------------------------------------------------------------------------\*/

int FillCommand(char \*Start, char \*End, char \*Value) {

int BufferStartParameter = 0, BufferEndParameter = 0, Index = 0, BufferValue = 0;

if (Start != NULL) {

if ((BufferStartParameter = HexStringToDec(Start, Integer)) == -1) {

printf("Please check the [start] address.\n");

return -1;

}

if (!(BufferStartParameter >= 0 && BufferStartParameter <= 0xFFFFF)) {

printf("The [start] address is out of the boundary. (Boundary: 0x00000 ~ 0xFFFFF)\n");

return -1;

}

if (End != NULL) {

if ((BufferEndParameter = HexStringToDec(End, Integer)) == -1) {

printf("Please check the [end] address.\n");

return -1;

}

if (BufferEndParameter < BufferStartParameter) {

printf("The [end] address should not be smaller than [start] address.\n");

return -1;

}

if (!(BufferEndParameter >= 0 && BufferEndParameter <= 0xFFFFF)) {

printf("The [end] address is out of the boundary. (Boundary: 0x00000 ~ 0xFFFFF)\n");

return -1;

}

}

if (Value == NULL) {

printf("Please write the vaild value.\n");

return -1;

}

if ((BufferValue = HexStringToDec(Value, Character)) == -1) {

printf("Please check the value.\n");

return -1;

}

for (Index = BufferStartParameter; Index <= BufferEndParameter; Index++) {

DumpMemory[Index] = (char)BufferValue;

}

}

return 0;

}

/\*------------------------------------------------------------------------------------\*/

/\*함수 : ResetCommand()\*/

/\*목적 : Reset Command를 사용자로부터 입력받으면 메모리(1MB)의 모든 Value 값을 0으로 초기화한다.

/\*리턴값 : 정상적으로 수행되면 0을 반환한다.\*/

/\*------------------------------------------------------------------------------------\*/

int ResetCommand() {

memset(DumpMemory, 0, sizeof(DumpMemory));

return 0;

}

/\*------------------------------------------------------------------------------------\*/

/\*함수 : HashFunction()\*/

/\*목적 : Opcode Mnemonic을 일대일 함수를 통해 대응되는 uniqly identified한 함수값을 구한다.

/\*리턴값 : Opcode Mnemonic과 일대일 대응되는 독자적인 Hash 값을 반환한다.\*/

/\*------------------------------------------------------------------------------------\*/

int HashFunction(char \*Value) {

int Index = 0, ReturnValue = 0;

while (Value[Index] != '\0') {

ReturnValue += (int)Value[Index++] % 19;

}

ReturnValue %= 20;

return ReturnValue;

}

/\*------------------------------------------------------------------------------------\*/

/\*함수 : OpcodeCommand()\*/

/\*목적 : HashFunction 함수를 통해 구한 Hash 값을 사용하여 Hash Table을 탐색해서 Mnemonic에 대응되는 opcode를 구한다.

/\*리턴값 : Opcode Mnemonic과 일대일 대응되는 opcode를 반환한다.\*/

/\*------------------------------------------------------------------------------------\*/

int OpcodeCommand(char \*Mnemonic) {

int HashCode;

HASH\* TempHash = NULL;

if (!((HashCode = HashFunction(Mnemonic)) >= 0 && HashCode < 20)) {

printf("Error occurs while reading hash table.\n");

return -1;

}

if ((TempHash = &(HashTable[HashCode])) == NULL) {

printf("Error occurs while reading hash table.\n");

return -1;

}

while ((strcmp(TempHash->Mnemonic, Mnemonic) != 0)) {

if (TempHash->Next == NULL) {

printf("There isn't an opcode you want to find.\n");

return -1;

}

TempHash = TempHash->Next;

}

printf("opcode is %s\n", TempHash->Number);

return 0;

}

/\*------------------------------------------------------------------------------------\*/

/\*함수 : \*/

/\*목적 :

/\*리턴값 : \*/

/\*------------------------------------------------------------------------------------\*/

int SearchOpcode(char \*Mnemonic, HASH\*\* TempPointer) {

int HashCode;

HASH\* TempHash = NULL;

if (Mnemonic[0] == '+') {

Mnemonic = Mnemonic + 1;

}

if (!((HashCode = HashFunction(Mnemonic)) >= 0 && HashCode < 20)) {

return -1;

}

if ((TempHash = &(HashTable[HashCode])) == NULL) {

return -1;

}

while ((strcmp(TempHash->Mnemonic, Mnemonic) != 0)) {

// Hash Table에서 Opcode Mnemonic를 찾지 못하면

if (TempHash->Next == NULL) {

return 1;

}

TempHash = TempHash->Next;

}

(\*TempPointer) = TempHash;

return 0;

}

/\*------------------------------------------------------------------------------------\*/

/\*함수 : OpcodeListCommand()\*/

/\*목적 : HashFunction 함수에 의해 구성된 Hash Table의 형태를 출력한다.

/\*리턴값 : Hash Table 탐색이 정상적으로 이루어지고 모든 opcode가 출력되면 0을 반환한다.\*/

/\*------------------------------------------------------------------------------------\*/

int OpcodeListCommand() {

int Index;

HASH \*TempHash = NULL;

for (Index = 0; Index < 20; Index++) {

printf("%d : ", Index);

if (strcmp(HashTable[Index].Mnemonic, "")) {

TempHash = &(HashTable[Index]);

while (TempHash->Next != NULL) {

printf("[%s, %s] -> ", TempHash->Mnemonic, TempHash->Number);

TempHash = TempHash->Next;

}

printf("[%s, %s]\n", TempHash->Mnemonic, TempHash->Number);

}

else {

printf("\n");

}

}

return 0;

}

/\*------------------------------------------------------------------------------------\*/

/\*함수 : DeleteHashTable()\*/

/\*목적 : Quit Command를 통해 프로그램을 종료할 때 기존에 저장된 Hash Table를 free로 메모리 해제한다.

/\*리턴값 : 없음\*/

/\*------------------------------------------------------------------------------------\*/

void DeleteHashTable() {

int Index;

HASH \*TempHash = NULL, \*DeleteHash = NULL;

for (Index = 0; Index < 20; Index++) {

if (strcmp(HashTable[Index].Mnemonic, "")) {

TempHash = &(HashTable[Index]);

if ((TempHash = TempHash->Next) != NULL) {

while (TempHash != NULL) {

DeleteHash = TempHash;

TempHash = TempHash->Next;

free(DeleteHash);

}

}

}

}

return;

}

/\*------------------------------------------------------------------------------------\*/

/\*------------------------------------------------------------------------------------\*/

int CreateHashTable() {

FILE \*FilePointer = NULL;

if ((FilePointer = fopen("opcode.txt", "r")) == NULL) {

printf("An error occurs while opening 'opcode.txt' file.\n");

return -1;

}

int Buffer;

char BufferNumber[5] = { '\0' }, BufferMnemonic[10] = { '\0' }, BufferFormat[10] = { '\0' };

while (fscanf(FilePointer, "%s %s %s\n", BufferNumber, BufferMnemonic, BufferFormat) != EOF) {

if (!strcmp(HashTable[(Buffer = HashFunction(BufferMnemonic))].Mnemonic, "")) {

strcpy(HashTable[Buffer].Number, BufferNumber);

strcpy(HashTable[Buffer].Mnemonic, BufferMnemonic);

strcpy(HashTable[Buffer].Format, BufferFormat);

HashTable[Buffer].Next = NULL;

