시스템 프로그래밍

담당 교수: 김지환

< Project 2 >

SIC/XE Machine Assembler 구현

서강대학교 컴퓨터공학과

학번: 20161577

이름: 김인호

제출 마감:  
2018년 04월 09일 (월)

**목차**

[1. 프로그램 개요 5](#_Toc511075688)

[2. 프로그램 설명 5](#_Toc511075689)

[2.1. 프로그램 흐름도 5](#_Toc511075690)

[2.2. Assembler 알고리즘 설명 6](#_Toc511075691)

[2.2.1. 사전 단계 6](#_Toc511075692)

[2.2.2. Pass1 – Symbol Table 생성과 assembly 소스 코드 분석 6](#_Toc511075693)

[2.2.3. Pass2 – list 파일과 object 파일 생성 7](#_Toc511075694)

[3. 모듈 정의 8](#_Toc511075695)

[3.1. 모듈 이름: assembleCMD(INPUT\_CMD) 8](#_Toc511075696)

[3.1.1. 기능 8](#_Toc511075697)

[3.1.2. 사용 변수 8](#_Toc511075698)

[3.2. 모듈 이름: assemblerPass1(FILE\*, int\*, int\*) 8](#_Toc511075699)

[3.2.1. 기능 8](#_Toc511075700)

[3.2.2. 사용 변수 8](#_Toc511075701)

[3.3. 모듈 이름: assemblerPass2(FILE\*, FILE\*, int, int) 9](#_Toc511075702)

[3.3.1. 기능 9](#_Toc511075703)

[3.3.2. 사용 변수 9](#_Toc511075704)

[3.4. 모듈 이름: printLST(FILE\*, ASM\_SRC\*, int, bool, bool) 10](#_Toc511075705)

[3.4.1. 기능 10](#_Toc511075706)

[3.4.2. 사용 변수 10](#_Toc511075707)

[3.5. 모듈 이름: printOBJ(FILE\*, ASM\_SRC\*, int, int) 10](#_Toc511075708)

[3.5.1. 기능 10](#_Toc511075709)

[3.5.2. 사용 변수 10](#_Toc511075710)

[3.6. 모듈 이름: printOBJList(FILE\*) 11](#_Toc511075711)

[3.6.1. 기능 11](#_Toc511075712)

[3.6.2. 사용 변수 11](#_Toc511075713)

[3.7. 모듈 이름: printModList(FILE\*) 11](#_Toc511075714)

[3.7.1. 기능 11](#_Toc511075715)

[3.7.2. 사용 변수 11](#_Toc511075716)

[3.8. 모듈 이름: addOBJList(ASM\_SRC\*) 11](#_Toc511075717)

[3.8.1. 기능 11](#_Toc511075718)

[3.8.2. 사용 변수 11](#_Toc511075719)

[3.9. 모듈 이름: addModList(int, int) 12](#_Toc511075720)

[3.9.1. 기능 12](#_Toc511075721)

[3.9.2. 사용 변수 12](#_Toc511075722)

[3.10. 모듈 이름: parseListFree() 12](#_Toc511075723)

[3.10.1. 기능 12](#_Toc511075724)

[3.10.2. 사용 변수 12](#_Toc511075725)

[3.11. 모듈 이름: objListFree() 12](#_Toc511075726)

[3.11.1. 기능 12](#_Toc511075727)

[3.11.2. 사용 변수 12](#_Toc511075728)

[3.12. 모듈 이름: modListFree() 13](#_Toc511075729)

[3.12.1. 기능 13](#_Toc511075730)

[3.12.2. 사용 변수 13](#_Toc511075731)

[3.13. 모듈 이름: printASMError(ASM\_SRC\*) 13](#_Toc511075732)

[3.13.1. 기능 13](#_Toc511075733)

[3.13.2. 사용 변수 13](#_Toc511075734)

[3.14. 모듈 이름: symbolCMD() 13](#_Toc511075735)

[3.14.1. 기능 13](#_Toc511075736)

[3.14.2. 사용 변수 13](#_Toc511075737)

[3.15. 모듈 이름: parsedASM(char\*) 14](#_Toc511075738)

[3.15.1. 기능 14](#_Toc511075739)

[3.15.2. 사용 변수 14](#_Toc511075740)

[3.16. 모듈 이름: isRegister(char) 14](#_Toc511075741)

[3.16.1. 기능 14](#_Toc511075742)

[3.16.2. 사용 변수 14](#_Toc511075743)

[3.17. 모듈 이름: setError(ASM\_SRC\*, ASM\_ERROR) 14](#_Toc511075744)

[3.17.1. 기능 14](#_Toc511075745)

[3.17.2. 사용 변수 14](#_Toc511075746)

[3.18. 모듈 이름: symTableAdd(char\*, int) 15](#_Toc511075747)

[3.18.1. 기능 15](#_Toc511075748)

[3.18.2. 사용 변수 15](#_Toc511075749)

[3.19. 모듈 이름: symTableSearch(char\*) 15](#_Toc511075750)

[3.19.1. 기능 15](#_Toc511075751)

[3.19.2. 사용 변수 15](#_Toc511075752)

[3.20. 모듈 이름: symTableFree() 15](#_Toc511075753)

[3.20.1. 기능 15](#_Toc511075754)

[3.20.2. 사용 변수 15](#_Toc511075755)

[3.21. 모듈 이름: typeCMD(INPUT\_CMD) 16](#_Toc511075756)

[3.21.1. 기능 16](#_Toc511075757)

[3.21.2. 사용 변수 16](#_Toc511075758)

[4. 전역 변수 정의 16](#_Toc511075759)

[4.1. SYMBOL\_ENTRY\* symTable 16](#_Toc511075760)

[4.2. ASM\_SRC\* parseList 16](#_Toc511075761)

[4.3. OBJ\_CODE\* objList 16](#_Toc511075762)

[4.4. MOD\_RECORD\* modList 16](#_Toc511075763)

[4.5. char directives[][] 17](#_Toc511075764)

[4.6. char registers[] 17](#_Toc511075765)

[5. 코드 17](#_Toc511075766)

[5.1. 20161577.h 17](#_Toc511075767)

[5.2. 20161577.c 20](#_Toc511075768)

[5.3. cmdProc.h 22](#_Toc511075769)

[5.4. cmdProc.c 22](#_Toc511075770)

[5.5. shell.h 26](#_Toc511075771)

[5.6. shell.c 27](#_Toc511075772)

[5.7. memory.h 29](#_Toc511075773)

[5.8. memory.c 30](#_Toc511075774)

[5.9. hash.h 31](#_Toc511075775)

[5.10. hash.c 32](#_Toc511075776)

[5.11. assembler.h 35](#_Toc511075777)

[5.12. assembler.c 35](#_Toc511075778)

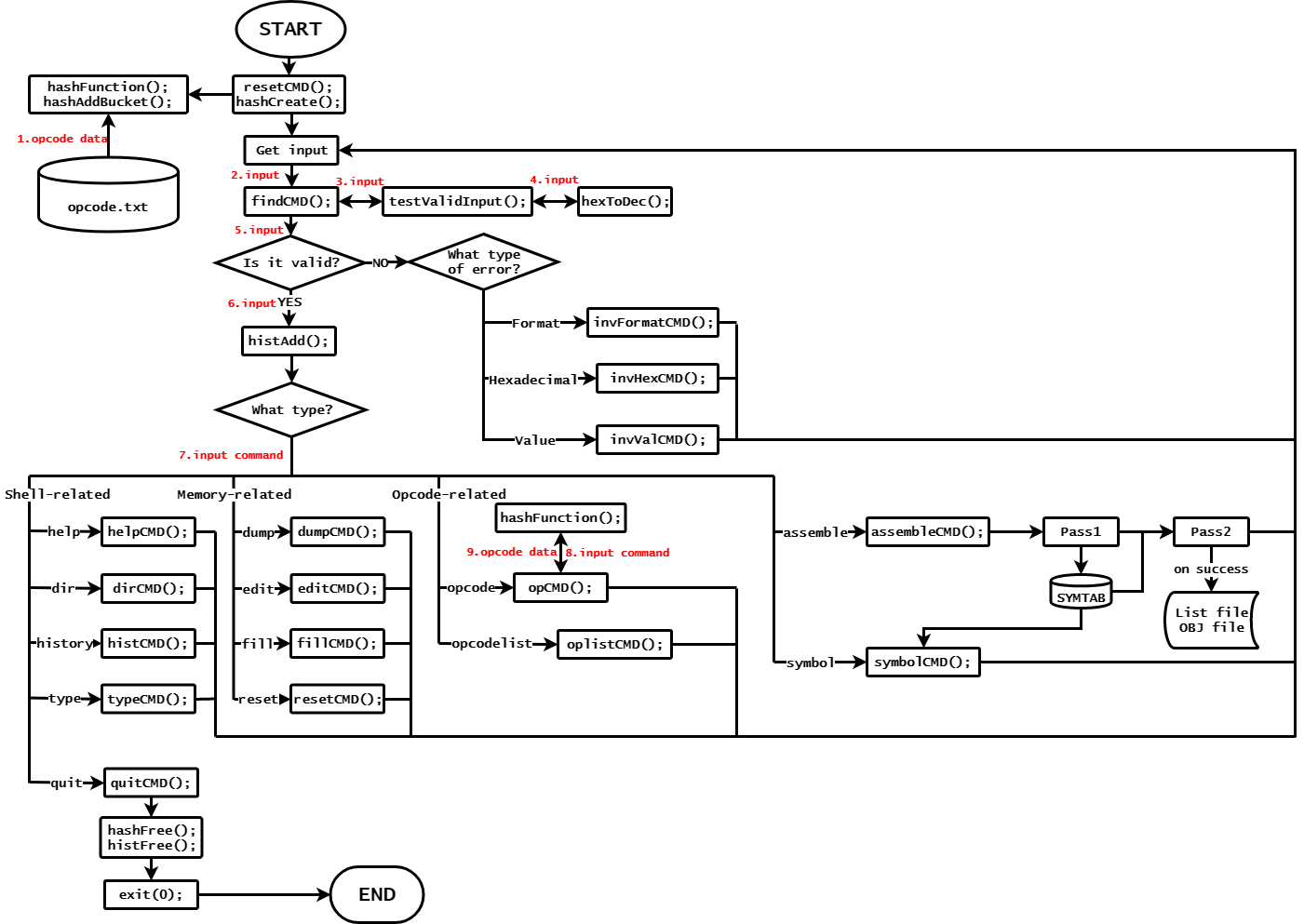
[5.13. Makefile 52](#_Toc511075779)

