과목 명: 시스템프로그래밍

담당 교수 명: 소 정 민

<<Assignment 3>>

**서강대학교 컴퓨터공학과**

**[학번] 20171665**

**[이름] 이선호**

목 차

1. 프로그램 개요 3

2. 프로그램 설명 4

2.1 프로그램 흐름도 4

3. 모듈 정의 7

3.1 모듈 이름: InitializeESTAB 7

3.1.1 기능

3.1.2 사용 변수

3.2 모듈이름: InsertESTAB 7

3.2.1 기능

3.2.2 사용변수

3.3 모듈이름: DeleteESTAB 8

3.3.1 기능

3.3.2 사용변수

3.4 모듈이름: SearchESTAB 8

3.4.1 기능

3.4.2 사용변수

3.5 모듈 이름: PrintESTAB 8

3.5.1 기능

3.5.2 사용 변수

3.6 모듈 이름: SearchOpcodeByValue 9

3.6.1 기능

3.6.2 사용 변수

3.7 모듈이름: LoadObjectCode 9

3.7.1 기능

3.7.2 사용변수

3.8 모듈이름: LoadPassOne 9

3.8.1 기능

3.8.2 사용변수

3.9 모듈이름: LoadPassTwo 10

3.9.1 기능

3.9.2 사용변수

3.10 모듈이름: RunProgram 11

3.10.1 기능

3.10.2 사용변수

3.11 모듈이름: RunCommand 12

3.11.1 기능

3.11.2 사용변수

3.12 모듈이름: ExecuteInstruction 12

3.12.1 기능

3.12.2 사용변수

3.13 모듈이름: FindBreakPoint 12

3.13.1 기능

3.13.2 사용변수

3.14 모듈이름: SetBreakPoint 12

3.14.1 기능

3.14.2 사용변수

3.15 모듈이름: PrintBreakPoint 13

3.15.1 기능

3.15.2 사용변수

3.16 모듈이름: ClearAllBreakPoint 13

3.16.1 기능

3.16.2 사용변수

3.17 모듈이름: PrintRegister 13

3.17.1 기능

3.17.2 사용변수

3.18 모듈이름: BreakPointCommand 13

3.18.1 기능

3.18.2 사용변수

4. 전역 변수 정의 14

4.1 typedef struct ExternalSymbol EXTSYMBOL 14

4.2 ProgramStartAddress 14

4.3 ExecutionAddress 14

4.4 ESTAB[20] 14

4.5 BreakPoint[0X100000] 14

4.6 BreakPointSize 14

4.7 BreakPointCheck 14

4.8 BreakPointCheckFlag 14

4.9 DebugFlag 15

5. 코드 15

5.1 20171665.h 15

5.2 20171665.c 18

5.3 AssemblePassOne.c 42

5.4 AssemblePassTwo.c 50

5.5 LoadPassTwo.c 64

5.6 LoadPassTwo.c 72

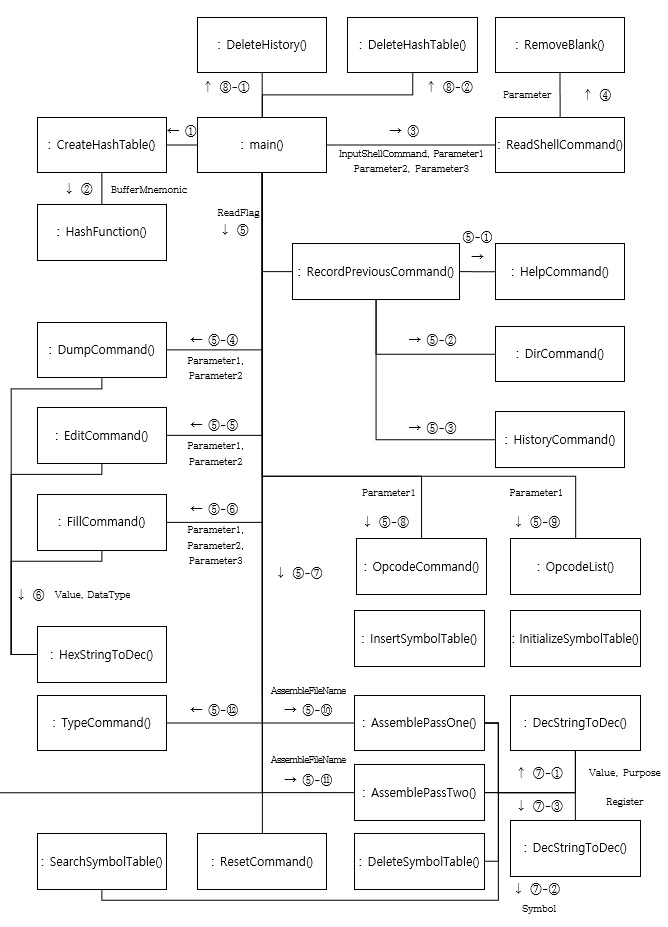
5.7 RunProgram.c 80

# 프로그램 개요

1 ~ 2주차에서 생성한 SIC(Simplified Instruction Computer) machine의 기본적인 기능에서 나아가 한 개 또는 두 개 이상의 object code를 linking하여 가상 메모리의 특정 위치에 load하고 이를 실행하는 프로그램을 제작하는 것을 목표로 했다.

# 프로그램 설명

## 프로그램 흐름도



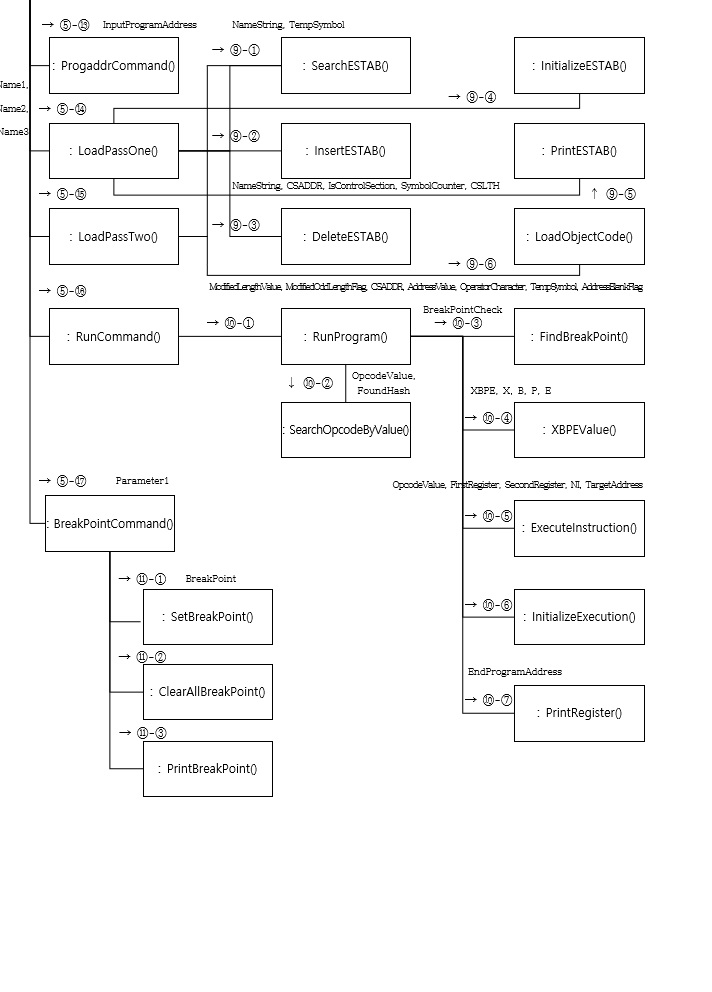


그림 1> 프로그램 흐름도

이 프로그램에서 1 ~ 2주차에서 제작했던 각 sub-routine들의 상호작용에다가 3주차에서 요구하는 기능을 수행하기 위해 loader와 run command에 관한 함수들을 추가하여 프로그램 흐름도를 그렸다.

1 ~ 2주차와 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, ProgaddrCommand, LoadPassOne, LoadPassTwo, RunCommand, BreakPointCommand 함수는 필요한 개수의 Parameter들을 인수로 전달받아서 실행한다.

Object Code를 메모리에 load하고자 Pass 1과 Pass 2로 두 번 읽기 위해서 각각 LoadPassOne과 LoadPassTwo라는 독립적인 모듈을 제작했다. Command 중에서 ‘loader’와 Parameter1을 통해 file name을 입력받으면 LoadPassOne에 의해 해당 name의 파일을 fopen으로 열어서 읽는다. 다수의 object code를 메모리에 load하고자 하면 마찬가지로 Parameter2와 Parameter3를 통해 file name을 입력받아 파일을 읽는다. 만약 열기 과정에서 오류가 있을 경우 error를 출력하고 해당 함수를 종료한다. 각 object code마다 header line(‘H’ line)을 읽어서 object code 이름(control section 명), object code 길이(control section 길이)를 구한 후, object code 이름을 external symbol로 간주하여 ESTAB(External Symbol Table)에 삽입한다. Symbol의 정의가 있는 definition line(‘D’ line)을 읽어서 해당 코드에서 정의된 symbol들을 프로그램 시작 주소에서 상대적인 offset을 더한 만큼의 값과 함께 마찬가지로 ESTAB에 삽입한다. End line(‘E’ line)이 나오면 Pass One을 종료하고 모든 object code에 대해 이러한 시행을 반복한다.