}

else {

HASH \*NewHash = NULL, \*TempHash = &(HashTable[Buffer]);

NewHash = (HASH\*)malloc(sizeof(HASH));

strcpy(NewHash->Number, BufferNumber);

strcpy(NewHash->Mnemonic, BufferMnemonic);

strcpy(NewHash->Format, BufferFormat);

NewHash->Next = NULL;

while (TempHash->Next != NULL) {

TempHash = TempHash->Next;

}

TempHash->Next = NewHash;

}

}

fclose(FilePointer);

return 0;

}

/\*------------------------------------------------------------------------------------\*/

/\*------------------------------------------------------------------------------------\*/

int InitializeSymbolTable() {

int Index;

char \*LabelNamePointer = NULL;

for (Index = 0; Index < 20; Index++) {

if ((LabelNamePointer = SymbolTable[Index].LabelName) != NULL) {

free(LabelNamePointer);

}

LabelNamePointer = NULL;

SymbolTable[Index].LabelAddress = 0;

SymbolTable[Index].Next = NULL;

}

return 0;

}

/\*------------------------------------------------------------------------------------\*/

/\*------------------------------------------------------------------------------------\*/

int InsertSymbolTable(char \*Label, int LocationCounter) {

int Buffer;

// Symbol Table의 해당 hash값을 갖는 index의 Label이 비어 있으면

if (SymbolTable[(Buffer = HashFunction(Label))].LabelName == NULL) {

SymbolTable[Buffer].LabelName = (char\*)malloc(sizeof(char) \* (int)strlen(Label) + 1);

strcpy(SymbolTable[Buffer].LabelName, Label);

SymbolTable[Buffer].LabelName[strlen(Label)] = '\0';

SymbolTable[Buffer].LabelAddress = LocationCounter;

SymbolTable[Buffer].LabelFlag = 1;

SymbolTable[Buffer].Next = NULL;

}

else {

SYMBOL \*NewSymbol = NULL, \*TempSymbol = &(SymbolTable[Buffer]);

NewSymbol = (SYMBOL\*)malloc(sizeof(SYMBOL));

NewSymbol->LabelName = (char\*)malloc(sizeof(char) \* (int)strlen(Label) + 1);

strcpy(NewSymbol->LabelName, Label);

NewSymbol->LabelAddress = LocationCounter;

NewSymbol->LabelFlag = 1;

NewSymbol->Next = NULL;

while (TempSymbol->Next != NULL) {

TempSymbol = TempSymbol->Next;

}

TempSymbol->Next = NewSymbol;

}

return 0;

}

/\*------------------------------------------------------------------------------------\*/

/\*함수 : \*/

/\*목적 :

/\*리턴값 : \*/

/\*------------------------------------------------------------------------------------\*/

void DeleteSymbolTable() {

int Index;

SYMBOL \*TempSymbol = NULL, \*DeleteSymbol = NULL;

for (Index = 0; Index < 20; Index++) {

TempSymbol = &(SymbolTable[Index]);

if (TempSymbol->LabelName != NULL) {

free(TempSymbol->LabelName);

TempSymbol->LabelName = NULL;

if ((TempSymbol = TempSymbol->Next) != NULL) {

while (TempSymbol != NULL) {

DeleteSymbol = TempSymbol;

TempSymbol = TempSymbol->Next;

if (DeleteSymbol->LabelName != NULL) {

free(DeleteSymbol->LabelName);

}

free(DeleteSymbol);

}

}

}

}

return;

}

/\*------------------------------------------------------------------------------------\*/

/\*------------------------------------------------------------------------------------\*/

int SearchSymbolTable(char \*Label, SYMBOL \*\*TempPointer) {

int HashCode;

SYMBOL\* TempSymbol = NULL;

if (!((HashCode = HashFunction(Label)) >= 0 && HashCode < 20)) {

printf("An error occurs while reading symbol table.\n");

return -1;

}

if ((TempSymbol = &(SymbolTable[HashCode])) == NULL) {

printf("An error occurs while reading symbol table.\n");

return -1;

}

if (TempSymbol->LabelName != NULL) {

while ((strcmp(TempSymbol->LabelName, Label) != 0)) {

// Label을 Symbol Table에서 찾지 못하면

if (TempSymbol->Next == NULL) {

return 1;

}

TempSymbol = TempSymbol->Next;

}

}

else {

return 1;

}

(\*TempPointer) = TempSymbol;

// Label을 Symbol Table에서 찾으면

return 0;

}

/\*------------------------------------------------------------------------------------\*/

/\*함수 : \*/

/\*목적 :

/\*리턴값 : \*/

/\*------------------------------------------------------------------------------------\*/

int PrintSymbolTable() {

int Index, SymbolNumber = 0, TempLabelAddress = 0;

SYMBOL \*TempSymbol = NULL, \*SortedSymbolList = NULL;;

char \*TempLabelName = NULL;

for (Index = 0; Index < 20; Index++) {

TempSymbol = &(SymbolTable[Index]);

if (TempSymbol->LabelName != NULL) {

while (TempSymbol->Next != NULL) {

SymbolNumber++;

TempSymbol = TempSymbol->Next;

}

SymbolNumber++;

}

}

if (SymbolNumber) {

SortedSymbolList = (SYMBOL\*)malloc(sizeof(SYMBOL) \* SymbolNumber);

for (int i = 0; i < SymbolNumber; i++) {

SortedSymbolList[i].LabelName = NULL;

SortedSymbolList[i].LabelAddress = 0;

SortedSymbolList[i].Next = NULL;

}

for (Index = 0, SymbolNumber = 0; Index < 20; Index++) {

TempSymbol = &(SymbolTable[Index]);

if (TempSymbol->LabelName != NULL) {

while (TempSymbol->Next != NULL) {

SortedSymbolList[SymbolNumber].LabelName = (char\*)malloc(sizeof(char) \* (int)strlen(TempSymbol->LabelName) + 1);

strcpy(SortedSymbolList[SymbolNumber].LabelName, TempSymbol->LabelName);

SortedSymbolList[SymbolNumber++].LabelAddress = TempSymbol->LabelAddress;

TempSymbol = TempSymbol->Next;

}

SortedSymbolList[SymbolNumber].LabelName = (char\*)malloc(sizeof(char) \* (int)strlen(TempSymbol->LabelName) + 1);

strcpy(SortedSymbolList[SymbolNumber].LabelName, TempSymbol->LabelName);

SortedSymbolList[SymbolNumber++].LabelAddress = TempSymbol->LabelAddress;

}

}

if (SymbolNumber > 1) {

for (int i = 0; i < SymbolNumber; i++) {

for (int j = i + 1; j < SymbolNumber; j++) {

if (strcmp(SortedSymbolList[i].LabelName, SortedSymbolList[j].LabelName) < 0) {

TempLabelName = SortedSymbolList[i].LabelName;

SortedSymbolList[i].LabelName = SortedSymbolList[j].LabelName;

SortedSymbolList[j].LabelName = TempLabelName;

TempLabelAddress = SortedSymbolList[i].LabelAddress;

SortedSymbolList[i].LabelAddress = SortedSymbolList[j].LabelAddress;

SortedSymbolList[j].LabelAddress = TempLabelAddress;

}

}

}

}

for (int i = 0; i < SymbolNumber; i++) {

printf("\t%s\t%04X\n", SortedSymbolList[i].LabelName, SortedSymbolList[i].LabelAddress);

}

for (int i = 0; i < SymbolNumber; i++) {

free(SortedSymbolList[i].LabelName);

}

free(SortedSymbolList);

}

return 0;