# 프로그램 개요

사전에 구현한 shell 환경에 추가되는 기능으로, SIC/XE Machine Assembler를 구현한다. 해당 directory에 있는 .asm 소스파일을 assemble 하여 결과를 분석할 수 있는 list 파일과 후에 loader와 linker단계에서 사용될 object 파일을 생성한다.

# 프로그램 설명

## 프로그램 흐름도



## Assembler 알고리즘 설명

구현된 SIC/XE Machine Assembler는 주어진 .asm 소스파일을 총 2번의 읽는 과정 (Pass1과 Pass2)을 거쳐 list 파일과 object 파일을 생성하는 알고리즘이다.

### 사전 단계

Project 1단계에서 구현한 shell 환경에서 assemble filename 명령어가 입력되면 먼저 filename의 이름으로 되어있는 파일이 존재하고 .asm 확장자를 갖는지 확인한다. 그 후 과거의 assemble로 인해 생성된 symbol table, assembly 소스 분석 리스트, object code list, modification record list 등을 free 해주고 각 리스트를 초기화하는 과정을 거친다. 마지막으로 list 파일과 object 파일을 각각의 파일 포인터로 생성한 후 Pass1 함수와 Pass2 함수를 호출해준다.

### Pass1 – Symbol Table 생성과 assembly 소스 코드 분석

명령어 인자로 입력된 .asm 파일의 내용을 한 줄씩 읽으면서 분석을 한다. Label, instruction, operand 각 필드에 대한 에러를 확인하고 주석과 instruction과 directives를 구분하고 각 줄 별로 분석한 결과를 linked list 형태로 저장한다. 각 분석된 줄을 따라 Pass1 알고리즘을 본격적으로 실행한다. 먼저 START directive가 있으면 location counter를 명시된 시작 주소로 초기화 해주고 START directive 가 없으면 임의로 (해당 구현에서는 0x0000) 정해준다. 또한 5부터 시작하여 각 줄마다 5의 배수를 차례대로 지정한다.

모든 주석이 아닌 줄에 대해서 label 필드에 symbol이 있다면 해당 symbol이 symbol table에 존재하는지 확인하고 만약에 있으면 에러 처리한다. 새로운 symbol이 발견되면 symbol table에 symbol 기준 내림차순으로 저장한다. 또한 앞으로의 과정에서 유용하게 쓰일 각 instruction의 byte 크기를 저장한다. 그리고 이 byte 크기만큼 location counter를 증가해주어 다음 instruction의 location을 구한다. END directive 가 있을 때까지 진행한다.

Symbol table의 내용은 assembler 과정이 성공적으로 끝난 후 symbol 명령어로 확인할 수 있다.

### Pass2 – list 파일과 object 파일 생성

Assembly 소스 파일의 각 instruction의 object code를 찾는 과정이다. 먼저 PC 레지스터를 시작 주소로 초기화하고 BASE directive가 나왔을 때 base 레지스터를 초기화해준다. 각 instruction을 읽으면서 PC 레지스터를 다음 instruction의 주소로 놓고 각 instruction의 operand의 참조 방식에 따라 simple addressing으로 PC 또는 Base relative인지, immediate addressing 인지 indirect addressing인지 확인하여 object code를 계산한다.

List 파일에는 각 assembly 소스 줄 전에 line number랑 location counter를 출력하고 소스를 출력한 후에는 해당 instruction의 object code를 출력한다. 주석의 경우에는 location 과 object code 모두 출력하지 않는다.

Object 파일에는 맨 첫 줄에 Header record로 프로그램의 이름과 시작 주소와 총 프로그램 크기를 출력한다. 그 다음 instruction의 object code를 Text record로 최대 길이 0x1E 내로 또는 변수가 나오면 새로운 줄에 출력하는 형식에 맞추고 프로그램 relocation이 있을 경우 수정되어야 할 주소 (format 4 instruction)들은 Modification record에 출력하고 맨 마지막에는 End record에 프로그램 첫 instruction의 주소를 출력한다.

이렇게 해서 성공적으로 모든 assemble 과정이 끝날 경우 현재 directory에 .lst 파일과 .obj 확장자로 list 파일과 object 파일이 생성된다. 각 파일의 내용은 type 명령어로 확인할 수 있다.

# 모듈 정의

## 모듈 이름: assembleCMD(INPUT\_CMD)

### 기능

Shell 환경에서 assemble filename 이 입력 됐을 때 호출되는 함수로 필요한 파일 포인터를 준비하고 Pass1과 Pass2에 대한 함수를 호출해준다..

### 사용 변수

|  |  |
| --- | --- |
| INPUT\_CMD ipcmd | 함수 인자, 사용자의 입력을 저장 변수 |
| FILE\* srcFile | 소스 파일 포인터 |
| FILE\* lstFile | List 파일 포인터 |
| FILE\* objFile | Object 파일 포인터 |
| char lstName[] | List 파일 이름 |
| char objName[] | Object 파일 이름 |
| int maxSrcLen | 최대 소스 길이 |
| int progLen | 프로그램 총 길이 |

## 모듈 이름: assemblerPass1(FILE\*, int\*, int\*)

### 기능

앞서 설명된 assembler의 Pass1 알고리즘을 실행하는 함수이다. Symbol table을 만들고 각 instruction의 location을 정한다.

### 사용 변수

|  |  |
| --- | --- |
| FILE\* srcFile | 함수 인자, 소스 파일 포인터 |
| int\* maxSrcLen | 함수 인자, 최대 소스 길이 |
| int\* progLen | 함수 인자, 프로그램 총 길이 |
| ASM\_SRC\* curParse | 현재 분석 중인 소스에 대한 정보 |
| ASM\_SRC\* prevParse | 이전 소스 줄의 분석 정보 |
| char source[] | 소스 string |
| int location | Pass1 알고리즘에 쓰이는 location counter |
| int lineNum | 각 소스 줄의 line number |
| bool errorFlag | 에러가 발견됐을 때 true 가 되는 flag |
| int mult | byte 크기 계산 용 변수 |

## 모듈 이름: assemblerPass2(FILE\*, FILE\*, int, int)

### 기능

앞서 설명된 assembler의 Pass2 알고리즘을 실행하는 함수이다. List 파일과 object 파일을 생성한다.

### 사용 변수

|  |  |
| --- | --- |
| FILE\*lstFile | 함수 인자, list 파일 포인터 |
| FILE\* objFile | 함수 인자, object 파일 포인터 |
| int maxSrcLen | 함수 인자, 최대 소스 길이 |
| int progLen | 함수 인자, 프로그램 총 길이 |
| ASM\_SRC\* curParse | 현재 분석중인 소스 |
| SYMBOL\_ENTRY\* curSymbol | 현재 발견된 symbol |
| int pcReg | PC 레지스터의 값 |
| int baseReg | BASE 레지스터의 값 |
| bool errorFlag | error 발생 시 true |
| bool locFlag | location 출력 정하는 flag |
| bool objFlag | object 출력 정하는 flag |
| bool nFlag | object code의 n flag 값 |
| bool iFlag | object code의 i flag 값 |
| bool xFlag | object code의 x flag 값 |
| bool bFlag | object code의 b flag 값 |
| bool pFlag | object code의 p flag 값 |
| bool eFlag | object code의 e flag 값 |
| int firstExec | 프로그램의 첫 instruction 주소 |

## 모듈 이름: printLST(FILE\*, ASM\_SRC\*, int, bool, bool)

### 기능

List 파일에 assemble 결과를 알 수 있는 내용을 출력한다. 앞서 설명된 알고리즘에 따라 호출된다.