Load에 대한 Pass 1이 정상적으로 error 없이 완료되면 앞에서 생성한 ESTAB을 바탕으로 data 또는 direct addressing 영역에 관해 modification of address를 수행하기 위해LoadPassTwo 함수를 통해 Load에 대한 Pass 2를 수행한다. Header line에서는 다수의 object code를 load할 때 다음 object code에 대한 시작 주소를 구해야 하므로 현재 읽고 있는 code의 길이를 구해 CSLTH에 저장한다. Reference line (‘R’ line)을 읽을 때 두 번째 열부터 0에서 9까지의 숫자 중 하나가 나온다면 해당 코드는 reference table을 사용하여 symbol을 구하는 것이므로 외부 참조 기호들을 reference number와 함께 reference table에 삽입하여 modification line(‘M’ line)에서 주소를 재배치할 때 reference number를 index로 사용하여 symbol을 참조할 수 있도록 한다. Reference table을 사용하지 않는 코드이면 ESTAB을 탐색하여 해당 symbol의 주소를 직접 참조하여 주소 재배치에 반영한다. 이를 위해 8-9번째 열의 값을 ModifiedLengthValue에 저장하고, 해당 값이 홀수인지 아닌지를 저장하는 ModifiedOddLengthFlag를 사용하여 프로그램 시작 주소인 CSADDR에서 해당 offset만큼을 더한 index로 DumpMemory 배열에 접근하여 ModifiedLengthValue를 2로 나눈 값만큼의 byte를 for loop을 돌면서 주소 재배치를 했을 때의 변경된 값인 (원래 메모리에 있던 value) + (Symbol에 해당하는 주소)를 strcpy를 통해 바이트별로 나눠서 DumpMemory의 각 byte에 할당한다. Load가 모두 완료되면 PrintESTAB 함수를 통해 object code의 이름과 길이, 그리고 symbol들의 주소를 모두 출력하고, 모든 코드들의 길이의 합을 total length에 출력한다.

Breakpoint로 인한 종료가 이전에 없었던 상태에서 RunCommand 함수를 실행하면 RunProgram 함수를 통해 PC 레지스터와 L 레지스터 값이 각각 실행 주소(프로그램 시작 주소)와 여기에 프로그램의 전체 길이인 ProgramTotalLength를 더한 값으로 할당된다. PC 레지스터가 전체 메모리의 총 길이를 초과하지 않는 한 while loop을 계속 실행한다. PC 레지스터를 하나씩 증가시키면서 OpcodeValue(Opcode또는 Opcode와 NI Flag가 저장된 영역)을 읽는데, Opcode의 형식을 알아야 앞으로 몇 byte를 더 읽을지 알 수 있으므로 SearchOpcodeByValue 함수를 통해 해당 opcode가 어떤 형식인지를 찾는다. Opcode가 1형식이면 바로 ExecuteInstruction 함수를 실행하고, 2형식이면 PC를 1만큼 증가시키고 해당 index의 DumpMemory에 있는 byte를 읽어서 첫 번째와 두 번째 레지스터 값을 알아낸 후에 이를 ExecuteInstruction 함수의 parameter로 넘겨서 이를 실행한다. 형식이 3 또는4이면 XBPE Flag와 displacement(또는 address 영역)의 첫 번째 byte를 읽어서 각각 X, B, P, E의 Flag 값을 지정한다. E Flag가 1로 설정이 되면 PC를 1 증가시킨만큼 1 byte를 더 읽어서 이를 target address에 반영한다. X Flag가 1로 설정이 되면 target address에 X 레지스터 값을 더한다. B가 0이고 P가 1이면 PC relative addressing이므로 target address에 PC 레지스터 값을 더한다. B가 1이고 P가 0이면 Base relative addressing이므로 target address에 Base 레지스터 값을 더한다. ExecuteInstruction 함수에서는 opcode의 값을 바탕으로 실제로 어떠한 instruction을 실행할지를 switch문을 통해 수행한다. PC Register가 프로그램 시작 주소에서 프로그램 총 길이를 더한 값과 같게 되면 while loop을 벗어난다.

# 모듈 정의

## 모듈 이름 : InitializeESTAB()

### 기능

External Symbol Table을 모두 빈 상태로 초기화한다.

### 사용 변수

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

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

## 모듈 이름: InsertESTAB(char \*Name, int LocationCounter, int ControlSectionOrNot, int OrderNumber, int ControlSectionLength)

### 기능

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

### 사용 변수

Buffer – HashFunction을 통해 구한 해당 Name에 대응되는 External Symbol의 Hash값을 갖는다.

NewSymbol – Linked List의 맨 뒤쪽에 삽입하고자 하는 External Symbol Node이다.

OrderNumber – 해당 External Symbol이 주소 오름차순으로 고려했을 때 몇 번째로 오는 symbol인지에 관한 값을 갖는다.

ControlSectionOrNot – 해당 External Symbol이 ControlSection의 이름이 해당하는지 아닌지에 관한 값을 갖는다.

ControlSectionLength – 해당 control section의 길이이다.

## 모듈이름: DeleteESTAB()

### 기능

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

### 사용변수

Index – 20개의 기본적인 원소를 가지고 있는 External Symbol Table의 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: SearchESTAB(char \*Name, EXTSYMBOL \*\*TempPointer)

### 기능

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

### 사용변수

HashCode - HashFunction으로 찾은 name의 Hash값을 의미한다.

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

## 모듈이름: PrintESTAB()

### 기능

External Symbol Table에 저장된 symbol들을 주소의 오름차순으로 창에 출력하는 역할을 한다.

### 사용변수

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

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

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

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

SortedSymbolList – External Symbol을 주소의 오름차순으로 정렬한 list를 저장한다.

TempSymbolName – External Symbol의 이름 배열의 첫 원소의 주소를 갖는다.

TempValue – External Symbol을 주소의 오름차순으로 정렬할 때 Symbol의 각 데이터를 sort하는 과정에서 데이터 값을 임시로 저장하기 위한 용도로 사용된다.

TotalLength – 메모리에 load한 프로그램의 총 길이 값을 갖는다.

## 모듈이름: SearchOpcodeByValue(int OpcodeValue, Hash\*\* TempPointer)

### 기능

Integer data type의 opcode에 대응되는 값을 통해 opcode의 format을 구하기 위해 사용하는 함수이다.

### 사용변수

Counter - 20개의 기본적인 원소를 가지고 있는 External Symbol Table의 각 배열의 첫 번째 원소를 loop를 돌며 접근하기 위해 사용한다.

## 모듈이름: LoadObjectCode(int ModifiedLengthValue, int ModifiedOddLengthFlag, int CSADDR, int AddressValue, char OperatorCharacter, EXTSYMBOL \*TempSymbol, int AddressBlankFlag)

### 기능

LoadPassTwo 함수에서 object code를 CSADDR로 시작하는 위치에 load한 메모리에서 modification line에서 수정이 되어야 할 메모리 byte를 실제로 수정하는 작업을 수행한다.

### 사용변수

ModifiedMemory – load된 메모리에서 주소 재배치 또는 값 변경을 목적으로 필요한 byte의 수정을 위해 수정되어야할 byte 값을 임시로 저장하기 위한 용도로 사용한다.

TempHexDigit – 메모리에서 수정해야할 byte가 홀수 개이면 실제 메모리에서 수정해야 할 홀수 개의 byte의 맨 앞의 first half byte는 수정을 해야하지 않아야 하므로 앞의 half byte를 임시로 따로 저장한 후 뒤의 half byte들의 값을 변경하기 위해 사용한다.

TempFirstByte – TempHexDigit의 값을 구하기 전에 실제로 수정하고자 하는 메모리 영역의 첫 byte에서 first half byte를 sprintf로 분리하기 위해 사용하는 character array이다.

ModifiedValue – ModifiedMemory에 저장된 hexadecimal string에 대응되는 integer data type의 decimal 값을 갖는다.

TempModifiedValue – 수정하고자 하는 메모리 영역이 음수 값을 갖는지를 bitwise and operator를 사용하여 구하기 위해 첫 번째 byte의 값을 임시로 갖는다.

## 모듈이름: LoadPassOne(char \*FileName1, char \*FileName2, char \*FileName3)

### 기능

Object Code를 메모리에 load하기 위해서는 현재 정의된 기호와 외부 참조 기호에 대한 데이터의 주소를 미리 구하여 symbol에 대응되는 주소를 참조하고자 두 번의 읽기가 필요한데, 그 중 첫 번째 읽기인 Pass 1을 수행하는 함수이다. 여기서는 ‘D’ line에 작성된 현재 code에서 정의된 symbol을 External Symbol Table에 저장함으로써 다음 Pass 2에서 다른 object code에서 reference 하는 external symbol의 주소를 찾을 수 있게끔 한다.

### 사용변수

FilePointer1 – 변환하고자 하는 첫 번째 파일의 File Pointer 변수이다.

FilePointer2 – 변환하고자 하는 두 번째 파일의 File Pointer 변수이다.

FilePointer3 – 변환하고자 하는 세 번째 파일의 File Pointer 변수이다.

FileName1 –변환하고자 하는 첫 번째 파일의 이름이다.

FileName2 –변환하고자 하는 두 번째 파일의 이름이다.

FileName3 –변환하고자 하는 세 번째 파일의 이름이다.

TempFilePointer – 다수의 object code를 메모리에 load할 때 loop를 돌면서 순서대로 object code를 load해야 하므로 file pointer가 각각의 object code가 저장된 파일을 가리키도록 하기 위해 사용한다.

LoadedFileNumber – 메모리에 load할 object code의 개수이다.

CSADDR – control section 또는 object code program의 시작 주소이다.

CSLTH – control section 또는 object code program의 길이이다.

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

EndFlag – object code에서 End line(‘E’ line)을 만난 후에 파일의 끝문자를 만나는지를 확인하기 위해 사용한다. 1이면 ‘E’ line을 만난 것이고, 0이면 만나지 않은 것이다.

LoopCounter – 메모리에 한 object code씩 loop를 돌며 load할 때 필요한 counter이다.

SymbolCounter – 주소의 오름차순으로 정렬하여 각 object에서 정의된 symbol들을 ESTAB에 저장하기 위한 번호로써 사용한다.

ReadString – object code를 한줄씩 읽으면서 각 코드의 데이터들을 분리하기 위해 사용한다. Object code의 파일에서 읽은 줄들을 한 줄씩 string으로 이 character array에 저장한다.