}

/\*------------------------------------------------------------------------------------\*/

/\*------------------------------------------------------------------------------------\*/

int TypeCommand(char \*FileName) {

FILE \*FilePointer = NULL;

char BufferChar = EOF;

if ((FilePointer = fopen(FileName, "r")) == NULL) {

printf("An error occurs while opening the file you want to print out.\n");

return -1;

}

while ((BufferChar = fgetc(FilePointer)) != EOF) {

printf("%c", BufferChar);

}

printf("\n");

fclose(FilePointer);

return 0;

}

/\*------------------------------------------------------------------------------------\*/

/\*함수 : \*/

/\*목적 :

/\*리턴값 : \*/

/\*------------------------------------------------------------------------------------\*/

int FindRegisterNumber(char \*Register) {

int ReturnValue = -1;

if (!strcmp(Register, "A")) {

ReturnValue = 0;

}

else if (!strcmp(Register, "X")) {

ReturnValue = 1;

}

else if (!strcmp(Register, "L")) {

ReturnValue = 2;

}

else if (!strcmp(Register, "B")) {

ReturnValue = 3;

}

else if (!strcmp(Register, "S")) {

ReturnValue = 4;

}

else if (!strcmp(Register, "T")) {

ReturnValue = 5;

}

else if (!strcmp(Register, "F")) {

ReturnValue = 6;

}

else if (!strcmp(Register, "PC")) {

ReturnValue = 8;

}

else if (!strcmp(Register, "SW")) {

ReturnValue = 9;

}

return ReturnValue;

}

/\*프로그램 시작\*/

int main() {

char \*InputShellCommand = NULL;

char \*Parameter1 = NULL, \*Parameter2 = NULL, \*Parameter3 = NULL;

int ReadFlag = 0;

if (CreateHashTable() == -1) {

printf("Error occurs while creating hash table\n.");

return -1;

}

while ((ReadFlag = ReadShellCommand(&InputShellCommand, &Parameter1, &Parameter2, &Parameter3)) != Quit) {

if (ReadFlag != 0) {

switch (ReadFlag) {

case Help:

RecordPreviousCommand(InputShellCommand);

HelpCommand();

break;

case Dir:

RecordPreviousCommand(InputShellCommand);

DirCommand();

break;

case History:

RecordPreviousCommand(InputShellCommand);

HistoryCommand();

break;

case Dump:

if (!DumpCommand(Parameter1, Parameter2)) {

RecordPreviousCommand(InputShellCommand);

}

break;

case Edit:

if (!EditCommand(Parameter1, Parameter2)) {

RecordPreviousCommand(InputShellCommand);

}

break;

case Fill:

if (!FillCommand(Parameter1, Parameter2, Parameter3)) {

RecordPreviousCommand(InputShellCommand);

}

break;

case Reset:

if (!ResetCommand()) {

RecordPreviousCommand(InputShellCommand);

}

break;

case Opcode:

if (!OpcodeCommand(Parameter1)) {

RecordPreviousCommand(InputShellCommand);

}

break;

case OpcodeList:

if (!OpcodeListCommand()) {

RecordPreviousCommand(InputShellCommand);

}

break;

case Assemble:

if (!AssemblePassOne(Parameter1)) {

if (!AssemblePassTwo(Parameter1)) {

RecordPreviousCommand(InputShellCommand);

}

}

break;

case SymbolCommand:

if (!PrintSymbolTable()) {

RecordPreviousCommand(InputShellCommand);

}

break;

case Type:

if (!TypeCommand(Parameter1)) {

RecordPreviousCommand(InputShellCommand);

}

break;

default:

printf("Please write the vaild command. You can refer to the valid command list by executing 'help' command.\n");

break;

}

}

else {

printf("Please write the vaild command. You can refer to the valid command list by executing 'help' command.\n");

}

}

DeleteHistory();

DeleteHashTable();

DeleteSymbolTable();

if (InputShellCommand != NULL) {

free(InputShellCommand);

}

if (Parameter1 != NULL) {

free(Parameter1);

}

if (Parameter2 != NULL) {

free(Parameter2);

}

if (Parameter3 != NULL) {

free(Parameter3);

}

//\_CrtSetDbgFlag(\_CRTDBG\_ALLOC\_MEM\_DF | \_CRTDBG\_LEAK\_CHECK\_DF); // 메모리 누수 확인

return 0;

}

## AssemblePassOne.c

#include "20171665.h"

/\*------------------------------------------------------------------------------------\*/

/\*함수 : AssemblePassOne(char \*AssembleFileName)\*/

/\*목적 : Pass 1

/\*리턴값 : 1이면 성공, 0이면 실패\*/

/\*------------------------------------------------------------------------------------\*/

int AssemblePassOne(char \*AssembleFileName) {

FILE \*FilePointer = NULL, \*ImmediateFilePointer = NULL;

if ((FilePointer = fopen(AssembleFileName, "r")) == NULL) {

printf("An error occurs while opening the file you want to assemble.\n");

return -1;

}

char \*FileName = NULL, \*ImmediateFileName = NULL, \*ListFileName = NULL, \*ObjectFileName = NULL;

int FileNameLength = 0, ErrorFlag = 0;

if (strlen(AssembleFileName) <= 4) {

printf("Please write the valid file format. It should be '\*.asm'.\n");

return -1;

}

FileName = (char\*)malloc((FileNameLength = sizeof(char) \* (int)strlen(AssembleFileName) + 1) - 4);

ImmediateFileName = (char\*)malloc(FileNameLength);

strncpy(FileName, AssembleFileName, FileNameLength - 5);

FileName[FileNameLength - 5] = '\0';

snprintf(ImmediateFileName, FileNameLength, "%s.imm", FileName);

ImmediateFileName[FileNameLength - 1] = '\0';

if ((ImmediateFilePointer = fopen(ImmediateFileName, "w")) == NULL) {

printf("An error occurs while writing the immediate file.\n");

if (FileName != NULL) {

free(FileName);

}

if (ImmediateFileName != NULL) {

free(ImmediateFileName);

}

return -1;

}

ListFileName = (char\*)malloc(FileNameLength);

ObjectFileName = (char\*)malloc(FileNameLength);

snprintf(ListFileName, FileNameLength, "%s.lst", FileName);

ListFileName[FileNameLength - 1] = '\0';

snprintf(ObjectFileName, FileNameLength, "%s.obj", FileName);

ObjectFileName[FileNameLength - 1] = '\0';

int LabelCounter = 0, OpcodeMnemonicCounter = 0, OperandCounter = 0, AddLocationCounter = 0;

int StartingAddress = 0, LocationCounter = 0, LineNumber = 0;

int Flag = 0, NotAddressFlag = 0, StartFlag = 0, EndFlag = 0, CharacterFlag = 0, CommentFlag = 0, NowStartFlag = 0;

char BufferChar = EOF, BeforeChar = EOF, \*Label = NULL, \*OpcodeMnemonic = NULL, \*Operand = NULL;

// 기존의 Symbol Table을 삭제하고 초기화

DeleteSymbolTable();

InitializeSymbolTable();

while (!EndFlag) {

Label = (char\*)realloc(Label, sizeof(char) \* (LabelCounter + 1));

OpcodeMnemonic = (char\*)realloc(OpcodeMnemonic, sizeof(char) \* (OpcodeMnemonicCounter + 1));

Operand = (char\*)realloc(Operand, sizeof(char) \* (OperandCounter + 1));

LineNumber += 5;

while ((BufferChar = fgetc(FilePointer)) != '\n') {

// Line의 첫 character로 '.'를 읽으면 다음 line으로 넘어간다.