### 사용 변수

|  |  |
| --- | --- |
| FILE\* lstFile | 함수 인자, list 파일 포인터 |
| ASM\_SRC\* parsedASM | 함수 인자, 현재 처리 중인 소스 |
| int maxSrcLen | 함수 인자, 소스 최대 길이 |
| bool printLOC | 함수 인자, location 출력 여부 flag |
| bool printOBJ | 함수 인자, object code 출력 여부 flag |

## 모듈 이름: printOBJ(FILE\*, ASM\_SRC\*, int, int)

### 기능

Object 파일에 assemble 결과를 출력한다. 앞서 설명된 알고리즘에 따라 호출된다.

### 사용 변수

|  |  |
| --- | --- |
| FILE\* objFile | 함수 인자, object 파일 포인터 |
| ASM\_SRC\* parsedASM | 함수 인자, 현재 처리 중인 소스 |
| int progLen | 함수 인자, 프로그램 길이 |
| int firstExec | 함수 인자, 프로그램 첫 instruction 주소 |
| static int lineLen | text record 길이 기억 변수 |

## 모듈 이름: printOBJList(FILE\*)

### 기능

Object 파일에 text record 한 줄을 출력한다.

### 사용 변수

|  |  |
| --- | --- |
| FILE\* objFile | 함수 인자, object 파일 포인터 |
| OBJ\_CODE\* cur | 현재 object code node |

## 모듈 이름: printModList(FILE\*)

### 기능

Modification record를 object 파일에 출력한다.

### 사용 변수

|  |  |
| --- | --- |
| FILE\* objFile | 함수 인자, object 파일 포인터 |
| MOD\_RECORD\* cur | 현재 modification record node |

## 모듈 이름: addOBJList(ASM\_SRC\*)

### 기능

Object code list에 새로운 object code를 linked list 형태로 추가해준다.

### 사용 변수

|  |  |
| --- | --- |
| ASM\_SRC\* parsedASM | 함수 인자, 현재 분석 중인 소스 |
| OBJ\_CODE\* newOBJ | 새 object code node |
| OBJ\_CODE\* cur | list 탐색 용 포인터 |

## 모듈 이름: addModList(int, int)

### 기능

Modification record list에 새로운 modification record를 linked list 형태로 추가해준다.

### 사용 변수

|  |  |
| --- | --- |
| int location | 함수 인자, relocation을 수행할 instruction의 위치 |
| int length | 함수 인자, relocation을 수행할 instruction의 위치 |
| MOD\_RECORD\* newRec | 새 modification record node |
| MOD\_RECORD\* cur | list 탐색 용 포인터 |

## 모듈 이름: parseListFree()

### 기능

Assembly 소스 분석 linked list에 할당 된 메모리를 해제 해주는 함수이다.

### 사용 변수

|  |  |
| --- | --- |
| ASM\_SRC\* cur | linked list 탐색 용 포인터 |
| ASM\_SRC\* next | linked list 탐색 용 포인터 |

## 모듈 이름: objListFree()

### 기능

Object code linked list에 할당 된 메모리를 해제 해주는 함수이다.

### 사용 변수

|  |  |
| --- | --- |
| OBJ\_CODE\* cur | linked list 탐색 용 포인터 |
| OBJ\_CODE\* next | linked list 탐색 용 포인터 |

## 모듈 이름: modListFree()

### 기능

Modification record linked list에 할당 된 메모리를 해제 해주는 함수이다.

### 사용 변수

|  |  |
| --- | --- |
| MOD\_RECORD\* cur | linked list 탐색 용 포인터 |
| MOD\_RECORD\* next | linked list 탐색 용 포인터 |

## 모듈 이름: printASMError(ASM\_SRC\*)

### 기능

발견된 에러와 종류에 따라 에러 메시지를 출력한다.

### 사용 변수

|  |  |
| --- | --- |
| ASM\_SRC\* parsedASM | 에러가 담긴 소스 분석 node |

## 모듈 이름: symbolCMD()

### 기능

가장 최근에 assemble된 소스에 대한 symbol table의 내용을 출력한다.

### 사용 변수

|  |  |
| --- | --- |
| SYMBOL\_ENTRY\* cur | Symbol Table 탐색 용 포인터 |

## 모듈 이름: parsedASM(char\*)

### 기능

소스 코드 한 줄을 분석하여 종류와 각 필드의 내용 그리고 필요한 여러 정보를 알아내는 함수이다.

### 사용 변수

|  |  |
| --- | --- |
| char\* source | 함수 인자, assembly source 한 줄 |
| char delim[] | string tokenizing 에 사용될 기준 문자 |
| char tmp[] | 소스를 임시적으로 저장할 변수 |
| char \*tok | tokenizing에 쓰일 변수 |

## 모듈 이름: isRegister(char)

### 기능

인자로 받는 character가 register이름 중 하나인지 확인한다.

### 사용 변수

|  |  |
| --- | --- |
| char reg | 함수 인자, 확인 할 character |

## 모듈 이름: setError(ASM\_SRC\*, ASM\_ERROR)

### 기능

잘못된 소스에 에러에 대한 정보를 저장한다.

### 사용 변수

|  |  |
| --- | --- |
| ASM\_SRC\* parsedResult | 함수 인자, 에러 정보를 저장할 node |
| ASM\_ERROR error | 저장될 에러 종류 |

.

## 모듈 이름: symTableAdd(char\*, int)

### 기능

새로 발견된 symbol을 symbol table 에 symbol 기준 내림차 순으로 추가해주는 함수이다.

### 사용 변수

|  |  |
| --- | --- |
| char\* symbol | 함수 인자, 새로 추가할 symbol string |
| int address | 함수 인자, symbol의 주소 |
| SYMBOL\_ENTRY\* cur | linked list 탐색 용 포인터 |
| SYMBOL\_ENTRY\* newEntry | 새로 추가할 symbol node |

## 모듈 이름: symTableSearch(char\*)

### 기능

Symbol Table 에서 주어진 symbol을 찾는 함수이다. 발견되지 않으면 NULL이 return 된다.

### 사용 변수

|  |  |
| --- | --- |
| char\* symbol | symbol table에서 찾을 symbol string |
| SYMBOL\_ENTRY\* cur | linked list 탐색 용 포인터 |

## 모듈 이름: symTableFree()

### 기능

Symbol Table에 할당된 메모리를 해제하는 함수이다.

### 사용 변수

|  |  |
| --- | --- |
| SYMBOL\_ENTRY\* cur | linked list 탐색 용 포인터 |
| SYMBOL\_ENTRY\* next | linked list 탐색 용 포인터 |

## 모듈 이름: typeCMD(INPUT\_CMD)

### 기능

Shell 환경에서 type 명령어 입력 시 수행되는 함수이다. 인자로 입력된 파일의 내용을 화면에 출력한다.

### 사용 변수

|  |  |
| --- | --- |
| INPUT\_CMD ipcmd | 함수 인자, 입력된 명령어 정보 |
| FILE\* fp | 화면에 출력할 파일의 포인터 |
| char c | 출력에 쓰이는 character |

# 전역 변수 정의

## SYMBOL\_ENTRY\* symTable

Symbol Table의 첫 symbol을 가리키는 pointer.

## ASM\_SRC\* parseList

Assembly 소스 분석 linked list의 첫 node를 가리키는 pointer.

## OBJ\_CODE\* objList

Object code linked list의 첫 node를 가리키는 pointer.

## MOD\_RECORD\* modList

Modification record linked list의 첫 node를 가리키는 pointer.

## char directives[][]

인식되는 directives를 저장하는 character array (string) 배열.

## char registers[]

레지스터 이름을 저장한 character array.

# 코드

## 20161577.h

/\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*

\* \*

\* Sogang University \*

\* Department of Computer Science and Engineering \*

\* \*

\* Subject name: System Programming \*

\* Project title: [2] SIC/XE Machine - The Assembler \*

\* \*

\* Author: Inho Kim \*

\* Student ID: 20161577 \*

\* \*

\* File name: 20161577.h \*

\* File description: Main header file for the project. \*

\* \*

\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*/

#include <stdio.h>

#include <string.h>

#include <ctype.h>

#include <stdlib.h>

#include <math.h>

#include <dirent.h>

#include <sys/types.h>

#include <sys/stat.h>

#define CMD\_LEN 257 // maximum length of input string

#define CMD\_CNT 14 // list of command formats

#define ARG\_MAX 3 // maximum argument count

#define MEM\_SIZE 1048576 // 2^20 = 1MB

#define HASH\_SIZE 20 // hash table size

#define ASM\_LEN 121 // maximum length of single asm source line

#define INC\_BYTE 16 \* 16 // increase byte in object code

#define INC\_HBYTE 16 // increase half byte in object code

#define MAX12DISP 4096 // highest number expressable with 12 bits

#define MAX15ADDR 32768 // highest number expressable with 15 bits

typedef struct dirent ENTRY; // for dir command

typedef struct stat STBUF; // for dir command

typedef enum { false, true } bool; // workaround for bool type in C

// error codes, for better readability

typedef enum {

SAFE, FORMAT, HEX, VALUE, FILENAME

} ERROR\_CODE;