ReadLine – ReadString array의 첫 번째 원소의 주소를 가리킨다.

NameString – 현재 object code에서 정의된 참조 기호의 이름을 character array로 저장한다.

AddressString – object code에서 정의된 참조 기호가 프로그램이 시작하는 주소에서 얼만큼 떨어져 있는지의 offset을 ReadString에서 읽어서 이를 character array로 저장한다.

TempSymbol – SearchESTAB 함수를 통해 찾고자 하는 external symbol의 주소를 구하기 위한 EXTSYMBOL pointer type의 변수이다.

## 모듈이름: LoadPassTwo(char \*FileName1, char \*FileName2, char \*FileName3)

### 기능

LoadPassOne에서 생성한 ESTAB의 데이터를 바탕으로 reference line에 있는 외부 참조 기호의 이름을 참조하여 각 external symbol의 실제 주소를 구하고, object code를 CSADDR 위치에서부터 시작하여 메모리에 load하는 작업을 수행한다.

### 사용변수

FilePointer1 – 변환하고자 하는 첫 번째 파일의 File Pointer 변수이다.

FilePointer2 – 변환하고자 하는 두 번째 파일의 File Pointer 변수이다.

FilePointer3 – 변환하고자 하는 세 번째 파일의 File Pointer 변수이다.

FileName1 –변환하고자 하는 첫 번째 파일의 이름이다.

FileName2 –변환하고자 하는 두 번째 파일의 이름이다.

FileName3 –변환하고자 하는 세 번째 파일의 이름이다.

TempFilePointer – 다수의 object code를 메모리에 load할 때 loop를 돌면서 순서대로 object code를 load해야 하므로 file pointer가 각각의 object code가 저장된 파일을 가리키도록 하기 위해 사용한다.

LoadedFileNumber – 메모리에 load할 object code의 개수이다.

CSADDR – control section 또는 object code program의 시작 주소이다.

CSLTH – control section 또는 object code program의 길이이다.

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

EndFlag – object code에서 End line(‘E’ line)을 만난 후에 파일의 끝문자를 만나는지를 확인하기 위해 사용한다. 1이면 ‘E’ line을 만난 것이고, 0이면 만나지 않은 것이다.

LoopCounter – 메모리에 한 object code씩 loop를 돌며 load할 때 필요한 counter이다.

SymbolCounter – 주소의 오름차순으로 정렬하여 각 object에서 정의된 symbol들을 ESTAB에 저장하기 위한 번호로써 사용한다.

ReadString – object code를 한줄씩 읽으면서 각 코드의 데이터들을 분리하기 위해 사용한다. Object code의 파일에서 읽은 줄들을 한 줄씩 string으로 이 character array에 저장한다.

ReadLine – ReadString array의 첫 번째 원소의 주소를 가리킨다.

NameString – 현재 object code에서 정의된 참조 기호의 이름을 character array로 저장한다.

AddressString – object code에서 정의된 참조 기호가 프로그램이 시작하는 주소에서 얼만큼 떨어져 있는지의 offset을 ReadString에서 읽어서 이를 character array로 저장한다.

TempSymbol – SearchESTAB 함수를 통해 찾고자 하는 external symbol의 주소를 구하기 위한 EXTSYMBOL pointer type의 변수이다.

## 모듈이름: RunProgram

### 기능

메모리에 load된 프로그램을 실제로 실행하기 위한 함수이다.

### 사용변수

FirstRegister – instruction의 operand에서 첫 번째 레지스터에 대응되는 번호 값을 갖는다.

SecondRegister – instruction의 operand에서 첫 번째 레지스터에 대응되는 번호 값을 갖는다.

TargetAddress – instruction에서 3 format일 때의 displacement 또는 4 format일 때의 address 영역의 값을 갖는다.

OpcodeValue - instruction에서 opcode의 값을 갖으며, 해당 opcode의 값을 통해 이에 대응되는 실제 opcode의 format을 구하기 위해 사용한다.

N, I, X, B, P, E – instruction 형식에서 각각 indirect addressing flag, immediate addressing flag, indexed addressing flag, base relative addressing flag, PC relative addresssing flag, 그리고 extension addressing flag를 나타낸다.

NI – Opcode의 값과 N, I flag의 값을 따로 알기 위해서는 instruction의 첫 번째 byte에서 이를 분리해야 하므로 첫 번째 byte를 4로 나눈 나머지의 값을 갖는다.

TempValue – Format 3 또는 4인 instruction code에서 X, B, P, E flag와 displacement 또는 address의 첫 번째 half byte영역을 합쳐 1 byte를 임시로 저장하기 위해 사용한다.

FirstAddressByte – displacement 또는 address의 첫 번째 half byte를 TempValue로부터 분리한 값을 갖는다.

XBPE - X, B, P, E flag를 TempValue로부터 분리한 값을 갖는다.

## 모듈이름: RunCommand()

### 기능

메모리에 load된 프로그램을 RunProgram 함수를 통해 실제로 실행한 후, 멈춘 지점에 대한 상태를 알려주는 BreakPointCheckFlag에 따라 레지스터를 출력하고 이를 초기화할지를 결정하는 작업을 수행한다.

### 사용변수

Result – RunProgram 함수를 통해 실행한 결과 값을 저장한다. 0 이하의 값을 가지면 프로그램이 끝까지 도달했다는 것이고, 그렇지 않으면 BreakPointCheckFlag 값에 따라 break point까지 도달했거나 정상적인 instruction이 실행되지 않아서 runtime error가 발생했다는 것이다.

## 모듈이름: ExecuteInstruction(int OpcodeValue, int FirstRegister, int SecondRegister, int NI, int TargetAddress)

### 기능

Instruction 실행에 필요한 parameter를 넘겨 받아서 실제로 opcode 값에 따라 대응되는 operator를 실행하는 기능을 수행한다.

### 사용변수

OperandValue – Indirect 또는 Immediate Addressing에 따라 instruction의 operand 영역의 string을 decimal 값으로 변환하여 이를 저장한다.

TempValue – Instruction을 수행할 때 임시로 레지스터에 저장된 값을 할당받아서 메모리에 byte별로 분리하여 load하기 위해 사용한다.

## 모듈이름: FindBreakPoint(int Address)

### 기능

프로그램이 RunProgram 함수를 통해 실행될 때 현재 실행 도중의 Address 값을 parameter로 넘겨 받아서 이와 대응되는 break point를 찾는다. 만약 해당 주소를 갖는 break point가 존재하면 1을 반환하고, 그렇지 않으면 0을 반환한다.

### 사용변수

i – BreakPointSize만큼 BreakPoint 배열의 loop를 돌면서 address에 대응되는 break point를 찾기 위한 counter 변수이다.

## 모듈이름: SetBreakPoint(int Address)

### 기능

Parameter로 넘겨받은 address를 값으로 갖는 break point가 BreakPoint array에 존재하는지 FindBreakPoint 함수를 통해 확인하고, 만약 있으면 아무런 작업을 수행하지 않는다. 그렇지 않으면 BreakPointSize를 1만큼 늘리고 해당 index의 BreakPoint Array의 element에 Address 값을 저장한다.

### 사용변수

전역 변수 외의 사용하는 local 변수는 존재하지 않는다.

## 모듈이름: PrintBreakPoint()

### 기능

현재 BreakPoint array에 저장된 break point의 주소를 모두 출력한다.

### 사용변수

i – BreakPointSize만큼 BreakPoint 배열의 loop를 돌면서 BreakPoint Array에 저장된 모든 break point를 출력하기 위한 counter 변수이다.

## 모듈이름: ClearAllBreakPoint()

### 기능

현재 BreakPoint array에 저장된 break point의 주소를 모두 0으로 초기화한다.

### 사용변수

i – BreakPointSize만큼 BreakPoint 배열의 loop를 돌면서 BreakPoint Array에 저장된 모든 break point를 지우기 위한 counter 변수이다.

## 모듈이름: PrintRegister()

### 기능

프로그램을 실행하다가 break point가 걸렸을 때 해당 주소에서 A, X, L, PC, B, S, T 레지스터의 값을 모두 출력하기 위해 사용한다.

### 사용변수

전역 변수 외의 사용하는 local 변수는 존재하지 않는다.

## 모듈이름: BreakPointCommand(char \*Parameter1)

### 기능

사용자로부터 break point로 설정하고자 하는 address를 입력받아서 이 string을 대응되는 decimal value로 변환한 후 SetBreakPoint 함수를 통해 break point를 삽입한다.

### 사용변수

BreakPoint – 사용자로부터 입력받은 hexadecimal address를 갖는 string을 decimal로 변환한 값을 할당받는다. SetBreakPoint 함수를 호출할 때 parameter로 쓰인다.

# 전역 변수 정의

## typedef struct ExternalSymbol{ char \*SymbolName, int SymbolAddress, int ControlSectionLength, int OrderNumber, int ControlSectionOrNot, struct ExternalSymbol \*Next} EXTSYMBOL

각 Object Code의 ‘H’ line에서 읽은 데이터에 존재하는 정의된 기호들을 갖는 external symbol table을 구성할 때 table 각각의 node의 데이터를 저장하기 위해 사용한다. EXTSYMBOL은 symbol의 이름에 해당하는 SymbolName, symbol이 존재하는 주소인 SymbolAddress, symbol이 control section의 이름이면 ControlSectionLength, 주소의 오름차 순으로 몇 번째 symbol에 해당되는지 번호가 저장된 OrderNumber, Control Section 이름에 해당하는지 그렇지 않은지에 관한 Flag를 저장하는 ControlSectionOrNot,그리고 다음 node를 linking하기 위한 자기 자신을 data type으로 갖는 포인터 변수를 멤버 변수로 갖는다.

## int ProgramStartAddress

프로그램이 처음으로 메모리에 load될 위치에 관한 주소를 갖는다.