if ((LabelCounter == 0) && (BufferChar == '.')) {

fprintf(ImmediateFilePointer, "%c", BufferChar);

while ((BufferChar = fgetc(FilePointer)) != '\n') {

fprintf(ImmediateFilePointer, "%c", BufferChar);

}

fprintf(ImmediateFilePointer, "%c", BufferChar);

CommentFlag = 1;

break;

}

else if ((BufferChar == ' ' || BufferChar == '\t') && CharacterFlag == 0) {

// 처음부터 공백이 들어온 경우

if (Flag == 0) {

Flag = 1;

}

// Opcode를 모두 읽고 공백이 들어온 경우

else if (OpcodeMnemonicCounter && (Flag == 1)) {

Flag = 2;

}

continue;

}

else if (BufferChar == EOF) {

break;

}

else {

if (BufferChar == '\'' && Flag == 2) {

if (BeforeChar == 'C' && CharacterFlag == 0) {

CharacterFlag = 1;

}

else if (CharacterFlag == 1) {

CharacterFlag = 0;

}

}

BeforeChar = BufferChar;

switch (Flag) {

case 0:

Label = (char\*)realloc(Label, sizeof(char) \* (LabelCounter + 2));

Label[LabelCounter++] = BufferChar;

break;

case 1:

OpcodeMnemonic = (char\*)realloc(OpcodeMnemonic, sizeof(char) \* (OpcodeMnemonicCounter + 2));

OpcodeMnemonic[OpcodeMnemonicCounter++] = BufferChar;

break;

case 2:

Operand = (char\*)realloc(Operand, sizeof(char) \* (OperandCounter + 2));

Operand[OperandCounter++] = BufferChar;

break;

}

}

}

if (CommentFlag) {

CommentFlag = 0;

continue;

}

Label[LabelCounter] = OpcodeMnemonic[OpcodeMnemonicCounter] = Operand[OperandCounter] = '\0';

RemoveBlank(&Label);

RemoveBlank(&OpcodeMnemonic);

RemoveBlank(&Operand);

//printf("Label: [%s] Opcode: [%s] Operand: [%s]\n", Label, OpcodeMnemonic, Operand);

if (OpcodeMnemonic != NULL) {

// 첫 번째 line의 opcode가 'START'이면

if (StartFlag == 0) {

if (strcmp(OpcodeMnemonic, "START") == 0) {

if (Operand != NULL) {

if ((StartingAddress = HexStringToDec(Operand, Integer)) == -1) {

printf("[Line: %d] A starting address operand is not valid.\n", LineNumber);

break;

}

LocationCounter = StartingAddress;

}

else {

LocationCounter = 0;

}

}

// 첫 번? 줄의 OPCODE가 START가 아닌 경우

else {

ErrorFlag = 1;

printf("[Line %d(%dth line)] An opcode of the first line is not 'START'.\n", LineNumber, LineNumber / 5);

break;

}

}

HASH \*FoundHash = NULL;

switch (SearchOpcode(OpcodeMnemonic, &FoundHash)) {

// Opcode Table에서 Opcode를 찾으면

case 0:

if (strcmp(FoundHash->Format, "1") == 0) {

AddLocationCounter = 1;

}

else if (strcmp(FoundHash->Format, "2") == 0) {

AddLocationCounter = 2;

}

else if (strcmp(FoundHash->Format, "3/4") == 0) {

if (OpcodeMnemonic[0] != '+') {

AddLocationCounter = 3;

}

else {

AddLocationCounter = 4;

}

}

// Opcode에 맞는 foramt이 없는 경우

else {

ErrorFlag = 1;

printf("[Line %d(%dth line)] There is no valid format that is associated with the opcode.\n", LineNumber, LineNumber / 5);

break;

}

break;

// Opcode Table에서 Opcode를 찾지 못하면

case 1:

if (Operand != NULL) {

if (strcmp(OpcodeMnemonic, "WORD") == 0) {

AddLocationCounter = 3;

}

else if (strcmp(OpcodeMnemonic, "RESW") == 0) {

int OperandToIntegerValue;

if ((OperandToIntegerValue = DecStringToDec(Operand, Integer)) >= 0) {

AddLocationCounter = 3 \* OperandToIntegerValue;

}

else {

ErrorFlag = 1;

printf("[Line %d(%dth line)] Please check the operand of 'RESW'. The valid range is unsigned integer.\n", LineNumber, LineNumber / 5);

break;

}

}

else if (strcmp(OpcodeMnemonic, "RESB") == 0) {

int OperandToIntegerValue;

if ((OperandToIntegerValue = DecStringToDec(Operand, Integer)) >= 0) {

AddLocationCounter = OperandToIntegerValue;

}

else {

ErrorFlag = 1;

printf("[Line %d(%dth line)] Please check the operand of 'RESB'. The valid range is unsigned integer.\n", LineNumber, LineNumber / 5);

break;

}

}

else if (strcmp(OpcodeMnemonic, "BYTE") == 0) {

if (Operand[1] == '\'') {

int DigitNumber = 0;

for (int i = 2; Operand[i] != '\''; i++) {

DigitNumber++;

}

if (Operand[0] == 'X') {

if (DigitNumber > 60) {

ErrorFlag = 1;

printf("[Line %d(%dth line)] The length of byte you want to write is too long. ", LineNumber, LineNumber / 5);

printf("It should be shorter than 60 digits.\n");

break;

}

AddLocationCounter = (DigitNumber + 1) / 2;

}

else if (Operand[0] == 'C') {

if (DigitNumber > 30) {

ErrorFlag = 1;

printf("[Line %d(%dth line)] The length of byte you want to write is too long. ", LineNumber, LineNumber / 5);

printf("It should be shorter than 30 characters.\n");

break;

}

AddLocationCounter = DigitNumber;

}

}

}

else if (!strcmp(OpcodeMnemonic, "END") || !strcmp(OpcodeMnemonic, "BASE") || !strcmp(OpcodeMnemonic, "NOBASE")) {

NotAddressFlag = 1;

}

else if (!strcmp(OpcodeMnemonic, "START")) {

if (StartFlag == 1) {

ErrorFlag = 1;

printf("[Line %d(%dth line)] 'START' opcode appears twice in the file.\n", LineNumber, LineNumber / 5);

}

else {

StartFlag = 1;

NowStartFlag = 1;

}

}

else {

ErrorFlag = 1;

printf("[Line %d(%dth line)] There is no corresponding opcode or directive.\n", LineNumber, LineNumber / 5);

break;

}

}

else {

if (!strcmp(OpcodeMnemonic, "START")) {

if (StartFlag == 1) {

ErrorFlag = 1;

printf("[Line %d(%dth line)] 'START' opcode appears twice in the file.\n", LineNumber, LineNumber / 5);

}

else {

StartFlag = 1;

NowStartFlag = 1;

}

break;

}

else if (!strcmp(OpcodeMnemonic, "END")) {

NotAddressFlag = 1;

}

else {

ErrorFlag = 1;

printf("[Line %d(%dth line)] The opcode or directive should have an operand.\n", LineNumber, LineNumber / 5);

break;

}

}

break;

default:

break;

}

if (ErrorFlag == 1) {

break;

}

if (Label != NULL && !NowStartFlag) {

SYMBOL \*TempPointer = NULL;

// Label을 Symbol Table에서 찾으면

if (SearchSymbolTable(Label, &TempPointer) == 0) {

ErrorFlag = 1;

printf("[Line %d(%dth line)] A label has already existed in the symbol table.\n", LineNumber, LineNumber / 5);

break;