// command types, for better readability

typedef enum {

shell, memory, opcode, assembler, invalid

} CMD\_TYPE;

// command functions, for better readablitiy

typedef enum {

help, dir, quit, hist, dump, edit, fill, reset, op, oplist, type, assemble, symbol, invFormat, invHex, invVal, invFile

} CMD\_FUNC;

// assembly source error type, for better readability

typedef enum {

OK, SYMBOL, INSTRUCTION, OPERAND

} ASM\_ERROR;

// directives list, for better readability

typedef enum {

NOTDR, START, END, BASE, BYTE, WORD, RESB, RESW

} DIREC\_NAME;

// register list, for better readability

typedef enum {

A, X, L, B, S, T, F

} REG\_NAME;

// command format structure

typedef struct {

char str[CMD\_LEN];

char abb[CMD\_LEN];

CMD\_TYPE category;

CMD\_FUNC func;

bool arg;

} COMMAND;

// user input command parsed structure

typedef struct {

CMD\_FUNC cmd;

short argCnt;

char arg[ARG\_MAX][10];

} INPUT\_CMD;

// history node structure

typedef struct HIST\_STRUCT {

char str[CMD\_LEN];

struct HIST\_STRUCT\* next;

} HIST\_NODE;

// hash table bucket structure

typedef struct HASH\_STRUCT {

int codeVal;

char code[3];

char inst[CMD\_LEN];

enum { f1, f2, f34 } format;

int operandCnt;

struct HASH\_STRUCT\* next;

} HASH\_ENTRY;

// assembly source parse structure

typedef struct ASM\_STRUCT {

char source[ASM\_LEN]; // source code

char label[ASM\_LEN]; // label field

char inst[ASM\_LEN]; // instruction field

char operand[2][ASM\_LEN]; // operand field

bool hasLabel; // flag for having label

bool indexing; // flag for having indexing mode

int lineNum; // line number in list file

int location; // location counter of instruction

int operandCnt; // operand count

int byteSize; // size of instruction in bytes

unsigned objCode; // value of object code

enum { ERROR, INST, PSEUDO, COMMENT } type; // type of source line

enum { NONE, format1, format2, format3, format4 } format; // type of instruction

DIREC\_NAME direcName; // directive type

ASM\_ERROR errorCode; // error type

struct ASM\_STRUCT\* next; // pointer to next assembly source

} ASM\_SRC;

// symbol table entry structure

typedef struct SYMBOL\_STRUCT {

char symbol[ASM\_LEN];

int address;

struct SYMBOL\_STRUCT\* next;

} SYMBOL\_ENTRY;

// Object Code list structure

typedef struct OBJ\_STRUCT {

unsigned objCode;

int location;

int byteSize;

struct OBJ\_STRUCT\* next;

} OBJ\_CODE;

// Modification Record list structure

typedef struct MOD\_STRUCT {

int location;

int lenHB;

struct MOD\_STRUCT\* next;

} MOD\_RECORD;

int hexToDec(char\*); // function to check for vaild hex value and return converted decimal value

void hashFree(); // function to free hash table memory allocation

void symTableFree(); // function to free SYMTAB

void parseListFree(); // function to free ASM parse list

void objListFree(); // function to free OBJ code list

void modListFree(); // function to free Modification Record list

HASH\_ENTRY\* bucketSearch(char\*); // function to search bucket

## 20161577.c

/\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*

\* \*

\* Sogang University \*

\* Department of Computer Science and Engineering \*

\* \*

\* Subject name: System Programming \*

\* Project title: [2] SIC/XE Machine - The Assembler \*

\* \*

\* Author: Inho Kim \*

\* Student ID: 20161577 \*

\* \*

\* File name: 20161577.c \*

\* File description: Main file for the project. \*

\* \*

\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*/

#include "20161577.h"

#include "cmdProc.h"

#include "shell.h"

#include "memory.h"

#include "hash.h"

#include "assembler.h"

int main() {

char inp[CMD\_LEN]; // input string

char tmp[CMD\_LEN]; // temporary string to copy input

int i, j;

INPUT\_CMD input; // storage for parsed input

resetCMD(); // initialize memory

hashCreate(); // create hash table of opcodes

while(1) {

printf("sicsim> ");

fgets(inp, CMD\_LEN, stdin); // get input string

inp[strlen(inp) - 1] = '\0'; // replace \n with null character

// copy input string to tmp but place one space before and after comma ','

j = 0;

for(i = 0; inp[i]; i++) {

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

strcpy(tmp + j, " , "); // place space around commma ','

j += 3;

}

else

tmp[j++] = inp[i];

}

tmp[j] = '\0';

input = findCMD(tmp); // find the command format from input string

if(input.cmd < invFormat && input.cmd != assemble)

histAdd(inp); // if command is not invalid add to history

// call function for each command

switch(input.cmd) {

case help:

helpCMD();

break;

case dir:

dirCMD();

break;

case quit:

quitCMD();

break;

case hist:

histCMD();

break;

case dump:

dumpCMD(input);

break;

case edit:

editCMD(input);

break;

case fill:

fillCMD(input);

break;

case reset:

resetCMD();

break;

case op:

opCMD(input);

break;

case oplist:

oplistCMD();

break;

case type:

typeCMD(input);

break;

case assemble:

if(assembleCMD(input))

histAdd(inp);

break;

case symbol:

symbolCMD();

break;

case invFormat:

invFormatCMD();

break;

case invHex:

invHexCMD();

break;

case invVal:

invValCMD();

break;

case invFile:

invFileCMD();

break;

}

}

return 0;

}

## cmdProc.h

/\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*

\* \*

\* Sogang University \*

\* Department of Computer Science and Engineering \*

\* \*

\* Subject name: System Programming \*

\* Project title: [2] SIC/XE Machine - The Assembler \*

\* \*

\* Author: Inho Kim \*

\* Student ID: 20161577 \*

\* \*

\* File name: cmdProc.h \*

\* File description: Header file for input processing tasks. \*

\* \*

\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*/

INPUT\_CMD findCMD(char\*); // function to find command type of input

ERROR\_CODE testValidInput(INPUT\_CMD, COMMAND); // function to check validity of input command

void invFormatCMD(); // called when invalid format input

void invHexCMD(); // called when invalid hexadecimal value

void invValCMD(); // called when invalid value

void invFileCMD(); // called when invalid filename

## cmdProc.c

/\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*

\* \*

\* Sogang University \*

\* Department of Computer Science and Engineering \*

\* \*

\* Subject name: System Programming \*

\* Project title: [2] SIC/XE Machine - The Assembler \*

\* \*

\* Author: Inho Kim \*

\* Student ID: 20161577 \*

\* \*

\* File name: cmdProc.c \*

\* File description: Tasks related to parse input by user. \*

\* \*

\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*/

#include "20161577.h"

#include "cmdProc.h"

// store command format in structure array to compare

COMMAND cmdList[CMD\_CNT] = {

{ "help", "h", shell, help, false }, { "dir", "d", shell, dir, false },

{ "quit", "q", shell, quit, false }, { "history", "hi", shell, hist, false },

{ "dump", "du", memory, dump, true }, { "edit", "e", memory, edit, true },

{ "fill", "f", memory, fill, true }, { "reset", "reset", memory, reset, false },

{ "opcode", "opcode", opcode, op, true }, { "opcodelist", "opcodelist", opcode, oplist, false },

{ "type", "type", shell, type, true }, { "assemble", "assemble", assembler, assemble, true },

{ "symbol", "symbol", assembler, symbol, false }, { "invalid", "invalid", invalid, invFormat, true }

};

INPUT\_CMD findCMD(char\* str) {

int i, j;

char delim[] = " \t\n"; // characters used to tokenize

char inp[CMD\_LEN];

char\* tok;

INPUT\_CMD ipcmd;

strcpy(inp, str); // copy input string

// initialize as invalid

ipcmd.cmd = invFormat;

ipcmd.argCnt = 0;

if(!strlen(str))

return ipcmd; // if empty string, return as invalid

tok = strtok(inp, delim); // first word of input

if(!tok)

return ipcmd; // emtpy token

for(i = 0; i < CMD\_CNT - 1; i++)

if(!strcmp(tok, cmdList[i].str) || !strcmp(tok, cmdList[i].abb)) {

ipcmd.cmd = cmdList[i].func; // if input command matches one of hard coded commands

break;

}

if(ipcmd.cmd == invFormat) // invalid command

return ipcmd;

// get arguments

j = 0;

while(tok) {

tok = strtok(NULL, delim); // next token (NOT expected comma if valid command)

if(!j && !tok) // no argument for command

break;

if((j && !tok) || tok[0] == ',') { // there was a previous argument but empty token or comma found

ipcmd.cmd = invFormat; //invalid command

return ipcmd;

}

strcpy(ipcmd.arg[j++], tok); // copy argument to input command structure

tok = strtok(NULL, delim); // next token (expected a comma if valid command)

if(tok && tok[0] != ',') { // if token not empty, expected a comma

ipcmd.cmd = invFormat;

return ipcmd;