## int ExecutionAddress

프로그램을 실행했을 때 시작하는 주소를 갖는다.

## EXTSYMBOL ESTAB[20]

Load에 관한 Pass 1에서 주소와 이름을 저장하는 크기가 20인 External Symbol Table이다.

## int BreakPoint[0X100000]

bp command를 통해 설정한 break point의 주소를 저장하는 array이다.

## int BreakPointSize

BreakPoint array에 저장된 유효한 break point의 개수를 의미한다.

## int BreakPointCheck

bp command를 통해 설정한 break point에서 실행이 멈추었을 때, 이를 다시 재개하기 위해 현재 break point주소와 PC가 같은지를 판단하고자 사용하는 변수이다.

## int BreakPointCheckFlag

프로그램이 끝까지 도달하여 멈춘 것인지, 아니면 bp command를 통해 설정한 break point에 의해 멈춘 것인지에 관한 flag 값을 갖는 변수이다. 1이면 break point에 의해 멈춘 것이고, 그렇지 않으면 프로그램이 끝까지 도달했거나 run 도중 오류가 발생한 것이다.

## int DebugFlag

프로그램을 처음 실행했을 때 PC와 L 레지스터의 값을 각각 프로그램의 시작 주소와 프로그램 시작 주소 + 프로그램의 총 길이로 설정하기 위해 필요한 변수이다. 이는 break point를 통해 중간에 실행이 멈추었을 때 다시 재개하는 경우 PC와 L 레지스터 값을 이로 변경하는 것을 방지하기 위해 break point에 의해 멈추지 않은 상태에서만 PC와 L 레지스터 값을 변경하기 위한 목적이다.

# 코드

## 20171665.h

#ifndef \_\_HEADER\_\_

#define \_\_HEADER\_\_

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.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>

#include <math.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 Progaddr 14

#define Loader 15

#define Run 16

#define BP 17

#define Word 100

#define Character 101

#define Integer 102

#define Hexadecimal 103

#define ForHexAddress 104

#define ForProgramCounter 105

#define IndirectAddressing 200

#define ImmediateAddressing 201

#define IsControlSection 300

#define IsSymbol 301

#define MaxAddress 0X100000

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;

typedef struct ExternalSymbol {

char \*SymbolName;

int SymbolAddress;

int ControlSectionLength;

int OrderNumber;

int ControlSectionOrNot;

struct ExternalSymbol\* Next;

}EXTSYMBOL;

extern NODE \*FirstNode;

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

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

extern EXTSYMBOL ESTAB[20]; // External Symbol Table 선언

extern char DumpMemory[MaxAddress];

extern int LastAddress;

extern int StartAddress;

extern int ProgramLength;

extern int ProgramStartAddress;

extern int ExecutionAddress;

extern int ProgramTotalLength;

extern int Register[10];

extern int DebugFlag;

extern int BreakPoint[0X100000];

extern int BreakPointSize;

extern int BreakPointCheck;

extern int BreakPointCheckFlag;

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

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

extern int HexCharToDec(char HexCharacter);

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 ProgaddrCommand(char\* InputProgramAddress);

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 SearchOpcodeByValue(int OpcodeValue, 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 void TrimRightBlank(char \*String);

extern int InitializeESTAB();

extern int InsertESTAB(char \*SymbolName, int LocationCounter, int ControlSectionOrNot, int OrderNumber, int ControlSectionLength);

extern void DeleteESTAB();

extern int SearchESTAB(char \*SymbolName, EXTSYMBOL \*\*TempPointer);

extern int PrintESTAB();

extern int LoadObjectCode(int ModifiedLengthValue, int ModifiedOddLengthFlag, int CSADDR, int AddressValue, char OperatorCharacter, EXTSYMBOL \*TempSymbol, int AddressBlankFlag);

extern int FindRegisterNumber(char \*Register);

extern int AssemblePassOne(char \*AssembleFileName);

extern int AssemblePassTwo(char \*AssembleFileName);

extern int LoadPassOne(char \*FileName1, char \*FileName2, char \*FileName3);

extern int LoadPassTwo(char \*FileName1, char \*FileName2, char \*FileName3);

extern int RunProgram();

extern int RunCommand();

extern int BreakPointCommand(char \*Parameter1);

extern int FindBreakPoint(int Address);

extern void SetBreakPoint(int Address);

extern void PrintBreakPoint();

extern void ClearAllBreakPoint();

extern void PrintRegister(int EndProgramAddress);

extern void ExecuteInstruction(int OpcodeValue, int FirstRegister, int SecondRegister, int NI, int TargetAddress);

extern void XBPEValue(int XBPE, int \*X, int \*B, int \*P, int \*E);

extern void InitializeExecution();

#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;

int ProgramStartAddress = 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 : \*/

/\*목적 : 전달받은 Parameter에서 16진수로 된 character를 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 HexCharToDec(char HexCharacter) {

if (HexCharacter >= '0' && HexCharacter <= '9')

return HexCharacter - '0';

if (HexCharacter >= 'A' && HexCharacter <= 'F')

return HexCharacter - 'A' + 10;

if (HexCharacter >= 'a' && HexCharacter <= 'f')

return HexCharacter - 'a' + 10;

return -1;

}

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

/\*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 : DecStringToDec(char \*Parameter, short Purpose)\*/

/\*목적 : Decimal string을 10진수 값으로 바꾼다.\*/

/\*리턴값 : 10진수로 바꾼 값\*/

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

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 : ProgaddrCommand(char\* InputProgramAddress)\*/

/\*목적 : 프로그램 시작 주소를 설정하고 이를 출력한다.\*/

/\*리턴값 : 성공적으로 설정하고 출력하면 0을 반환\*/

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

int ProgaddrCommand(char\* InputProgramAddress) {

if (InputProgramAddress == NULL) {

printf("You should write the program address.\n");

return -1;

}

ProgramStartAddress = HexStringToDec(InputProgramAddress, ForHexAddress);

printf("Program starting address set to 0x%X\n", ProgramStartAddress);

return 0;

}

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

/\*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, LoaderFlag = 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 (!LoaderFlag) {

if (CommandCounter && Flag == 0) {

if (!strcmp(BufferCommand, "loader")) {

LoaderFlag = 1;

}

Flag = 1;

}

}

else {

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 ((!ParameterCounter1 && Flag == 1) || (Flag == 0)) {

Invalid = 1;

}

}

BeforeChar = BufferChar;

continue;

}

else if (!LoaderFlag) {

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;

BufferCommand[CommandCounter] = '\0';

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);

//printf("%s [%s][%s][%s]\n", BufferCommand, \*Parameter1, \*Parameter2, \*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;

}

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

ResultValue = Run;

}

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

ResultValue = BP;

}

}

//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;

}

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

ResultValue = Progaddr;

}

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

ResultValue = BP;

}

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

ResultValue = Loader;

}

}

}

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");

printf("progaddr address\n");

printf("loader [filename1, filename2, filename3]\n");

printf("run\n");

printf("bp [address|clear]\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));

DebugFlag = 0;

ProgramTotalLength = 0;

BreakPointCheck = 0;

BreakPointCheckFlag = 0;

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 : SearchOpcodeByValue(int OpcodeValue, HASH\*\* TempPointer)\*/

/\*목적 : Integer data type의 opcode에 대응되는 값을 통해 opcode의 format을 구하기 위해 사용하는 함수이다.\*/

/\*리턴값 : 찾으면 0을, 찾지 못하면 1을 반환한다.\*/

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

int SearchOpcodeByValue(int OpcodeValue, HASH\*\* TempPointer) {

int Counter = 0, FoundFlag = 0;

HASH\* TempHash = NULL;

for (TempHash = &(HashTable[Counter]); Counter < 20; TempHash = &(HashTable[++Counter])) {

while (TempHash != NULL) {

//printf("Counter: %d / Value: %s / %s\n", Counter, TempHash->Number, TempHash->Format);

if (OpcodeValue == HexStringToDec(TempHash->Number, ForHexAddress)) {

FoundFlag = 1;

break;

}

TempHash = TempHash->Next;

}

if (FoundFlag) {

break;

}

}

if (FoundFlag) {

(\*TempPointer) = TempHash;

}

else {

return 1;

}

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;

}

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;

}

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;

}

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

/\*함수 : TypeCommand(char \*FileName)\*/

/\*목적 : type 명령어를 입력하면 파일 내용을 출력한다.\*/

/\*리턴값 : 성공적으로 출력하면 0을 반환\*/

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

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 : FindRegisterNumber(char \*Register)\*/

/\*목적 : Register에 대응되는 번호를 찾는다.\*/

/\*리턴값 : Register 번호\*/

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

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;

case Progaddr:

if (!ProgaddrCommand(Parameter1)) {

RecordPreviousCommand(InputShellCommand);

}

break;

case Loader:

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

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

RecordPreviousCommand(InputShellCommand);

}

}

break;

case Run:

if (!RunCommand()) {

RecordPreviousCommand(InputShellCommand);

}

break;

case BP:

if (!BreakPointCommand(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 : 0이면 성공, -1이면 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;

}

## LoadPassOne.c

#include "20171665.h"

EXTSYMBOL ESTAB[20];

int ProgramTotalLength = 0;

void TrimRightBlank(char \*String) {

char TempString[100], \*StringEnd = NULL;

strcpy(TempString, String);

StringEnd = TempString + strlen(TempString) - 1;

while (StringEnd != TempString && (\*StringEnd == ' ' || \*StringEnd == '\0' || \*StringEnd == '\n')) {

StringEnd--;

}

\*(StringEnd + 1) = '\0';

strcpy(String, TempString);

return;

}

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

/\*함수 : InitializeESTAB()\*/

/\*목적 : External Symbol Table을 모두 빈 상태로 초기화한다.\*/

/\*리턴값 : 정상적으로 수행되면 0을 반환한다.\*/

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

int InitializeESTAB() {

int Index;

char \*LabelNamePointer = NULL;