}

// Label을 Symbol Table에서 찾지 못하면

else if (SearchSymbolTable(Label, &TempPointer) == 1) {

InsertSymbolTable(Label, LocationCounter);

}

}

if (strcmp(OpcodeMnemonic, "END") == 0) {

if (EndFlag == 0) {

EndFlag = 1;

}

else {

ErrorFlag = 1;

printf("[Line %d(%dth line)] 'END' opcode appears twice in the file.\n", LineNumber, LineNumber / 5);

break;

}

}

if (NowStartFlag) {

NowStartFlag = 0;

}

}

if (OpcodeMnemonicCounter) {

if (!NotAddressFlag) {

fprintf(ImmediateFilePointer, "%04X\t", LocationCounter & 0xFFFF);

}

else {

fprintf(ImmediateFilePointer, "\t");

}

if (Label != NULL) {

fprintf(ImmediateFilePointer, "%s\t", Label);

}

else {

fprintf(ImmediateFilePointer, "\t");

}

if (OpcodeMnemonic != NULL) {

fprintf(ImmediateFilePointer, "%s\t", OpcodeMnemonic);

}

else {

fprintf(ImmediateFilePointer, "\t");

}

if (Operand != NULL) {

fprintf(ImmediateFilePointer, "%s\t", Operand);

}

else {

fprintf(ImmediateFilePointer, "\t");

}

fprintf(ImmediateFilePointer, "\n");

}

LocationCounter += AddLocationCounter;

LabelCounter = AddLocationCounter = OpcodeMnemonicCounter = OperandCounter = 0;

Flag = CharacterFlag = CommentFlag = 0, NotAddressFlag = 0;

if (BufferChar == EOF) {

if (EndFlag == 0) {

ErrorFlag = 1;

printf("[Line %d(%dth line)] There is no 'END' operand.\n", LineNumber, LineNumber / 5);

}

break;

}

}

ProgramLength = LocationCounter - (StartAddress = StartingAddress);

fclose(FilePointer);

fclose(ImmediateFilePointer);

if (ErrorFlag == 1) {

DeleteSymbolTable();

remove(ImmediateFileName);

remove(ListFileName);

remove(ObjectFileName);

}

if (Label != NULL) {

free(Label);

}

if (OpcodeMnemonic != NULL) {

free(OpcodeMnemonic);

}

if (Operand != NULL) {

free(Operand);

}

if (FileName != NULL) {

free(FileName);

}

if (ImmediateFileName != NULL) {

free(ImmediateFileName);

}

if (ListFileName != NULL) {

free(ListFileName);

}

if (ObjectFileName != NULL) {

free(ObjectFileName);

}

if (ErrorFlag == 1) {

return -1;

}

return 0;

}

## AssemblePassTwo.c

#include "20171665.h"

/\*------------------------------------------------------------------------------------\*/

/\*함수 : AssemblePassTwo(char \*AssembleFileName)\*/

/\*목적 : Pass 2 시행

/\*리턴값 : 0은 성공, -1은 실패\*/

/\*------------------------------------------------------------------------------------\*/

int AssemblePassTwo(char \*AssembleFileName) {

FILE \*ImmediateFilePointer = NULL, \*ListFilePointer = NULL, \*ObjectFilePointer = NULL;

char \*FileName = NULL, \*ImmediateFileName = NULL, \*ListFileName = NULL, \*ObjectFileName = NULL;

int FileNameLength = 0, ErrorFlag = 0;

if (strlen(AssembleFileName) <= 4) {

return -1;

}

FileName = (char\*)malloc((FileNameLength = sizeof(char) \* (int)strlen(AssembleFileName) + 1) - 4);

strncpy(FileName, AssembleFileName, FileNameLength - 5);

FileName[FileNameLength - 5] = '\0';

ImmediateFileName = (char\*)malloc(FileNameLength);

snprintf(ImmediateFileName, FileNameLength, "%s.imm", FileName);

ImmediateFileName[FileNameLength - 1] = '\0';

if ((ImmediateFilePointer = fopen(ImmediateFileName, "r")) == NULL) {

printf("An error occurs while opening the file you want to assemble.\n");

if (FileName != NULL) {

free(FileName);

}

if (ImmediateFileName != NULL) {

free(ImmediateFileName);

}

return -1;

}

ListFileName = (char\*)malloc(FileNameLength);

ObjectFileName = (char\*)malloc(FileNameLength);

snprintf(ListFileName, FileNameLength, "%s.lst", FileName);

ListFileName[FileNameLength - 1] = '\0';

snprintf(ObjectFileName, FileNameLength, "%s.obj", FileName);

ObjectFileName[FileNameLength - 1] = '\0';

if ((ListFilePointer = fopen(ListFileName, "w")) == NULL) {

printf("An error occurs while writing the list file.\n");

return -1;

}

if ((ObjectFilePointer = fopen(ObjectFileName, "w")) == NULL) {

printf("An error occurs while writing the list file.\n");

return -1;

}

int HexAddressCounter = 0, LabelCounter = 0, OpcodeMnemonicCounter = 0, OperandCounter = 0, LineNumber = 0, TempBuffer = 0;

int AddLocationCounter = 0, BaseRegister = 0, ProgramCounter = 0, RelocationCounter = 0, RealOperand = 0;

int Flag = 0, NIFlag = 0, BaseFlag = 0, IndexFlag = 0, MemoryFlag = 0, ReserveFlag = 0, CommentFlag = 0;

char BufferChar = EOF, \*HexAddress = NULL, \*Label = NULL, \*OpcodeMnemonic = NULL, \*Operand = NULL, \*TempOperand = NULL;

char HexCode[12] = { '\0' }, HexCodeOpcodeAndNI[3] = { '\0' }, HexCodeXBPE[2] = { '\0' }, HexCodeOperand[6] = { '\0' };

char FirstRegister[3] = { '\0' }, SecondRegister[3] = { '\0' }, MemoryContent[200] = { '\0' }, ProgramName[7] = { '\0' };

char TextRecordStartAddress[7] = { '\0' }, TextRecordObjectCode[61] = { '\0' }, ModifiedAddress[10000][7] = { '\0' };

int TextRecordCounter = 0;

while (1) {

HexAddress = (char\*)realloc(HexAddress, sizeof(char) \* (HexAddressCounter + 1));

Label = (char\*)realloc(Label, sizeof(char) \* (LabelCounter + 1));

OpcodeMnemonic = (char\*)realloc(OpcodeMnemonic, sizeof(char) \* (OpcodeMnemonicCounter + 1));

Operand = (char\*)realloc(Operand, sizeof(char) \* (OperandCounter + 1));

TempOperand = (char\*)realloc(TempOperand, sizeof(char) \* (OperandCounter + 1));

LineNumber += 5;

while ((BufferChar = fgetc(ImmediateFilePointer)) != '\n') {

// Line의 첫 character로 '.'를 읽으면

if ((LabelCounter == 0) && (BufferChar == '.')) {

fprintf(ListFilePointer, "\t%c", BufferChar);

while ((BufferChar = fgetc(ImmediateFilePointer)) != '\n') {

fprintf(ListFilePointer, "%c", BufferChar);

}

fprintf(ListFilePointer, "%c", BufferChar);

CommentFlag = 1;

break;

}

else if (BufferChar == '\t') {

// 첫 character로 공백이 들어온 경우

if (Flag == 0) {

Flag = 1;