}

}

ipcmd.argCnt = j; // save argument count

// after input string parsed, do further check for validity

switch(testValidInput(ipcmd, cmdList[i])) { // get error type, if any

case FORMAT:

ipcmd.cmd = invFormat;

break;

case HEX:

ipcmd.cmd = invHex;

break;

case VALUE:

ipcmd.cmd = invVal;

break;

case FILENAME:

ipcmd.cmd = invFile;

break;

default: // no error found

break;

}

return ipcmd;

}

ERROR\_CODE testValidInput(INPUT\_CMD ipcmd, COMMAND format) {

int i;

int arg[3];

ERROR\_CODE code = SAFE; // initialize as correct command

if(ipcmd.cmd == invFormat)

return FORMAT;

// check argument count

switch(ipcmd.cmd) {

// strictly 0 arguments

case help:

case dir:

case quit:

case hist:

case reset:

case oplist:

case symbol:

if(ipcmd.argCnt)

code = FORMAT;

break;

// strictly 1 argument

case op:

case type:

case assemble:

if(ipcmd.argCnt != 1)

code = FORMAT;

break;

// strictly 2 arguments

case edit:

if(ipcmd.argCnt != 2)

code = FORMAT;

break;

// strictly 3 arguments

case fill:

if(ipcmd.argCnt != 3)

code = FORMAT;

break;

// need less than 3

case dump:

if(ipcmd.argCnt > 2)

code = FORMAT;

break;

default:

break;

}

if(code == FORMAT)

return code;

// check hexadecimal number if command is memory-related

if(format.category == memory) {

for(i = 0; i < ipcmd.argCnt; i++)

if((arg[i] = hexToDec(ipcmd.arg[i])) == -1)

code = HEX;

// check with each command's criteria

switch(ipcmd.cmd) {

case edit:

if(arg[0] >= MEM\_SIZE || arg[1] > 255)

code = VALUE;

break;

case fill:

if(arg[0] >= MEM\_SIZE || arg[1] >= MEM\_SIZE || arg[0] > arg[1] || arg[2] > 255)

code = VALUE;

break;

case dump:

switch(ipcmd.argCnt) {

case 2:

if(arg[1] >= MEM\_SIZE || arg[0] > arg[1])

code = VALUE;

case 1:

if(arg[0] >= MEM\_SIZE)

code = VALUE;

break;

}

break;

default:

break;

}

}

else if(format.category == assembler) {

if(ipcmd.cmd == assemble)

if(strcmp(ipcmd.arg[0] + strlen(ipcmd.arg[0]) - 4, ".asm"))

code = FILENAME;

}

return code;

}

void invFormatCMD() {

puts("ERROR: Invalid command.");

puts("Type \"help\" for list and formats of commands.");

}

void invHexCMD() {

puts("ERROR: Incorrect hexadecimal.");

}

void invValCMD() {

puts("ERROR: Invalid address.");

puts("Memory size:\t\t1MB [0x00000 ~ 0xFFFFF]");

puts("Edit/Fill value range:\t 1B [0x00 ~ 0xFF]");

}

void invFileCMD() {

puts("ERROR: Invalid filename.");

puts("Assembly source file extension must be .asm");

}

int hexToDec(char\* hex) {

int i, dec = 0, multiplier = 1;

for(i = strlen(hex) - 1; i >= 0; i--) {

if(!isxdigit(hex[i])) // check if character if hexadecimal digit

return -1;

dec += multiplier \* (isdigit(hex[i]) ? (hex[i] - '0') : (toupper(hex[i]) - 'A' + 10));

multiplier \*= 16;

}

return dec;

}

## shell.h

/\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*

\* \*

\* Sogang University \*

\* Department of Computer Science and Engineering \*

\* \*

\* Subject name: System Programming \*

\* Project title: [2] SIC/XE Machine - The Assembler \*

\* \*

\* Author: Inho Kim \*

\* Student ID: 20161577 \*

\* \*

\* File name: shell.h \*

\* File description: Header file for shell related tasks. \*

\* \*

\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*/

void helpCMD(); // COMMAND: help

void dirCMD(); // COMMAND: dir

void quitCMD(); // COMMAND: quit

void histCMD(); // COMMAND: history

void typeCMD(INPUT\_CMD); // COMMAND: type

void histAdd(char\*); // function to add input command into history

void histFree(); // function to free history linked list

## shell.c

/\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*

\* \*

\* Sogang University \*

\* Department of Computer Science and Engineering \*

\* \*

\* Subject name: System Programming \*

\* Project title: [2] SIC/XE Machine - The Assembler \*

\* \*

\* Author: Inho Kim \*

\* Student ID: 20161577 \*

\* \*

\* File name: shell.c \*

\* File description: Tasks processed in shell environment. \*

\* \*

\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*/

#include "20161577.h"

#include "shell.h"

HIST\_NODE\* histHead = NULL; // head of history linked list

void helpCMD() {

printf("h[elp]\n"

"d[ir]\n"

"q[uit]\n"

"hi[story]\n"

"du[mp] [start, end]\n"

"e[dit] address, value\n"

"f[ill] start, end, value\n"

"reset\n"

"opcode mnemonic\n"

"opcodelist\n"

"assemble filename\n"

"type filename\n"

"symbol\n");

}

void dirCMD() {

DIR\* dir = opendir("."); // current directory

char\* entStr;

char path[258] = "./"; // entry path string

ENTRY\* ent; // entry

STBUF buf; // stat

if(!dir) {

puts("ERROR opening directory...");

return;

}

ent = readdir(dir); // read entry

while(ent) {

path[2] = '\0'; // clear path string

entStr = ent->d\_name; // entry name

stat(strcat(path, entStr), &buf);

printf("%-s", entStr); // print entry name

if(S\_ISDIR(buf.st\_mode)) // check for directory

printf("/");

else if(buf.st\_mode & S\_IXUSR) // check for exec file

printf("\*");

ent = readdir(dir); // read next entry

if(ent)

puts("");

}

closedir(dir);

puts("");

}

void quitCMD() {

puts("Exiting SIC...");

histFree(); // free history linked list

hashFree(); // free hash table

symTableFree(); // free SYMTAB

parseListFree(); // free ASM parse list

objListFree(); // free object code list

modListFree(); // free modification record list

exit(0);

}

void histCMD() {

HIST\_NODE\* cur = histHead;

int cnt = 1;

while(cur) {

printf("%-3d ", cnt++);

puts(cur->str); // print command in history

cur = cur->next;

}

}

void typeCMD(INPUT\_CMD ipcmd) {

FILE\* fp = fopen(ipcmd.arg[0], "r");

char c;

if(!fp) {

puts("ERROR: File not found.");

return;

}

while((c = fgetc(fp)) != EOF)

putchar(c);

if(fclose(fp))

puts("WARNING: Error closing file.");

}

void histAdd(char\* str) {

HIST\_NODE\* cur = histHead;

HIST\_NODE\* newHist = (HIST\_NODE\*) malloc(sizeof(HIST\_NODE));

strcpy(newHist->str, str);

newHist->next = NULL;

if(!histHead) { // if history linked list is empty

histHead = newHist;

return;

}

while(cur->next)

cur = cur->next;

cur->next = newHist;

}

void histFree() {

HIST\_NODE\* cur = histHead;

HIST\_NODE\* nex;

while(cur) {

nex = cur->next;

free(cur);

cur = nex;

}

histHead = NULL;

}

## memory.h

/\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*

\* \*

\* Sogang University \*

\* Department of Computer Science and Engineering \*

\* \*

\* Subject name: System Programming \*

\* Project title: [2] SIC/XE Machine - The Assembler \*

\* \*

\* Author: Inho Kim \*

\* Student ID: 20161577 \*

\* \*

\* File name: memory.h \*

\* File description: Header file for memory related tasks. \*

\* \*

\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*/

short mem[MEM\_SIZE]; // virtual memory (1MB)

void dumpCMD(INPUT\_CMD); // COMMAND: dump

void editCMD(INPUT\_CMD); // COMMAND: edit

void fillCMD(INPUT\_CMD); // COMMAND: fill

void resetCMD(); // COMMAND: reset

## memory.c

/\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*

\* \*

\* Sogang University \*

\* Department of Computer Science and Engineering \*

\* \*

\* Subject name: System Programming \*

\* Project title: [2] SIC/XE Machine - The Assembler \*

\* \*

\* Author: Inho Kim \*

\* Student ID: 20161577 \*

\* \*

\* File name: memory.c \*

\* File description: Tasks related to virtual memory handing. \*

\* \*

\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*/

#include "20161577.h"

#include "memory.h"

void dumpCMD(INPUT\_CMD ipcmd) {

static int start = 0;

int end, i, j;

if(start >= MEM\_SIZE)

start = 0; // if start address exceeded memory limit, reset to 0x00000

end = start + 159; // set initial end value

if(ipcmd.argCnt) { // if there was argument entered in command

start = hexToDec(ipcmd.arg[0]);

end = start + 159;

if(ipcmd.argCnt == 2) // if there was 2 arguments

end = hexToDec(ipcmd.arg[1]);