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

if ((LabelNamePointer = ESTAB[Index].SymbolName) != NULL) {

free(LabelNamePointer);

}

LabelNamePointer = NULL;

ESTAB[Index].SymbolAddress = 0;

ESTAB[Index].ControlSectionLength = 0;

ESTAB[Index].OrderNumber = 0;

ESTAB[Index].ControlSectionOrNot = 0;

ESTAB[Index].Next = NULL;

}

return 0;

}

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

/\*함수 : InsertESTAB(char \*Name, int LocationCounter, int ControlSectionOrNot, int OrderNumber, int ControlSectionLength)\*/

/\*목적 : HashFunction을 사용하여 External Symbol에 대응되는 Hash 값을 구하여 해당 번째 linked list 뒤에 삽입하는 기능을 수행한다.\*/

/\*리턴값 : 정상적으로 수행되면 0을 반환한다.\*/

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

int InsertESTAB(char \*Name, int LocationCounter, int ControlSectionOrNot, int OrderNumber, int ControlSectionLength) {

int Buffer;

// ESTAB의 해당 hash값을 갖는 index의 Name이 비어 있으면

if (ESTAB[(Buffer = HashFunction(Name))].SymbolName == NULL) {

ESTAB[Buffer].SymbolName = (char\*)malloc(sizeof(char) \* (int)strlen(Name) + 1);

strcpy(ESTAB[Buffer].SymbolName, Name);

ESTAB[Buffer].SymbolName[strlen(Name)] = '\0';

if (ControlSectionOrNot == IsControlSection) {

ESTAB[Buffer].ControlSectionOrNot = IsControlSection;

}

else if (ControlSectionOrNot == IsSymbol) {

ESTAB[Buffer].ControlSectionOrNot = IsSymbol;

}

ESTAB[Buffer].SymbolAddress = LocationCounter;

ESTAB[Buffer].OrderNumber = OrderNumber;

ESTAB[Buffer].ControlSectionLength = ControlSectionLength;

ESTAB[Buffer].Next = NULL;

}

else {

EXTSYMBOL \*NewSymbol = NULL, \*TempSymbol = &(ESTAB[Buffer]);

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

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

strcpy(NewSymbol->SymbolName, Name);

NewSymbol->SymbolName[strlen(Name)] = '\0';

if (ControlSectionOrNot == IsControlSection) {

NewSymbol->ControlSectionOrNot = IsControlSection;

}

else if (ControlSectionOrNot == IsSymbol) {

NewSymbol->ControlSectionOrNot = IsSymbol;

}

NewSymbol->SymbolAddress = LocationCounter;

NewSymbol->OrderNumber = OrderNumber;

NewSymbol->ControlSectionLength = ControlSectionLength;

NewSymbol->Next = NULL;

while (TempSymbol->Next != NULL) {

TempSymbol = TempSymbol->Next;

}

TempSymbol->Next = NewSymbol;

}

return 0;

}

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

/\*함수 : DeleteESTAB()\*/

/\*목적 : External Symbol Table을 사용하기 전에 기존에 저장되어 있던 table의 내용을 지운다.\*/

/\*리턴값 : 없음\*/

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

void DeleteESTAB() {

int Index;

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

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

TempSymbol = &(ESTAB[Index]);

if (TempSymbol->SymbolName != NULL) {

free(TempSymbol->SymbolName);

TempSymbol->SymbolName = NULL;

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

while (TempSymbol != NULL) {

DeleteSymbol = TempSymbol;

TempSymbol = TempSymbol->Next;

if (DeleteSymbol->SymbolName != NULL) {

free(DeleteSymbol->SymbolName);

}

free(DeleteSymbol);

}

}

}

}

return;

}

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

/\*함수 : SearchESTAB(char \*Name, EXTSYMBOL \*\*TempPointer)\*/

/\*목적 : 찾고자 하는 External Symbol Name이 Symbol Table에 존재하는지 HashFunction 함수를 사용하여 hash 값을 구한 후 Symbol 배열의 해당 원소로 접근하여 Linked list를 탐색하는 용도로 쓰인다.\*/

/\*리턴값 : 정상적으로 수행되면 0을 반환한다.\*/

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

int SearchESTAB(char \*Name, EXTSYMBOL \*\*TempPointer) {

int HashCode;

EXTSYMBOL\* TempSymbol = NULL;

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

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

return -1;

}

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

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

return -1;

}

if (TempSymbol->SymbolName != NULL) {

while ((strcmp(TempSymbol->SymbolName, Name) != 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 : PrintESTAB()\*/

/\*목적 : External Symbol Table에 저장된 symbol들을 주소의 오름차순으로 창에 출력하는 역할을 한다.\*/

/\*리턴값 : 정상적으로 수행되면 0을 반환한다.\*/

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

int PrintESTAB() {

int Index, SymbolNumber = 0, TempValue = 0, TotalLength = 0;

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

char \*TempSymbolName = NULL;

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

TempSymbol = &(ESTAB[Index]);

if (TempSymbol->SymbolName != NULL) {

while (TempSymbol->Next != NULL) {

SymbolNumber++;

TempSymbol = TempSymbol->Next;

}

SymbolNumber++;

}

}

if (SymbolNumber) {

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

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

SortedSymbolList[i].SymbolName = NULL;

SortedSymbolList[i].SymbolAddress = 0;

SortedSymbolList[i].Next = NULL;

}

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

TempSymbol = &(ESTAB[Index]);

if (TempSymbol->SymbolName != NULL) {

while (TempSymbol->Next != NULL) {

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

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

SortedSymbolList[SymbolNumber].ControlSectionLength = TempSymbol->ControlSectionLength;

SortedSymbolList[SymbolNumber].ControlSectionOrNot = TempSymbol->ControlSectionOrNot;

SortedSymbolList[SymbolNumber].OrderNumber = TempSymbol->OrderNumber;

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

TempSymbol = TempSymbol->Next;

}

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

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

SortedSymbolList[SymbolNumber].ControlSectionLength = TempSymbol->ControlSectionLength;

SortedSymbolList[SymbolNumber].ControlSectionOrNot = TempSymbol->ControlSectionOrNot;

SortedSymbolList[SymbolNumber].OrderNumber = TempSymbol->OrderNumber;

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

}

}

if (SymbolNumber > 1) {

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

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

if (SortedSymbolList[i].OrderNumber > SortedSymbolList[j].OrderNumber) {

TempSymbolName = SortedSymbolList[i].SymbolName;

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

SortedSymbolList[j].SymbolName = TempSymbolName;

TempValue = SortedSymbolList[i].SymbolAddress;

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

SortedSymbolList[j].SymbolAddress = TempValue;

TempValue = SortedSymbolList[i].ControlSectionLength;

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

SortedSymbolList[j].ControlSectionLength = TempValue;

TempValue = SortedSymbolList[i].ControlSectionOrNot;

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

SortedSymbolList[j].ControlSectionOrNot = TempValue;

TempValue = SortedSymbolList[i].OrderNumber;

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

SortedSymbolList[j].OrderNumber = TempValue;

}

}

}

}

printf("control\t\tsymbol\t\taddress\t\tlength\n");

printf("section\t\tname\n");

printf("-------------------------------------------------------\n");

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

if (SortedSymbolList[i].ControlSectionOrNot == IsControlSection) {

printf("%s\t\t\t\t%05X\t\t%05X\n", SortedSymbolList[i].SymbolName, SortedSymbolList[i].SymbolAddress & 0XFFFFF, SortedSymbolList[i].ControlSectionLength & 0XFFFFF);

TotalLength += SortedSymbolList[i].ControlSectionLength;

}

else if (SortedSymbolList[i].ControlSectionOrNot == IsSymbol) {

printf("\t\t%s\t\t%05X\n", SortedSymbolList[i].SymbolName, SortedSymbolList[i].SymbolAddress & 0XFFFFF);

}

}

ProgramTotalLength = TotalLength;

printf("-------------------------------------------------------\n");

printf("total length:\t%05X\n", TotalLength & 0XFFFFF);

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

free(SortedSymbolList[i].SymbolName);

}

free(SortedSymbolList);

}

return 0;

}

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

/\*함수 : LoadPassOne(char \*FileName1, char \*FileName2, char \*FileName3)\*/

/\*목적 : Object Code를 메모리에 load하기 위해서는 현재 정의된 기호와 외부 참조 기호에 대한 데이터의 주소를 미리 구하여 symbol에 대응되는 주소를 참조하고자 두 번의 읽기가 필요한데, 그 중 첫 번째 읽기인 Pass 1을 수행하는 함수이다. \*/

/\*리턴값 : 정상적으로 수행되면 0을 반환한다.\*/

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

int LoadPassOne(char \*FileName1, char \*FileName2, char \*FileName3) {

FILE \*FilePointer1 = NULL, \*FilePointer2 = NULL, \*FilePointer3 = NULL, \*TempFilePointer = NULL;

int LoadedFileNumber = 0, CSADDR = 0, CSLTH = 0, LoopCounter = 0, SymbolCounter = 0;

int ErrorFlag = 0, EndFlag = 0;

if (FileName1 != NULL) {

if ((FilePointer1 = fopen(FileName1, "r")) != NULL) {

LoadedFileNumber += 1;

}

else {

printf("Please write the name of the first valid file that can be read.\n");

ErrorFlag = 1;

}

if (FileName2 != NULL) {

if ((FilePointer2 = fopen(FileName2, "r")) != NULL) {

LoadedFileNumber += 1;

}

else {

printf("Please write the name of the second valid file that can be read.\n");

ErrorFlag = 1;

}

if (FileName3 != NULL) {

if ((FilePointer3 = fopen(FileName3, "r")) != NULL) {

LoadedFileNumber += 1;

}

else {

printf("Please write the name of the third valid file that can be read.\n");

ErrorFlag = 1;

}

}

}

}

if (ErrorFlag == 1) {

printf("An error occurs. Please check the error.\n");

return 1;