}

// Label과 Opcode를 모두 읽고 공백이 들어온 경우

else if (Flag == 1) {

Flag = 2;

}

else if (OpcodeMnemonicCounter && (Flag == 2)) {

Flag = 3;

}

continue;

}

else if (BufferChar == EOF) {

break;

}

else {

switch (Flag) {

case 0:

HexAddress = (char\*)realloc(HexAddress, sizeof(char) \* (HexAddressCounter + 2));

HexAddress[HexAddressCounter++] = BufferChar;

break;

case 1:

Label = (char\*)realloc(Label, sizeof(char) \* (LabelCounter + 2));

Label[LabelCounter++] = BufferChar;

break;

case 2:

OpcodeMnemonic = (char\*)realloc(OpcodeMnemonic, sizeof(char) \* (OpcodeMnemonicCounter + 2));

OpcodeMnemonic[OpcodeMnemonicCounter++] = BufferChar;

break;

case 3:

Operand = (char\*)realloc(Operand, sizeof(char) \* (OperandCounter + 2));

Operand[OperandCounter++] = BufferChar;

break;

}

}

}

if (CommentFlag) {

CommentFlag = 0;

continue;

}

HexAddress[HexAddressCounter] = Label[LabelCounter] = OpcodeMnemonic[OpcodeMnemonicCounter] = Operand[OperandCounter] = '\0';

TempOperand = (char\*)realloc(TempOperand, sizeof(char) \* OperandCounter + 1);

//printf("Label: [%s] Opcode: [%s] Operand: [%s]\n", Label, OpcodeMnemonic, Operand);

RemoveBlank(&HexAddress);

RemoveBlank(&Label);

RemoveBlank(&OpcodeMnemonic);

RemoveBlank(&Operand);

if (OpcodeMnemonic != NULL) {

if (strcmp(OpcodeMnemonic, "START") == 0) {

if (Label != NULL) {

strncpy(ProgramName, Label, sizeof(ProgramName) - 1);

ProgramName[6] = '\0';

}

fprintf(ObjectFilePointer, "H%-6s%06X%06X\n", ProgramName, StartAddress & 0XFFFFFF, ProgramLength & 0XFFFFFF);

}

HASH \*FoundHash = NULL;

switch (SearchOpcode(OpcodeMnemonic, &FoundHash)) {

// Opcode Table에서 Opcode를 찾으면

case 0:

if (strcmp(FoundHash->Format, "1") == 0) {

AddLocationCounter = 1;

TempBuffer = HexStringToDec(FoundHash->Number, Integer);

snprintf(HexCodeOpcodeAndNI, sizeof(char) \* 3, "%02X", TempBuffer & 0xFF);

sprintf(HexCode, "%s", HexCodeOpcodeAndNI);

}

else if (strcmp(FoundHash->Format, "2") == 0) {

AddLocationCounter = 2;

// SHIFTL, SHIFTR Opcode가 아닐 때

if (Operand != NULL) {

if ((strcmp(OpcodeMnemonic, "SHIFTL") != 0) && (strcmp(OpcodeMnemonic, "SHIFTR") != 0) && (strcmp(OpcodeMnemonic, "SVC") != 0)) {

int RegisterFlag = 0, FirstRegisterValue = 0, SecondRegisterValue = 0;

TempBuffer = HexStringToDec(FoundHash->Number, Integer);

snprintf(HexCodeOpcodeAndNI, sizeof(char) \* 3, "%02X", TempBuffer & 0xFF);

for (int i = 0; i < OperandCounter; i++) {

if (Operand[i] == ',') {

strncpy(FirstRegister, Operand, sizeof(char) \* i);

strncpy(SecondRegister, Operand + i + 1, sizeof(SecondRegister) - 1);

RegisterFlag = 1;

break;

}

}

if (!RegisterFlag) {

if (strcmp(OpcodeMnemonic, "CLEAR") == 0 || strcmp(OpcodeMnemonic, "TIXR") == 0) {

if ((FirstRegisterValue = FindRegisterNumber(Operand)) != -1) {

snprintf(FirstRegister, sizeof(char) \* 3, "%01X", FirstRegisterValue & 0xF);

}

else {

ErrorFlag = 1;

printf("[Line %d(%dth line)] Please write one when you write 'CLEAR' and 'TIXR' operand.\n", LineNumber, LineNumber / 5);

break;

}

sprintf(HexCode, "%s%s0", HexCodeOpcodeAndNI, FirstRegister);

}

else {

ErrorFlag = 1;

printf("[Line %d(%dth line)] Please write two registers except for 'CLEAR' and 'TIXR' operand.\n", LineNumber, LineNumber / 5);

break;

}

}

else {

if (strcmp(OpcodeMnemonic, "CLEAR") != 0 && strcmp(OpcodeMnemonic, "TIXR") != 0) {

if ((FirstRegisterValue = FindRegisterNumber(FirstRegister)) != -1 && (SecondRegisterValue = FindRegisterNumber(SecondRegister)) != -1) {

snprintf(FirstRegister, sizeof(char) \* 3, "%01X", FirstRegisterValue & 0xF);

snprintf(SecondRegister, sizeof(char) \* 3, "%01X", SecondRegisterValue & 0xF);

}

else {

ErrorFlag = 1;

printf("[Line %d(%dth line)] Please write two registers except for 'CLEAR' and 'TIXR' operand.\n", LineNumber, LineNumber / 5);

break;

}

sprintf(HexCode, "%s%s%s", HexCodeOpcodeAndNI, FirstRegister, SecondRegister);

}

else {

ErrorFlag = 1;

printf("[Line %d(%dth line)] Please write one register when you write 'CLEAR' and 'TIXR' operand.\n", LineNumber, LineNumber / 5);

break;

}

}

}

// SVC일 때

else if (strcmp(OpcodeMnemonic, "SVC") == 0) {

int Nbit = 0;

TempBuffer = HexStringToDec(FoundHash->Number, Integer);

snprintf(HexCodeOpcodeAndNI, sizeof(char) \* 3, "%02X", TempBuffer & 0xFF);

if (Nbit = DecStringToDec(Operand, Integer)) {

sprintf(HexCode, "%s%01X0", HexCodeOpcodeAndNI, Nbit & 0xF);

}

else {

ErrorFlag = 1;

printf("[Line %d(%dth line)] Please check the format of N bits.\n", LineNumber, LineNumber / 5);

break;

}

}

// SHIFTL 또는 SHIFTR Opcode일 때

else {

int RegisterValue = 0, Nbit = 0;

TempBuffer = HexStringToDec(FoundHash->Number, Integer);

snprintf(HexCodeOpcodeAndNI, sizeof(char) \* 3, "%02X", TempBuffer & 0xFF);

Nbit = DecStringToDec(Operand, Integer);

for (int i = 0; i < OperandCounter; i++) {

if (Operand[i] == ',') {

strncpy(FirstRegister, Operand, sizeof(char) \* i);

strncpy(SecondRegister, Operand + i + 1, sizeof(SecondRegister) - 1);

break;

}

}

if ((RegisterValue = FindRegisterNumber(FirstRegister)) != -1 && (Nbit = DecStringToDec(SecondRegister, Integer) - 1) >= 0) {

snprintf(FirstRegister, sizeof(char) \* 3, "%01X", RegisterValue & 0xF);

snprintf(SecondRegister, sizeof(char) \* 3, "%01X", Nbit & 0xF);

}

else {

ErrorFlag = 1;

printf("[Line %d(%dth line)] Please check the line. The valid format is SHIFTL r, n (or SHIFTR r, n).\n", LineNumber, LineNumber / 5);

break;