}

if(end >= MEM\_SIZE)

end = MEM\_SIZE - 1; // if end address exceed memory limit, set to 0xFFFFF

// start loop from the beginning of each 16 Bytes

// finish loop till end of each 16 Bytes

for(i = start / 16 \* 16; i < (end / 16 + 1) \* 16; i++) {

if(!(i % 16))

printf("%05X ", i); // print address for beginning of each line

if(i < start || i > end)

printf(" "); // do NOT dump memory outside print range

else

printf("%02X ", mem[i]); // dump memory in hexadecimal value

if(!((i + 1) % 16)) { // line finished

printf("; ");

// print content converted into ASCII character

for(j = i - 15; j <= i; j++)

printf("%c", ((j >= start && j <= end) && mem[j] >= 32 && mem[j] <= 126) ? mem[j] : '.');

puts("");

}

}

start = end + 1; // remember last print address

}

void editCMD(INPUT\_CMD ipcmd) {

int add, val;

add = hexToDec(ipcmd.arg[0]); // address to edit

val = hexToDec(ipcmd.arg[1]); // replace value

mem[add] = val;

}

void fillCMD(INPUT\_CMD ipcmd) {

int i, start, end, val;

start = hexToDec(ipcmd.arg[0]); // fill start address

end = hexToDec(ipcmd.arg[1]); // fill end address

val = hexToDec(ipcmd.arg[2]); // fill value

for(i = start; i <= end; i++)

mem[i] = val;

}

void resetCMD() {

int i;

for(i = 0; i < MEM\_SIZE; i++)

mem[i] = 0; // set memory to 0x00000

}

## hash.h

/\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*

\* \*

\* Sogang University \*

\* Department of Computer Science and Engineering \*

\* \*

\* Subject name: System Programming \*

\* Project title: [1] SIC/XE Machine - The Assembler \*

\* \*

\* Author: Inho Kim \*

\* Student ID: 20161577 \*

\* \*

\* File name: hash.h \*

\* File description: Header file for hash table related tasks. \*

\* \*

\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*/

#define OPCODE\_LEN 7

HASH\_ENTRY\* hashTable[HASH\_SIZE]; // hash table pointer array

void opCMD(INPUT\_CMD); // COMMAND: opcode

void oplistCMD(); // COMMAND: opcodelist

void hashCreate(); // function to create hash table

void checkOperandCnt(HASH\_ENTRY\*);

void hashAddBucket(int, HASH\_ENTRY\*); // function to add bucket to hash table

int hashFunction(char\*); // function that returns hash function

## hash.c

/\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*

\* \*

\* Sogang University \*

\* Department of Computer Science and Engineering \*

\* \*

\* Subject name: System Programming \*

\* Project title: [2] SIC/XE Machine - The Assembler \*

\* \*

\* Author: Inho Kim \*

\* Student ID: 20161577 \*

\* \*

\* File name: hash.c \*

\* File description: Tasks handling opcode and hash table. \*

\* \*

\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*/

#include "20161577.h"

#include "hash.h"

// Format 2 instruction but 1 operand

char exceptionFmt2[][OPCODE\_LEN] = {

"CLEAR",

"SVC",

"TIXR"

};

// Format 3 instruction but NO operand

char exceptionFmt3[][OPCODE\_LEN] = {

"RSUB"

};

void opCMD(INPUT\_CMD ipcmd) {

HASH\_ENTRY\* bucket = bucketSearch(ipcmd.arg[0]);

if(bucket) // target found!

printf("opcode is %s\n", bucket->code);

else { // target NOT found..

puts("ERROR: mnemonic not found.");

puts("Type \"opcodelist\" for list of available opcodes.");

}

}

void oplistCMD() {

HASH\_ENTRY\* bucket;

int i;

for(i = 0; i < HASH\_SIZE; i++) {

printf("%2d : ", i); // print table index

bucket = hashTable[i]; // get front bucket

while(bucket) {

printf("[%s,%s]", bucket->inst, bucket->code);

if((bucket = bucket->next)) // if there exists next bucket

printf(" -> ");

}

puts("");

}

}

void hashCreate() {

FILE\* fp = fopen("opcode.txt", "r"); // open file

char cd[3], ins[10], md[4];

HASH\_ENTRY\* bucket;

if(!fp) {

puts("ERROR: Unable to load \"opcode.txt\".");

return;

}

while(fscanf(fp, "%s %s %s", cd, ins, md) == 3) {

bucket = (HASH\_ENTRY\*) malloc(sizeof(HASH\_ENTRY));

strcpy(bucket->code, cd);

strcpy(bucket->inst, ins);

bucket->codeVal = hexToDec(cd);

bucket->format = md[0] - '1';

checkOperandCnt(bucket);

bucket->next = NULL;

hashAddBucket(hashFunction(bucket->inst), bucket);

}

if(fclose(fp)) {

puts("WARNING: Error closing \"opcode.txt\".");

return;

}

}

void checkOperandCnt(HASH\_ENTRY\* bucket) {

int i;

switch(bucket->format) {

case f1: // Format 1 instruction

bucket->operandCnt = 0;

break;

case f2: // Format 2 instruction

bucket->operandCnt = 2; // Default is 2 operands

for(i = 0; i < 3; i++) // Exception check (CLEAR, SVC, TIXR)

if(!strcmp(bucket->inst, exceptionFmt2[i])) {

bucket->operandCnt = 1;

break;

}

break;

case f34: // Format 3 or 4 instruction

bucket->operandCnt = 1; // Default is 1 operand

for(i = 0; i < 1; i++) // Exception check (RSUB)

if(!strcmp(bucket->inst, exceptionFmt3[i])) {

bucket->operandCnt = 0;

break;

}

break;

}

}

int hashFunction(char\* inst) {

return abs( (int) inst[0] \* 2 + abs(inst[0] + inst[1] + inst[2]) ) % HASH\_SIZE;

}

void hashAddBucket(int hash, HASH\_ENTRY\* bucket) {

HASH\_ENTRY\* cur = hashTable[hash];

if(!cur) { // if empty hash table index

hashTable[hash] = bucket;

return;

}

while(cur->next)

cur = cur->next; // go to the end of list

cur->next = bucket;

}

void hashFree() {

HASH\_ENTRY\* cur;

HASH\_ENTRY\* nex;

int i;

for(i = 0; i < HASH\_SIZE; i++) {

cur = hashTable[i]; // front bucket in each hash table index

while(cur) {

nex = cur->next;

free(cur);

cur = nex;

}

hashTable[i] = NULL; // reset pointer to NULL

}

}

HASH\_ENTRY\* bucketSearch(char\* inst) {

HASH\_ENTRY\* bucket;

char tmp[ASM\_LEN] = {'\0'};

strcpy(tmp, inst);

bucket = hashTable[hashFunction(tmp)]; // get front bucket from hash function

while(bucket && strcmp(bucket->inst, inst)) // search till match or end

bucket = bucket->next;

return bucket;

}

## assembler.h

/\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*

\* \*

\* Sogang University \*

\* Department of Computer Science and Engineering \*

\* \*

\* Subject name: System Programming \*

\* Project title: [2] SIC/XE Machine - The Assembler \*

\* \*

\* Author: Inho Kim \*

\* Student ID: 20161577 \*

\* \*

\* File name: assembler.h \*

\* File description: Header file for assembler tasks. \*

\* \*

\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*/

bool assembleCMD(INPUT\_CMD); // COMMAND: assemble

ASM\_SRC\* parseASM(char\*); // parse ASM source

bool assemblerPass1(FILE\*, int\*, int\*); // Pass1 of SIC/XE Assembler

bool assemblerPass2(FILE\*, FILE\*, int, int); // Pass2 of SIC/XE Assembler

void printLST(FILE\*, ASM\_SRC\*, int, bool, bool); // create content into .lst file

void printOBJ(FILE\*, ASM\_SRC\*, int, int); // create content into .obj file

void printOBJList(FILE\*); // print object code list

void printModList(FILE\*); // print modification record list

void addOBJList(ASM\_SRC\*); // add node to object code list

void addModList(int, int); // add node to modification record list

void setError(ASM\_SRC\*, ASM\_ERROR); // mark error in ASM source

void printASMError(ASM\_SRC\*); // print detail about error

bool isRegister(char); // check whether operand is a register name

void symbolCMD(); // COMMAND: symbol

void symTableAdd(char\*, int); // add node to SYMTAB

SYMBOL\_ENTRY\* symTableSearch(char\*); // search for node in SYMTAB

## assembler.c

/\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*

\* \*

\* Sogang University \*

\* Department of Computer Science and Engineering \*

\* \*

\* Subject name: System Programming \*

\* Project title: [2] SIC/XE Machine - The Assembler \*

\* \*

\* Author: Inho Kim \*

\* Student ID: 20161577 \*

\* \*

\* File name: assembler.c \*

\* File description: Tasks for SIC/XE Assembler. \*

\* \*

\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*/

#include "20161577.h"

#include "assembler.h"