}

DeleteESTAB();

InitializeESTAB();

CSADDR = ProgramStartAddress;

while ((LoopCounter < LoadedFileNumber) && !ErrorFlag) {

switch (LoopCounter) {

case 0:

TempFilePointer = FilePointer1; break;

case 1:

TempFilePointer = FilePointer2; break;

case 2:

TempFilePointer = FilePointer3; break;

default:

printf("The number of loaded file is over the limit.\n");

ErrorFlag = 1;

break;

}

char ReadString[100], \*ReadLine = NULL, NameString[7], AddressString[7];

EXTSYMBOL \*TempSymbol = NULL;

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

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

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

ReadLine = fgets(ReadString, 20, TempFilePointer);

//printf("ReadString: %s / ReadLine: %s\n", ReadString, ReadLine);

if (ReadLine[0] == 'H') {

CSLTH = HexStringToDec(ReadLine + 13, ForHexAddress);

strncpy(NameString, ReadLine + 1, 6);

NameString[6] = '\0';

TrimRightBlank(NameString);

strncpy(AddressString, ReadLine + 7, 6);

AddressString[6] = '\0';

TrimRightBlank(AddressString);

if (!SearchESTAB(NameString, &TempSymbol)) {

ErrorFlag = 1;

break;

}

else {

CSADDR += HexStringToDec(AddressString, ForHexAddress);

if (CSADDR + CSLTH > 0XFFFFF){

ErrorFlag = 1;

break;

}

InsertESTAB(NameString, CSADDR, IsControlSection, (++SymbolCounter), CSLTH);

}

}

else {

ErrorFlag = 1;

break;

}

while (!feof(TempFilePointer)) {

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

ReadLine = fgets(ReadString, 74, TempFilePointer);

//printf("ReadString: %s / ReadLine: %s\n", ReadString, ReadLine);

if (ReadLine[0] == 'E') {

EndFlag = 1;

break;

}

else if (ReadLine[0] == 'D') {

int TempIndex = 0, AddressValue;

for (int i = 0; i < 6; i++) {

TempIndex = i \* 12 + 1;

if (\*(ReadLine + TempIndex) != '\0') {

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

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

strncpy(NameString, ReadLine + TempIndex, 6);

NameString[6] = '\0';

TrimRightBlank(NameString);

if (\*(ReadLine + TempIndex + 6) != '\0') {

strncpy(AddressString, ReadLine + TempIndex + 6, 6);

AddressString[6] = '\0';

TrimRightBlank(AddressString);

//printf("NameString: %s", NameString);

//printf("AddressString: %s\n", AddressString);

AddressValue = HexStringToDec(AddressString, ForHexAddress);

if (!SearchESTAB(NameString, &TempSymbol)) {

ErrorFlag = 1;

break;

}

else {

AddressValue += CSADDR;

InsertESTAB(NameString, AddressValue, IsSymbol, (++SymbolCounter), 0);

}

}

}

}

}

}

if (EndFlag == 0) {

ErrorFlag = 1;

break;

}

CSADDR += CSLTH;

LoopCounter++;

}

PrintESTAB();

if (ErrorFlag) {

printf("An error occurs. Please check the error.\n");

ResetCommand();

return 1;

}

return 0;

}

## LoadPassTwo.c

#include "20171665.h"

int ExecutionAddress = 0;

int LoadObjectCode(int ModifiedLengthValue, int ModifiedOddLengthFlag, int CSADDR, int AddressValue, char OperatorCharacter, EXTSYMBOL \*TempSymbol, int AddressBlankFlag) {

char \*ModifiedMemory = NULL, TempHexDigit, TempFirstByte[3] = { '\0' };

int ModifiedValue = 0, TempModifiedValue = 0;

ModifiedMemory = (char\*)malloc(sizeof(char) \* ((ModifiedLengthValue + 1) / 2) \* 2 + 1);

memset(ModifiedMemory, '\0', sizeof(char) \* ((ModifiedLengthValue + 1) / 2) \* 2 + 1);

for (int i = 0; i < (ModifiedLengthValue + 1) / 2; i++) {

// displacement에서 수정해야 할 코드 길이가 홀수이고 첫 번째 halfbyte이면

if (ModifiedOddLengthFlag && !i) {

sprintf(TempFirstByte, "%02X", DumpMemory[CSADDR + AddressValue + i] & 0XF0);

//printf("TempFirstByte: %s\n", TempFirstByte);

TempHexDigit = TempFirstByte[0];

//printf("TempHexDigit: %c\n", TempHexDigit);

sprintf(ModifiedMemory, "0%01X", DumpMemory[CSADDR + AddressValue + i] & 0X0F);

}

else {

sprintf(ModifiedMemory, "%s%02X", ModifiedMemory, DumpMemory[CSADDR + AddressValue + i] & 0XFF);

}

}

ModifiedMemory[(ModifiedLengthValue + 1) / 2 \* 2] = '\0';

if (ModifiedOddLengthFlag) {

TempModifiedValue = HexCharToDec(ModifiedMemory[1]);

// displacement가 음수일 때

if ((TempModifiedValue & 0X8) == 0X8) {

sprintf(ModifiedMemory, "0%01X%s", TempModifiedValue & 0X7, ModifiedMemory + 2);

ModifiedValue = (-1)\*(0X8 \* (int)pow(16, ModifiedLengthValue - 1)) + HexStringToDec(ModifiedMemory, ForHexAddress);

}

else {

ModifiedValue = HexStringToDec(ModifiedMemory, ForHexAddress);

}

}

else {

TempModifiedValue = HexCharToDec(ModifiedMemory[0]);

// value가 음수일 때

if ((TempModifiedValue & 0X8) == 0X8) {

sprintf(ModifiedMemory, "%01X%s", TempModifiedValue & 0X7, ModifiedMemory + 1);

ModifiedValue = (-1)\*(0X8 \* (int)pow(16, ModifiedLengthValue - 1)) + HexStringToDec(ModifiedMemory, ForHexAddress);

}

else {

ModifiedValue = HexStringToDec(ModifiedMemory, ForHexAddress);

}

}

//printf("1: %s / %d / %04X\n", ModifiedMemory, ModifiedValue, TempSymbol->SymbolAddress & 0XFFFF);

if (!AddressBlankFlag) {

if (OperatorCharacter == '+') {

ModifiedValue += (TempSymbol->SymbolAddress & 0XFFFFF);

}

else if (OperatorCharacter == '-') {

ModifiedValue -= (TempSymbol->SymbolAddress & 0XFFFFF);

}

}

else {

if (OperatorCharacter == '+') {

ModifiedValue += (CSADDR & 0XFFFFF);

}

else if (OperatorCharacter == '-') {

ModifiedValue -= (CSADDR & 0XFFFFF);

}

else {

ModifiedValue += (CSADDR & 0XFFFFF);

}

}

//printf("2: %s / %d / %04X\n", ModifiedMemory, ModifiedValue, TempSymbol->SymbolAddress & 0XFFFF);

memset(ModifiedMemory, '\0', sizeof(char) \* ((ModifiedLengthValue + 1) / 2) \* 2 + 1);

sprintf(ModifiedMemory, "%06X", ModifiedValue & 0XFFFFFF);

//printf("[%s]\n", ModifiedMemory);

for (int i = 0; i < (ModifiedLengthValue + 1) / 2; i++) {

char MemoryString[3] = { '\0' };

if (ModifiedOddLengthFlag && !i) {

sprintf(MemoryString, "%c%c", TempHexDigit, ModifiedMemory[1]);

}

else {

strncpy(MemoryString, ModifiedMemory + i \* 2, 2);

//printf("MemoryString: %s\n", MemoryString);

}

MemoryString[2] = '\0';

DumpMemory[CSADDR + AddressValue + i] = HexStringToDec(MemoryString, ForHexAddress);

}

//printf("[%s]\n", ModifiedMemory);

if (ModifiedMemory != NULL) {

free(ModifiedMemory);

ModifiedMemory = NULL;

}

return 0;

}

int LoadPassTwo(char \*FileName1, char \*FileName2, char \*FileName3) {

FILE \*FilePointer1 = NULL, \*FilePointer2 = NULL, \*FilePointer3 = NULL, \*TempFilePointer = NULL;

int LoadedFileNumber = 0, CSADDR = 0, CSLTH = 0, LoopCounter = 0;

int ErrorFlag = 0, EndFlag = 0;

if (FileName1 != NULL) {

if ((FilePointer1 = fopen(FileName1, "r")) != NULL) {

LoadedFileNumber += 1;

}

else {

ErrorFlag = 1;

}

if (FileName2 != NULL) {

if ((FilePointer2 = fopen(FileName2, "r")) != NULL) {

LoadedFileNumber += 1;

}

else {

ErrorFlag = 1;

}

if (FileName3 != NULL) {

if ((FilePointer3 = fopen(FileName3, "r")) != NULL) {

LoadedFileNumber += 1;

}

else {

ErrorFlag = 1;

}

}

}

}

if (ErrorFlag == 1) {

return 1;

}

CSADDR = ProgramStartAddress;

while ((LoopCounter < LoadedFileNumber) && !ErrorFlag) {

switch (LoopCounter) {

case 0:

TempFilePointer = FilePointer1; break;

case 1:

TempFilePointer = FilePointer2; break;

case 2:

TempFilePointer = FilePointer3; break;

default:

printf("The number of loaded file is over the limit.\n");

ErrorFlag = 1;

break;

}

char ReadString[100], \*ReadLine = NULL, AddressString[7], LengthString[3], NameString[7], \*\*ReferenceTable = NULL;

int AddressValue = 0, LengthValue = 0;

EXTSYMBOL \*TempSymbol = NULL;

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

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

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

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

ReadLine = fgets(ReadString, 20, TempFilePointer);