}

sprintf(HexCode, "%s%s%s", HexCodeOpcodeAndNI, FirstRegister, SecondRegister);

}

}

else {

TempBuffer = HexStringToDec(FoundHash->Number, Integer);

snprintf(HexCodeOpcodeAndNI, sizeof(char) \* 3, "%02X", TempBuffer & 0xFF);

sprintf(HexCode, "%s00", HexCodeOpcodeAndNI);

}

}

else if (strcmp(FoundHash->Format, "3/4") == 0) {

AddLocationCounter = 3;

// Format 3일 때

if (OpcodeMnemonic[0] != '+') {

SYMBOL \*TempSymbol = NULL;

char \*TempOperandPointer = NULL;

TempBuffer = HexStringToDec(FoundHash->Number, Integer);

if (Operand != NULL) {

for (int i = 0; i < OperandCounter; i++) {

if (i == OperandCounter - 2) {

if (Operand[i] == ',' && Operand[i + 1] == 'X') {

strncpy(TempOperand, Operand, sizeof(char) \* i);

TempOperand[i] = '\0';

IndexFlag = 8;

}

}

}

if (!IndexFlag) {

strncpy(TempOperand, Operand, sizeof(char) \* OperandCounter + 1);

TempOperand[OperandCounter] = '\0';

}

if (Operand[0] == '@') {

TempBuffer += 2;

NIFlag = IndirectAddressing;

}

else if (Operand[0] == '#') {

TempBuffer += 1;

NIFlag = ImmediateAddressing;

}

else {

TempBuffer += 3;

}

snprintf(HexCodeOpcodeAndNI, sizeof(char) \* 3, "%02X", TempBuffer & 0xFF);

ProgramCounter = HexStringToDec(HexAddress, ForProgramCounter) + AddLocationCounter;

if (NIFlag) {

if (IndexFlag) {

// Index Addressing은 Immediate 또는 Indirect Addressing과 함께 쓰일 수 없으므로 (출처: 번역본 교재 1.3장 13페이지)

ErrorFlag = 1;

printf("[Line %d(%dth line)] An index addressing cannot use with immediate or indirect addressing.\n", LineNumber, LineNumber / 5);

break;

}

TempOperandPointer = &(TempOperand[1]);

}

else {

TempOperandPointer = TempOperand;

}

if (SearchSymbolTable(TempOperandPointer, &TempSymbol) == 0) {

RealOperand = TempSymbol->LabelAddress - ProgramCounter;

if (RealOperand >= -2048 && RealOperand <= 2047) {

snprintf(HexCodeXBPE, sizeof(char) \* 2, "%01X", (TempBuffer = 2 + IndexFlag) & 0xF);

}

else {

if (BaseFlag) {

if ((RealOperand = TempSymbol->LabelAddress - BaseRegister) >= 0 && RealOperand <= 4095) {

//printf("TA: %d / BaseRegister: %d / Displacement: %d\n", TempSymbol->LabelAddress, BaseRegister, RealOperand);

snprintf(HexCodeXBPE, sizeof(char) \* 2, "%01X", (TempBuffer = 4 + IndexFlag) & 0xF);

}

else {

ErrorFlag = 1;

printf("[Line %d(%dth line)] The relative addresssing is not valid because of an excess of the range. ", LineNumber, LineNumber / 5);

printf("You can use an immediate addressing by writing '+' at the front of the opcode.\n");

break;

}

}

else {

ErrorFlag = 1;

printf("[Line %d(%dth line)] Please write the 'BASE' directive forward.\n", LineNumber, LineNumber / 5);

break;

}

}

}

else if (NIFlag == ImmediateAddressing) {

RealOperand = DecStringToDec(TempOperandPointer, ImmediateAddressing);

snprintf(HexCodeXBPE, sizeof(char) \* 2, "%01X", (TempBuffer = 0) & 0xF);

}

else {

ErrorFlag = 1;

printf("[Line %d(%dth line)] Please check the line. You should write '#' in the front of the instruction when you want to use an immediate addressing. ", LineNumber, LineNumber / 5);

printf("Otherwise, you should write the valid symbol.\n");

break;

}

snprintf(HexCodeOperand, sizeof(char) \* 4, "%03X", RealOperand & 0xFFF);

sprintf(HexCode, "%s%s%s", HexCodeOpcodeAndNI, HexCodeXBPE, HexCodeOperand);

}

else {

TempBuffer = HexStringToDec(FoundHash->Number, Integer);

snprintf(HexCodeOpcodeAndNI, sizeof(char) \* 3, "%02X", (TempBuffer += 3) & 0xFF);

sprintf(HexCode, "%s0000", HexCodeOpcodeAndNI);

}

}

// Format 4일 때

else {

AddLocationCounter = 4;

SYMBOL \*TempSymbol = NULL;

char \*TempOperandPointer = NULL;

TempBuffer = HexStringToDec(FoundHash->Number, Integer);

if (Operand != NULL) {

for (int i = 0; i < OperandCounter; i++) {

if (i == OperandCounter - 2) {

if (Operand[i] == ',' && Operand[i + 1] == 'X') {

strncpy(TempOperand, Operand, sizeof(char) \* i);

TempOperand[i] = '\0';

IndexFlag = 8;

}

}

}

if (!IndexFlag) {

strncpy(TempOperand, Operand, sizeof(char) \* OperandCounter + 1);

TempOperand[OperandCounter] = '\0';

}

if (Operand[0] == '@') {

TempBuffer += 2;

NIFlag = IndirectAddressing;

}

if (Operand[0] == '#') {

TempBuffer += 1;

NIFlag = ImmediateAddressing;

}

else {

TempBuffer += 3;

}

snprintf(HexCodeOpcodeAndNI, sizeof(char) \* 3, "%02X", TempBuffer & 0xFF);

if (NIFlag) {

if (IndexFlag) {

// Index Addressing은 Immediate 또는 Indirect Addressing과 함께 쓰일 수 없으므로 (출처: 번역본 교재 1.3장 13페이지)

ErrorFlag = 1;

printf("[Line %d(%dth line)] An index Addressing cannot use with immediate or indirect addressing.\n", LineNumber, LineNumber / 5);

break;

}

TempOperandPointer = &(TempOperand[1]);

}

else {

TempOperandPointer = TempOperand;

}

if (SearchSymbolTable(TempOperandPointer, &TempSymbol) == 0) {

RealOperand = TempSymbol->LabelAddress;

snprintf(HexCodeXBPE, sizeof(char) \* 2, "%01X", (TempBuffer = 1 + IndexFlag) & 0xF);

sprintf(ModifiedAddress[RelocationCounter++], "%06X", (HexStringToDec(HexAddress, ForProgramCounter) - StartAddress + 1) & 0xFFFFFF);

}

else if (NIFlag == ImmediateAddressing) {

RealOperand = DecStringToDec(TempOperandPointer, ImmediateAddressing);

snprintf(HexCodeXBPE, sizeof(char) \* 2, "%01X", (TempBuffer = 1) & 0xF);

}

else {

ErrorFlag = 1;

printf("[Line %d(%dth line)] Please check the line. You should write '#' in the front of the instruction when you want to use an immediate addressing. ", LineNumber, LineNumber / 5);

printf("Otherwise, you should write the valid symbol.\n");

break;

}

snprintf(HexCodeOperand, sizeof(char) \* 6, "%05X", RealOperand & 0xFFFFF);

sprintf(HexCode, "%s%s%s", HexCodeOpcodeAndNI, HexCodeXBPE, HexCodeOperand);