SYMBOL\_ENTRY\* symTable = NULL; // symbol table

ASM\_SRC\* parseList = NULL; // ASM source parse list

OBJ\_CODE\* objList = NULL; // object code listt

MOD\_RECORD\* modList = NULL; // modification record list

// ASM directives list

char directives[8][6] = {

"NOTDR",

"START",

"END",

"BASE",

"BYTE",

"WORD",

"RESB",

"RESW"

};

// ASM register list

char registers[7] = {

'A', 'X', 'L', 'B', 'S', 'T', 'F'

};

// COMMAND: assemble

bool assembleCMD(INPUT\_CMD ipcmd) {

FILE \*srcFile, \*lstFile, \*objFile;

char lstName[CMD\_LEN] = {'\0'}, objName[CMD\_LEN] = {'\0'};

int maxSrcLen = 0, progLen = 0;

// initialize

symTableFree();

parseListFree();

objListFree();

modListFree();

lstFile = objFile = NULL;

srcFile = fopen(ipcmd.arg[0], "r"); // open source file

if(!srcFile) {

puts("ERROR: File not found.");

return false;

}

// create file name for .lst and .obj files

strncpy(lstName, ipcmd.arg[0], strlen(ipcmd.arg[0]) - 3);

strncpy(objName, ipcmd.arg[0], strlen(ipcmd.arg[0]) - 3);

strcat(lstName, "lst");

strcat(objName, "obj");

lstFile = fopen(lstName, "w");

objFile = fopen(objName, "w");

if(!lstFile || !objFile) {

puts("ERROR: Problem while creating .list and .obj files.");

fclose(srcFile);

if(lstFile) fclose(lstFile);

if(objFile) fclose(objFile);

return false;

}

// both Pass1 and Pass2 have to be completed

if(!assemblerPass1(srcFile, &maxSrcLen, &progLen) || !assemblerPass2(lstFile, objFile, maxSrcLen, progLen)) {

puts(".lst file and .obj file were not created.");

remove(lstName);

remove(objName);

fclose(srcFile);

if(lstFile) fclose(lstFile);

if(objFile) fclose(objFile);

return false;

}

else

printf("\toutput file : [%s], [%s]\n", lstName, objName);

if(fclose(srcFile))

puts("WARNING: Error closing file.");

if(fclose(lstFile))

puts("WARNING: Error closing file.");

if(fclose(objFile))

puts("WARNING: Error closing file.");

return true;

}

// Pass1 of Assembler

bool assemblerPass1(FILE\* srcFile, int\* maxSrcLen, int\* progLen) {

ASM\_SRC \*curParse = NULL, \*prevParse = NULL;

char source[ASM\_LEN] = {'\0'};

int location = -1;

int lineNum = 5;

bool errorFlag = false;

int mult;

// process ASM source file line by line

while(fgets(source, ASM\_LEN, srcFile) != NULL) {

// remove '\n' at the end of string

if(source[strlen(source) - 1] == '\n')

source[strlen(source) - 1] = '\0';

curParse = parseASM(source); // get parsed line

// START directive found

if(curParse->direcName == START) {

// if START directive appears for the second time

if(location != -1) {

puts("ERROR: START directive appeared twice.");

errorFlag = true;

break;

}

// initialize LOCCTR

location = hexToDec(curParse->operand[0]);

curParse->location = location;

parseList = curParse;

prevParse = NULL;

}

// no START directive, set start address as 0

if(location == -1) {

location = 0;

parseList = curParse;

prevParse = NULL;

curParse->location = location;

}

else if(prevParse)

prevParse->next = curParse; // link node to list

curParse->lineNum = lineNum;

curParse->location = location;

// if current line is NOT a comment

if(curParse->type != COMMENT) {

// remember longest ASM source line

if(\*maxSrcLen < strlen(source))

\*maxSrcLen = strlen(source);

// if current line has symbol in label field

if(curParse->hasLabel && curParse->direcName != START) {

// if symbol is already in SYMTAB

if(symTableSearch(curParse->label)) {

setError(curParse, SYMBOL);

errorFlag = true;

break;

}

// new symbol found

else

symTableAdd(curParse->label, location);

}

switch(curParse->type) {

case ERROR: // Error

errorFlag = true;

break;

case INST: // Instruction

curParse->byteSize = curParse->format;

break;

case PSEUDO: // Pseudo-instruction

// directives

switch(curParse->direcName) {

case START:

break;

case END:

case BASE:

break;

case BYTE:

// calculate byte size according to operand

switch(curParse->operand[0][0]) {

case 'X':

curParse->byteSize = (strlen(curParse->operand[0]) - 3 + 1) / 2;

break;

case 'C':

curParse->byteSize = strlen(curParse->operand[0]) - 3;

break;

default:

mult = INC\_BYTE;

for(curParse->byteSize = 1;

mult <= atoi(curParse->operand[0]);

mult \*= INC\_BYTE)

curParse->byteSize++;

break;

}

break;

case WORD:

curParse->byteSize = 3;

break;

case RESB:

curParse->byteSize = atoi(curParse->operand[0]);

break;

case RESW:

curParse->byteSize = atoi(curParse->operand[0]) \* 3;

break;

default:

break;

}

break;

case COMMENT:

break;

default:

break;

}

}

if(errorFlag)

break;

location += curParse->byteSize; // calculate next location counter

lineNum += 5; // increase line number

prevParse = curParse; // previous node

// condition to end parsing

if(!strcmp(curParse->inst, "END"))

break;

}

\*progLen = location; // save the program length for .obj file

if(errorFlag) {

printASMError(curParse);

symTableFree();

return false;

}

return true;

}

// Pass2 of Assembler

bool assemblerPass2(FILE\* lstFile, FILE\* objFile, int maxSrcLen, int progLen) {

ASM\_SRC \*curParse = NULL;

SYMBOL\_ENTRY \*curSymbol = NULL;

int pcReg, baseReg;

bool errorFlag = false;

bool locFlag, objFlag;

bool nFlag, iFlag, xFlag, bFlag, pFlag, eFlag;

int i, firstExec;

// initialize

curParse = parseList;

baseReg = 0;

pcReg = parseList->location;

while(curParse) {

// initialize

locFlag = objFlag = true;

curSymbol = NULL;

// change PC register to next instruction

if(curParse->next)

pcReg = curParse->next->location;

// if no START directive at start, choose a blank program name

if(curParse == parseList && curParse->direcName != START) {

printf("H%6s%06X%06X\n", " ", 0, progLen);

firstExec = curParse->location;

}

switch(curParse->type) {

case INST: // instruction

// initialize

nFlag = iFlag = true;

xFlag = bFlag = pFlag = eFlag = false;

curParse->objCode = bucketSearch(curParse->inst + (curParse->inst[0] == '+' ? 1 : 0) )->codeVal;

switch(curParse->format) {

case format1:

break;

case format2:

// determine object code of each register

for(i = 0; i < 2; i++) {

curParse->objCode \*= 16;

switch(curParse->operand[i][0]) {

case 'A':

curParse->objCode += 0;

break;

case 'X':

curParse->objCode += 1;

break;

case 'L':

curParse->objCode += 2;

break;

case 'B':

curParse->objCode += 3;

break;

case 'S':

curParse->objCode += 4;

break;

case 'T':

curParse->objCode += 5;

break;

case 'F':

curParse->objCode += 6;

break;

default:

break;

}

}

break;

case format3:

if(!curParse->operandCnt) {

curParse->objCode += 3;

curParse->objCode \*= INC\_BYTE \* INC\_BYTE;

break;

}

// indexing mode

if(curParse->indexing)

xFlag = true;

// immediate addressing

if(curParse->operand[0][0] == '#')

nFlag = false;

// indirect addressing

else if(curParse->operand[0][0] == '@')

iFlag = false;

// if there is a symbol in operand

if((curSymbol = symTableSearch(curParse->operand[0] + ((!nFlag || !iFlag) ? 1 : 0)))) {

// if pc relative addressing is possible

if(curSymbol->address - pcReg + MAX12DISP / 2 >= 0) {

curParse->objCode += (!nFlag ? 1 : (!iFlag ? 2 : 3));

curParse->objCode \*= INC\_HBYTE;

curParse->objCode += (xFlag ? 10 : 2);

curParse->objCode \*= INC\_BYTE \* INC\_HBYTE;

curParse->objCode += (curSymbol->address - pcReg) & 0xFFF;

}

// if base relative addressing is possible

else if(curSymbol->address - baseReg < MAX12DISP) {

curParse->objCode += (!nFlag ? 1 : (!iFlag ? 2 : 3));

curParse->objCode \*= INC\_HBYTE;

curParse->objCode += (xFlag ? 12 : 4);

curParse->objCode \*= INC\_BYTE \* INC\_HBYTE;

curParse->objCode += (curSymbol->address - baseReg) & 0xFFF;

}

// approach as SIC instruction

else if(curSymbol->address < MAX15ADDR){

curParse->objCode \*= INC\_HBYTE;

curParse->objCode += (xFlag ? 8 : 0);

curParse->objCode \*= INC\_BYTE \* INC\_HBYTE;

curParse->objCode += curSymbol->address;