//printf("ReadString: %s / ReadLine: %s\n", ReadString, ReadLine);

if (ReadLine[0] == 'H') {

CSLTH = HexStringToDec(ReadLine + 13, ForHexAddress);

strncpy(AddressString, ReadLine + 7, 6);

AddressString[6] = '\0';

TrimRightBlank(AddressString);

strncpy(NameString, ReadLine + 1, 6);

NameString[6] = '\0';

TrimRightBlank(NameString);

//printf("AddressString: %s\n", AddressString);

CSADDR += HexStringToDec(AddressString, ForHexAddress);

}

else {

printf("The first line should be a header line.\n");

ErrorFlag = 1;

break;

}

while (!feof(TempFilePointer)) {

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

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

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

ReadLine = fgets(ReadString, 74, TempFilePointer);

//printf("ReadString: %s / ReadLine: %s\n", ReadString, ReadLine);

if (ReadLine[0] == 'E') {

if (ReadLine[1] != '\0') {

strncpy(AddressString, ReadLine + 1, 6);

AddressString[6] = '\0';

TrimRightBlank(AddressString);

if (AddressString[0] != '\0') {

AddressValue = HexStringToDec(AddressString, ForHexAddress);

ExecutionAddress = CSADDR + AddressValue;

}

}

EndFlag = 1;

break;

}

else if (ReadLine[0] == 'R') {

if ((ReadLine[1] >= '0' && ReadLine[1] <= '9') && (ReadLine[2] >= '0' && ReadLine[2] <= '9')) {

ReferenceTable = (char\*\*)malloc(sizeof(char\*) \* 50);

for (int i = 0; i < 50; i++) {

ReferenceTable[i] = (char\*)malloc(sizeof(char) \* 7);

memset(ReferenceTable[i], '\0', 7);

}

sprintf(ReferenceTable[1], "%s", NameString);

for (int i = 0; i < 12; i++) {

char ReferenceNumber[3] = { '\0' };

int ReferenceIndex = 0;

strncpy(ReferenceNumber, ReadLine + i \* 8 + 1, 2);

ReferenceNumber[2] = '\0';

TrimRightBlank(ReferenceNumber);

ReferenceIndex = DecStringToDec(ReferenceNumber, Word);

//printf("Reference Index: %d\n", ReferenceIndex);

strncpy(ReferenceTable[ReferenceIndex], ReadLine + i \* 8 + 3, 6);

ReferenceTable[ReferenceIndex][6] = '\0';

TrimRightBlank(ReferenceTable[ReferenceIndex]);

}

/\*

for (int i = 0; i < 50; i++) {

printf("%d: %s", i, ReferenceTable[i]);

}

printf("\n");

\*/

}

}

else if (ReadLine[0] == 'T') {

strncpy(AddressString, ReadLine + 1, 6);

AddressString[6] = '\0';

TrimRightBlank(AddressString);

AddressValue = HexStringToDec(AddressString, ForHexAddress);

strncpy(LengthString, ReadLine + 7, 2);

LengthString[2] = '\0';

LengthValue = HexStringToDec(LengthString, ForHexAddress);

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

char MemoryString[3] = { '\0' };

strncpy(MemoryString, ReadLine + 9 + (i \* 2), 2);

MemoryString[2] = '\0';

DumpMemory[CSADDR + AddressValue + i] = HexStringToDec(MemoryString, ForHexAddress);

}

}

else if (ReadLine[0] == 'M') {

char OperatorCharacter, ModifiedLength[3] = { '\0' };

int ModifiedLengthValue = 0, ModifiedOddLengthFlag = 0, AddressBlankFlag = 0;

strncpy(AddressString, ReadLine + 1, 6);

AddressString[6] = '\0';

TrimRightBlank(AddressString);

AddressValue = HexStringToDec(AddressString, ForHexAddress);

OperatorCharacter = ReadLine[9];

strncpy(ModifiedLength, ReadLine + 7, 2);

ModifiedLength[2] = '\0';

TrimRightBlank(ModifiedLength);

ModifiedLengthValue = DecStringToDec(ModifiedLength, Word);

if (ModifiedLengthValue % 2 != 0) {

ModifiedOddLengthFlag = 1;

}

if (ReferenceTable == NULL) {

char SymbolString[7] = { '\0' };

strncpy(SymbolString, ReadLine + 10, 6);

SymbolString[6] = '\0';

TrimRightBlank(SymbolString);

if (SymbolString[0] == '\0') {

AddressBlankFlag = 1;

if (LoadObjectCode(ModifiedLengthValue, ModifiedOddLengthFlag, CSADDR, AddressValue, OperatorCharacter, TempSymbol, AddressBlankFlag)) {

ErrorFlag = 1;

break;

}

}

else {

if (!SearchESTAB(SymbolString, &TempSymbol)) {

if (LoadObjectCode(ModifiedLengthValue, ModifiedOddLengthFlag, CSADDR, AddressValue, OperatorCharacter, TempSymbol, AddressBlankFlag)) {

ErrorFlag = 1;

break;

}

}

else {

printf("The symbol does not exist in the external symbol table.\n");

ErrorFlag = 1;

break;

}

}

}

else {

char ReferenceIndexString[3] = { '\0' };

int ReferenceIndex = 0, AddressBlankFlag = 0;

//printf("Modified Half Byte Length: %d / Operator: %c\n", ModifiedLengthValue, OperatorCharacter);

strncpy(ReferenceIndexString, ReadLine + 10, 2);

ReferenceIndexString[2] = '\0';

TrimRightBlank(ReferenceIndexString);

if (ReferenceIndexString[0] == '\0') {

AddressBlankFlag = 1;

if (LoadObjectCode(ModifiedLengthValue, ModifiedOddLengthFlag, CSADDR, AddressValue, OperatorCharacter, TempSymbol, AddressBlankFlag)) {

ErrorFlag = 1;

break;

}

}

else {

ReferenceIndex = DecStringToDec(ReferenceIndexString, Word);

if (!SearchESTAB(ReferenceTable[ReferenceIndex], &TempSymbol)) {

if (LoadObjectCode(ModifiedLengthValue, ModifiedOddLengthFlag, CSADDR, AddressValue, OperatorCharacter, TempSymbol, AddressBlankFlag)) {

ErrorFlag = 1;

break;

}

}

else {

printf("The symbol does not exist in the external symbol table.\n");

ErrorFlag = 1;

break;

}

}

//("RI: %d / [%s]\n", ReferenceIndex, ReferenceTable[ReferenceIndex]);

}

}

}

if (ReferenceTable != NULL) {

/\*

printf("Loop Counter: %d\n", LoopCounter);

for (int i = 0; i < 50; i++) {

printf("%d: %s", i, ReferenceTable[i]);

}

printf("\n");

\*/

for (int i = 0; i < 50; i++) {

free(ReferenceTable[i]);

}

free(ReferenceTable);

}

if (EndFlag == 0) {

ErrorFlag = 1;

break;

}

CSADDR += CSLTH;

LoopCounter++;

}

if (ErrorFlag) {

printf("An error occurs. Please check the error.\n");

ResetCommand();

return 1;

}

return 0;

}

## RunProgram.c

#include "20171665.h"

#define LDA 0X00

#define LDB 0X68

#define LDT 0X74

#define LDCH 0X50

#define STA 0X0C

#define STX 0X10

#define STL 0X14

#define STCH 0X54

#define J 0X3C

#define JSUB 0X48

#define JLT 0X38

#define JEQ 0X30

#define RSUB 0X4C

#define COMP 0X28

#define COMPR 0XA0

#define CLEAR 0XB4

#define TIXR 0XB8

#define TD 0XE0

#define RD 0XD8

#define WD 0XDC

#define LessThan 0

#define Equal 1

#define GreaterThan 2

enum RegisterNumber {

RegisterA, RegisterX, RegisterL, RegisterB, RegisterS, RegisterT, RegisterF, RegisterTemp, RegisterPC, RegisterSW

};

int Register[10] = { 0 };

int DebugFlag = 0;

int BreakPoint[0X100000] = { 0 };

int BreakPointSize = 0;

int BreakPointCheck = -1;

int BreakPointCheckFlag = 0;

int FindBreakPoint(int Address) {

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

if (BreakPoint[i] == Address) {

return 1;

}

}

return 0;

}

void SetBreakPoint(int Address) {

if (FindBreakPoint(Address)) {

return;

}

BreakPoint[BreakPointSize++] = Address;

/\*

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

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

if (BreakPoint[j] > BreakPoint[j + 1]) {

int TempBreakPoint = BreakPoint[j];

BreakPoint[j] = BreakPoint[j + 1];

BreakPoint[j + 1] = TempBreakPoint;

}

}

}

\*/

printf("\n[ok] create breakpoint %05X\n\n", Address);

return;

}

void PrintBreakPoint() {

if (BreakPointSize) {

printf("\nbreakpoints\n");

printf("-----------\n");

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

printf("%05X\n", BreakPoint[i]);

}

printf("\n");

}

else {

printf("\nno breakpoints set.\n\n");

}

return;

}

void ClearAllBreakPoint() {

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

BreakPoint[i] = 0;

}

BreakPointSize = 0;

printf("\n[ok] clear all breakpoints\n\n");

return;

}

void PrintRegister(int EndProgramAddress) {

printf("A : %06X X : %06X\n", Register[RegisterA], Register[RegisterX]);

printf("L : %06X PC: %06X\n", Register[RegisterL], Register[RegisterPC]);

printf("B : %06X S : %06X\n", Register[RegisterB], Register[RegisterS]);

printf("T : %06X\n", Register[RegisterT]);

if (EndProgramAddress == -1) {

printf("End program\n");

}

else {

printf("Stop at checkpoint[%05X]\n", EndProgramAddress);

}

}

void ExecuteInstruction(int OpcodeValue, int FirstRegister, int SecondRegister, int NI, int TargetAddress) {

int OperandValue = 0, TempValue;

if (NI == 1) {

OperandValue = TargetAddress;