}

else {

TempBuffer = HexStringToDec(FoundHash->Number, Integer);

snprintf(HexCodeOpcodeAndNI, sizeof(char) \* 3, "%02X", (TempBuffer += 3) & 0xFF);

snprintf(HexCodeXBPE, sizeof(char) \* 2, "%01X", (TempBuffer = 1) & 0xF);

snprintf(HexCodeOperand, sizeof(char) \* 6, "00000");

sprintf(HexCode, "%s%s%s", HexCodeOpcodeAndNI, HexCodeXBPE, HexCodeOperand);

}

}

}

else {

}

break;

// Opcode Table에서 Opcode를 찾지 못하면

case 1:

if (Operand != NULL) {

if (strcmp(OpcodeMnemonic, "WORD") == 0) {

MemoryFlag = Word;

sprintf(MemoryContent, "%06X", DecStringToDec(Operand, Word) & 0xFFFFFF);

AddLocationCounter = 3;

}

else if (strcmp(OpcodeMnemonic, "RESW") == 0) {

ReserveFlag = 1;

}

else if (strcmp(OpcodeMnemonic, "RESB") == 0) {

ReserveFlag = 1;

}

else if (strcmp(OpcodeMnemonic, "BYTE") == 0) {

if (Operand[1] == '\'') {

int DigitNumber = 0;

if (Operand[0] == 'X') {

MemoryFlag = Hexadecimal;

for (int i = 2; Operand[i] != '\''; i++) {

DigitNumber++;

MemoryContent[i - 2] = Operand[i];

}

AddLocationCounter = (DigitNumber + 1) / 2;

}

else if (Operand[0] == 'C') {

MemoryFlag = Character;

for (int i = 2; Operand[i] != '\''; i++) {

DigitNumber++;

sprintf(MemoryContent, "%s%02X", MemoryContent, Operand[i] & 0xFF);

}

AddLocationCounter = DigitNumber;

}

}

}

else if (strcmp(OpcodeMnemonic, "BASE") == 0) {

SYMBOL \*BaseSymbol = NULL;

if (SearchSymbolTable(Operand, &BaseSymbol) == 0) {

BaseRegister = BaseSymbol->LabelAddress;

BaseFlag = 1;

}

}

else if (strcmp(OpcodeMnemonic, "NOBASE") == 0) {

BaseRegister = 0;

BaseFlag = 0;

}

}

break;

default:

break;

}

if (ErrorFlag == 1) {

break;

}

if (strcmp(OpcodeMnemonic, "END") == 0) {

if (TextRecordCounter) {

fprintf(ObjectFilePointer, "T%-6s%02X%-30s\n", TextRecordStartAddress, TextRecordCounter & 0xFF, TextRecordObjectCode);

}

for (int i = 0; i < RelocationCounter; i++) {

fprintf(ObjectFilePointer, "M%s05\n", ModifiedAddress[i]);

}

if (Operand != NULL) {

SYMBOL \*StartAddressSymbol = NULL;

if (SearchSymbolTable(Operand, &StartAddressSymbol) == 0) {

fprintf(ObjectFilePointer, "E%06X\n", StartAddressSymbol->LabelAddress & 0xFFFFFF);

}

else {

ErrorFlag = 1;

printf("[Line %d(%dth line)] The label of starting address isn't found.\n", LineNumber, LineNumber / 5);

break;

}

}

else {

fprintf(ObjectFilePointer, "E%06X\n", StartAddress & 0xFFFFFF);

}

}

}

if (ErrorFlag == 1) {

break;

}

if (Flag) {

// LOCCTR

if (HexAddress != NULL) {

fprintf(ListFilePointer, "%s\t", HexAddress);

}

else {

fprintf(ListFilePointer, "\t");

}

// SYMBOL

if (Label != NULL) {

fprintf(ListFilePointer, "%s\t", Label);

}

else {

fprintf(ListFilePointer, "\t");

}

// Opcode Mnemonic

if (OpcodeMnemonic != NULL) {

// Object Code를 작성했을 때 Text Record의 길이가 30 byte를 초과하거나 예약어에 의해서 로더에 업로드될 데이터가 있을 때

if ((TextRecordCounter + AddLocationCounter > 30) || (ReserveFlag && TextRecordCounter)) {

fprintf(ObjectFilePointer, "T%-6s%02X%-30s\n", TextRecordStartAddress, TextRecordCounter & 0xFF, TextRecordObjectCode);

memset(TextRecordStartAddress, '\0', sizeof(TextRecordStartAddress));

memset(TextRecordObjectCode, '\0', sizeof(TextRecordObjectCode));

TextRecordCounter = 0;

}

fprintf(ListFilePointer, "%s\t", OpcodeMnemonic);

}

else {

fprintf(ListFilePointer, "\t");

}

// Operand

if (Operand != NULL) {

fprintf(ListFilePointer, "%s\t", Operand);

}

else {

fprintf(ListFilePointer, "\t");

}

// ObjectCode

if (HexCode[0] != '\0' || MemoryFlag) {

if (!TextRecordCounter && !ReserveFlag) {

sprintf(TextRecordStartAddress, "%06X", HexStringToDec(HexAddress, ForProgramCounter) & 0xFFFFFF);

}

if (HexCode[0] != '\0' && !MemoryFlag) {

fprintf(ListFilePointer, "%s\t", HexCode);

sprintf(TextRecordObjectCode, "%s%s", TextRecordObjectCode, HexCode);

}

else if (MemoryFlag) {

fprintf(ListFilePointer, "%s\t", MemoryContent);

sprintf(TextRecordObjectCode, "%s%s", TextRecordObjectCode, MemoryContent);

}

TextRecordCounter += AddLocationCounter;

}

else {

fprintf(ListFilePointer, "\t");

}

fprintf(ListFilePointer, "\n");

}

HexAddressCounter = LabelCounter = OpcodeMnemonicCounter = OperandCounter = AddLocationCounter = 0;

ReserveFlag = IndexFlag = NIFlag = MemoryFlag = Flag = 0;

memset(HexCode, '\0', sizeof(HexCode));

memset(HexCodeOpcodeAndNI, '\0', sizeof(HexCodeOpcodeAndNI));

memset(HexCodeXBPE, '\0', sizeof(HexCodeXBPE));

memset(HexCodeOperand, '\0', sizeof(HexCodeOperand));

memset(FirstRegister, '\0', sizeof(FirstRegister));

memset(SecondRegister, '\0', sizeof(SecondRegister));

memset(MemoryContent, '\0', sizeof(MemoryContent));

if (BufferChar == EOF) {

break;

}

}

fclose(ImmediateFilePointer);

fclose(ListFilePointer);

fclose(ObjectFilePointer);

remove(ImmediateFileName);

if (ErrorFlag == 1) {

DeleteSymbolTable();

remove(ListFileName);

remove(ObjectFileName);

}

else {

printf("output file: [%s], [%s]\n", ListFileName, ObjectFileName);

}

if (HexAddress != NULL) {

free(HexAddress);

}

if (Label != NULL) {

free(Label);

}

if (OpcodeMnemonic != NULL) {

free(OpcodeMnemonic);

}

if (Operand != NULL) {

free(Operand);

}

if (TempOperand != NULL) {

free(TempOperand);

}

if (FileName != NULL) {

free(FileName);

}

if (ImmediateFileName != NULL) {

free(ImmediateFileName);

}

if (ListFileName != NULL) {

free(ListFileName);

}

if (ObjectFileName != NULL) {

free(ObjectFileName);

}

return 0;

}