}

else if (NI == 2) {

int IndirectTargetAddress;

IndirectTargetAddress = DumpMemory[TargetAddress] & 0XFF;

IndirectTargetAddress = IndirectTargetAddress << 8;

IndirectTargetAddress += DumpMemory[TargetAddress + 1] & 0XFF;

IndirectTargetAddress = IndirectTargetAddress << 8;

IndirectTargetAddress += DumpMemory[TargetAddress + 2] & 0XFF;

TargetAddress = IndirectTargetAddress;

OperandValue += DumpMemory[TargetAddress] & 0XFF;

OperandValue = OperandValue << 8;

OperandValue += DumpMemory[TargetAddress + 1] & 0XFF;

OperandValue = OperandValue << 8;

OperandValue += DumpMemory[TargetAddress + 2] & 0XFF;

}

else {

OperandValue += DumpMemory[TargetAddress] & 0XFF;

OperandValue = OperandValue << 8;

OperandValue += DumpMemory[TargetAddress + 1] & 0XFF;

OperandValue = OperandValue << 8;

OperandValue += DumpMemory[TargetAddress + 2] & 0XFF;

}

if (OperandValue >= (int)pow(2, 23)) {

OperandValue |= 0XFF000000;

}

switch (OpcodeValue) {

case LDA:

Register[RegisterA] = OperandValue;

break;

case LDB:

Register[RegisterB] = OperandValue;

break;

case LDT:

Register[RegisterT] = OperandValue;

break;

case LDCH:

Register[RegisterA] = DumpMemory[TargetAddress] & 0XFF;

break;

case STA:

TempValue = Register[RegisterA];

DumpMemory[TargetAddress + 2] = (TempValue % 256) & 0XFF;

TempValue /= 256;

DumpMemory[TargetAddress + 1] = (TempValue % 256) & 0XFF;

TempValue /= 256;

DumpMemory[TargetAddress] = (TempValue % 256) & 0XFF;

break;

case STX:

TempValue = Register[RegisterX];

DumpMemory[TargetAddress + 2] = (TempValue % 256) & 0XFF;

TempValue /= 256;

DumpMemory[TargetAddress + 1] = (TempValue % 256) & 0XFF;

TempValue /= 256;

DumpMemory[TargetAddress] = (TempValue % 256) & 0XFF;

break;

case STL:

TempValue = Register[RegisterL];

DumpMemory[TargetAddress + 2] = (TempValue % 256) & 0XFF;

TempValue /= 256;

DumpMemory[TargetAddress + 1] = (TempValue % 256) & 0XFF;

TempValue /= 256;

DumpMemory[TargetAddress] = (TempValue % 256) & 0XFF;

break;

case STCH:

TempValue = Register[RegisterA];

DumpMemory[TargetAddress] = (TempValue % 256) & 0XFF;

break;

case J:

Register[RegisterPC] = TargetAddress;

break;

case JSUB:

Register[RegisterL] = Register[RegisterPC];

Register[RegisterPC] = TargetAddress;

break;

case JLT:

if (Register[RegisterSW] == LessThan) {

Register[RegisterPC] = TargetAddress;

}

break;

case JEQ:

if (Register[RegisterSW] == Equal) {

Register[RegisterPC] = TargetAddress;

}

break;

case RSUB:

Register[RegisterPC] = Register[RegisterL];

break;

case COMP:

if (Register[RegisterA] < OperandValue) {

Register[RegisterSW] = LessThan;

}

else if (Register[RegisterA] == OperandValue) {

Register[RegisterSW] = Equal;

}

else {

Register[RegisterSW] = GreaterThan;

}

break;

case COMPR:

if (Register[FirstRegister] < Register[SecondRegister]) {

Register[RegisterSW] = LessThan;

}

else if (Register[FirstRegister] == Register[SecondRegister]) {

Register[RegisterSW] = Equal;

}

else {

Register[RegisterSW] = GreaterThan;

}

break;

case CLEAR:

Register[FirstRegister] = 0;

break;

case TIXR:

Register[RegisterX]++;

if (Register[RegisterX] < Register[FirstRegister]) {

Register[RegisterSW] = LessThan;

}

else if (Register[RegisterX] == Register[FirstRegister]) {

Register[RegisterSW] = Equal;

}

else {

Register[RegisterSW] = GreaterThan;

}

break;

case TD:

Register[RegisterSW] = LessThan;

break;

case RD:

Register[RegisterA] /= 0X100;

Register[RegisterA] \*= 0X100;

Register[RegisterA] += 0X00;

break;

case WD:

break;

default:

break;

}

}

void XBPEValue(int XBPE, int \*X, int \*B, int \*P, int \*E) {

if (XBPE % 2) {

\*E = 1;

}

else {

\*E = 0;

}

XBPE /= 2;

if (XBPE % 2) {

\*P = 1;

}

else {

\*P = 0;

}

XBPE /= 2;

if (XBPE % 2) {

\*B = 1;

}

else {

\*B = 0;

}

XBPE /= 2;

if (XBPE % 2) {

\*X = 1;

}

else {

\*X = 0;

}

return;

}

void InitializeExecution() {

for (int i = 0; i < 10; i++) {

Register[i] = 0;

}

DebugFlag = 0;

BreakPointCheck = -1;

}

int RunCommand() {

int Result = 0;

Result = RunProgram();

if (!BreakPointCheckFlag && Result <= 0) {

//Register[RegisterPC] = ExecutionAddress + ProgramTotalLength;

PrintRegister(-1);

InitializeExecution();

}

else if (BreakPointCheckFlag) {

PrintRegister(Result \* -1);

BreakPointCheckFlag = 0;

}

else {

PrintRegister(-1);

printf("[Runtime error] Check it whether the invalid memory access or opcode occurs.\n");

InitializeExecution();

return -1;

// 에러

}

return 0;

}

int BreakPointCommand(char \*Parameter1) {

if (Parameter1 != NULL) {

if (!strcmp(Parameter1, "clear")) {

ClearAllBreakPoint();

}

else {

int BreakPoint = HexStringToDec(Parameter1, ForHexAddress);

SetBreakPoint(BreakPoint);

}

}

else {

PrintBreakPoint();

}

return 0;

}

int RunProgram() {

if (!DebugFlag) {

Register[RegisterPC] = ExecutionAddress;

Register[RegisterL] = ExecutionAddress + ProgramTotalLength;

}

while (Register[RegisterPC] < MaxAddress) {

if (FindBreakPoint(Register[RegisterPC]) && BreakPointCheck != Register[RegisterPC]) {

BreakPointCheck = Register[RegisterPC];

BreakPointCheckFlag = 1;

return -1 \* Register[RegisterPC];

}

int FirstRegister = -1, SecondRegister = -1, TargetAddress = -1;

//printf("PC: %05X\n", Register[RegisterPC] & 0XFFFFF);

int OpcodeValue = (DumpMemory[Register[RegisterPC]++] & 0XFF);

int X, B, P, E, NI = OpcodeValue % 4;

X = B = P = E = 0;

DebugFlag = 1;

OpcodeValue -= NI;

//printf("Opcode: %02X\n", OpcodeValue);

HASH\* FoundHash = NULL;

if (!SearchOpcodeByValue(OpcodeValue, &FoundHash)) {

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

// 에러

return 1;

}

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

//if (Register[RegisterPC] >= MaxAddress) {

// 에러

//}

int RegisterValue = DumpMemory[Register[RegisterPC]++];

if (OpcodeValue % 4 != 0) {

// 에러

return 1;

}

FirstRegister = RegisterValue / 16;

SecondRegister = RegisterValue % 16;

if (FirstRegister > 9 || SecondRegister > 9) {

//에러

return 1;

}

}

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

//if (Register[RegisterPC] >= MaxAddress) {

// 에러

//}

int TempValue = (DumpMemory[Register[RegisterPC]++] & 0XFF);

//printf("[2] PC: %05X\n", Register[RegisterPC] & 0XFFFFF);

int FirstAddressByte = TempValue % 0X10;

int XBPE = TempValue / 0X10;

XBPEValue(XBPE, &X, &B, &P, &E);

//if (Register[RegisterPC] >= MaxAddress) {

// 에러

//}

TargetAddress = (DumpMemory[Register[RegisterPC]++] & 0xFF);

//printf("[3] PC: %05X\n", Register[RegisterPC] & 0XFFFFF);

TargetAddress += (FirstAddressByte << 8);

if (E) {

//if (Register[RegisterPC] >= MaxAddress) {

// 에러

//}

int ExtendValue = (DumpMemory[Register[RegisterPC]++] & 0XFF);

//printf("[4] PC: %05X\n", Register[RegisterPC] & 0XFFFFF);

TargetAddress = TargetAddress << 8;

TargetAddress += ExtendValue;

if (B || P) {

// 에러

return 1;

}

}

if (X) {

TargetAddress += Register[RegisterX];

}

if (!B && P) {

if (TargetAddress > 2047) {

TargetAddress |= 0XFFFFF000;

}

TargetAddress += Register[RegisterPC];

}

else if (B && !P) {

TargetAddress += Register[RegisterB];

}

/\*

if (TargetAddress < 0 || TargetAddress >= MaxAddress) {

// 에러

return 1;

}

\*/

if (NI == 2) {

/\*

if (DumpMemory[TargetAddress] < 0 || DumpMemory[TargetAddress] >= MaxAddress) {

// 에러

return 1;

}

\*/

}

}

//BreakPointCheck = Register[RegisterPC];

ExecuteInstruction(OpcodeValue, FirstRegister, SecondRegister, NI, TargetAddress);

if (Register[RegisterPC] == ExecutionAddress + ProgramTotalLength) {

break;

}

//if (FindBreakPoint(BreakPointCheck)) {

// return -1 \* BreakPointCheck;

//}

}

else {

return 0;

}

}

if (Register[RegisterPC] >= MaxAddress) {

Register[RegisterPC] = 0;

}

return 0;

}