}

// NOT able to make reference

else {

setError(curParse, INSTRUCTION);

return false;

}

}

// if there is a value in operand

else if(pcReg - atoi(curParse->operand[0] + 1) + MAX12DISP / 2 >= 0) {

curParse->objCode += (!nFlag ? 1 : 2);

curParse->objCode \*= INC\_HBYTE;

curParse->objCode += 0;

curParse->objCode \*= INC\_BYTE \* INC\_HBYTE;

curParse->objCode += atoi(curParse->operand[0] + 1);

}

break;

case format4:

eFlag = true;

// search for symbol

curSymbol = symTableSearch(curParse->operand[0] + (!isalnum(curParse->operand[0][0]) ? 1 : 0));

// if there is a symbol in operand

if(curSymbol) {

curParse->objCode += 3;

curParse->objCode \*= INC\_HBYTE;

curParse->objCode += 1;

curParse->objCode \*= INC\_BYTE \* INC\_BYTE \* INC\_HBYTE;

curParse->objCode += curSymbol->address & 0xFFFFF;

}

// if there is a numeric value in operand

else {

curParse->objCode += 1;

curParse->objCode \*= INC\_HBYTE;

curParse->objCode += 1;

curParse->objCode \*= INC\_BYTE \* INC\_BYTE \* INC\_HBYTE;

curParse->objCode += atoi(curParse->operand[0] + 1) & 0xFFFFF;

}

break;

default:

break;

}

break;

case PSEUDO: // pseudo-instruction

switch(curParse->direcName) {

case START:

// initialize base register and remember first location

objFlag = false;

baseReg = curParse->location;

firstExec = curParse->location;

printOBJ(objFile, curParse, progLen, firstExec);

break;

case BASE:

// set base register

locFlag = objFlag = false;

baseReg = symTableSearch(curParse->operand[0])->address;

break;

case END:

locFlag = objFlag = false;

printOBJ(objFile, curParse, progLen, firstExec);

break;

case BYTE:

switch(curParse->operand[0][0]) {

case 'C': // a character constant

for(i = 2; curParse->operand[0][i] != '\''; i++) {

curParse->objCode \*= INC\_BYTE;

curParse->objCode += curParse->operand[0][i];

}

break;

case 'X': // a hexadecimal constant

curParse->objCode += strtol(curParse->operand[0] + 2, NULL, 16);

break;

default:

curParse->objCode = atoi(curParse->operand[0]);

break;

}

break;

case WORD:

curParse->objCode = atoi(curParse->operand[0]);

break;

case RESB:

case RESW:

objFlag = false;

break;

default:

break;

}

break;

case COMMENT:

locFlag = objFlag = false;

default:

break;

}

// if it has object code, print into .obj file

if(curParse->objCode)

printOBJ(objFile, curParse, progLen, firstExec);

// print into .lst file

printLST(lstFile, curParse, maxSrcLen, locFlag, objFlag);

curParse = curParse->next;

}

if(errorFlag) {

printASMError(curParse);

symTableFree();

return false;

}

return true;

}

void printLST(FILE\* lstFile, ASM\_SRC\* parsedASM, int maxSrcLen, bool printLOC, bool printOBJ) {

int i;

// print line number

fprintf(lstFile, "%4d\t", parsedASM->lineNum);

// print location if flag is set

(printLOC ? fprintf(lstFile, "%04X\t", parsedASM->location) : fprintf(lstFile, "\t"));

// print the source line with blanks filled to match the max length

for(i = 0; parsedASM->source[i]; i++)

fprintf(lstFile, "%c", parsedASM->source[i]);

for(;i < maxSrcLen; i++)

fprintf(lstFile, " ");

// print object code according to byte size

if(printOBJ) {

fprintf(lstFile, "\t\t");

switch(parsedASM->byteSize) {

case 1:

fprintf(lstFile, "%02X", parsedASM->objCode);

break;

case 2:

fprintf(lstFile, "%04X", parsedASM->objCode);

break;

case 3:

fprintf(lstFile, "%06X", parsedASM->objCode);

break;

case 4:

fprintf(lstFile, "%08X", parsedASM->objCode);

break;

default:

break;

}

}

fprintf(lstFile, "\n");

}

void printOBJ(FILE\* objFile, ASM\_SRC\* parsedASM, int progLen, int firstExec) {

static int lineLen = 0; // remember line lenght

// if START directive

if(parsedASM->direcName == START) {

fprintf(objFile, "H%-6s%06X%06X\n", parsedASM->label, parsedASM->location, progLen);

return;

}

// if END directive

if(parsedASM->direcName == END) {

printModList(objFile);

fprintf(objFile, "E%06X\n", firstExec);

return;

}

// if format 4 instruction with simple addressing add to modification record list

if(parsedASM->format == format4 && symTableSearch(parsedASM->operand[0] + (!isalnum(parsedASM->operand[0][0]) ? 1 : 0)))

addModList(parsedASM->location - firstExec + 1, 5);

// if line lenght exceeds 0x1E

if(lineLen + parsedASM->byteSize > 0x1E) {

fprintf(objFile, "T%06X%02X", objList ? objList->location : parsedASM->location, lineLen);

printOBJList(objFile);

objListFree();

lineLen = 0;

}

addOBJList(parsedASM);

lineLen += parsedASM->byteSize;

// if node was a variable, cut line

if(parsedASM->direcName == BYTE || parsedASM->direcName == WORD) {

fprintf(objFile, "T%06X%02X", objList ? objList->location : parsedASM->location, lineLen);

printOBJList(objFile);

objListFree();

lineLen = 0;

}

}

void printOBJList(FILE\* objFile) {

OBJ\_CODE\* cur = objList;

// print depending on byte size

while(cur) {

switch(cur->byteSize) {

case 1:

fprintf(objFile, "%02X", cur->objCode);

break;

case 2:

fprintf(objFile, "%04X", cur->objCode);

break;

case 3:

fprintf(objFile, "%06X", cur->objCode);

break;

case 4:

fprintf(objFile, "%08X", cur->objCode);

break;

default:

break;

}

cur = cur->next;

}

fprintf(objFile, "\n");

}

void printModList(FILE\* objFile) {

MOD\_RECORD\* cur = modList;

// print all modification records

while(cur) {

fprintf(objFile, "M%06X%02d\n", cur->location, cur->lenHB);

cur = cur->next;

}

}

void addOBJList(ASM\_SRC\* parsedASM) {

OBJ\_CODE \*newOBJ, \*cur = objList;

// create new node of object code list

newOBJ = (OBJ\_CODE\*) malloc(sizeof(OBJ\_CODE));

newOBJ->objCode = parsedASM->objCode;

newOBJ->location = parsedASM->location;

newOBJ->byteSize = parsedASM->byteSize;

newOBJ->next = NULL;

if(!objList) {

objList = newOBJ;

return;

}

while(cur->next)

cur = cur->next;

cur->next = newOBJ;

}

void addModList(int location, int length) {

MOD\_RECORD \*newRec, \*cur = modList;

// create new node of modification record list

newRec = (MOD\_RECORD\*) malloc(sizeof(MOD\_RECORD));

newRec->lenHB = length;

newRec->location = location;

newRec->next = NULL;

if(!modList) {

modList = newRec;

return;

}

while(cur->next)

cur = cur->next;

cur->next = newRec;

}

// free ASM source parse list

void parseListFree() {

ASM\_SRC \*cur, \*next;

cur = parseList;

while(cur) {

next = cur->next;

free(cur);

cur = next;

}

parseList = NULL;

}

// free object code list

void objListFree() {

OBJ\_CODE \*cur, \*next;

cur = objList;

while(cur) {

next = cur->next;

free(cur);

cur = next;

}

objList = NULL;

}

// free modification record list

void modListFree() {

MOD\_RECORD \*cur, \*next;

cur = modList;

while(cur) {

next = cur->next;

free(cur);

cur = next;

}

modList = NULL;

}

// print error detail of ASM source

void printASMError(ASM\_SRC\* parsedASM) {

printf("ERROR: Invalid assembly source.\n");

printf("[Line %d] Error in ", parsedASM->lineNum);

switch(parsedASM->errorCode) {

case SYMBOL:

puts("symbol field:");

break;

case INSTRUCTION:

puts("instruction field:");

break;

case OPERAND:

puts("operand field:");

break;

default:

break;

}

puts(parsedASM->source);

}

// COMMADN: symbol

void symbolCMD() {

SYMBOL\_ENTRY\* cur = symTable;

if(!symTable) {

puts("Symbol table is empty.");

return;

}

while(cur) {

printf("\t%s\t%04X\n", cur->symbol, cur->address);

cur = cur->next;

}

}

ASM\_SRC\* parseASM(char\* source) {

char delim[] = " \t\n";

char tmp[ASM\_LEN] = {'\0'};

char \*tok = NULL;

int i, j;

ASM\_SRC\* parseResult = NULL;

HASH\_ENTRY\* bucket = NULL;

// initialize parse structure

parseResult = (ASM\_SRC\*) malloc(sizeof(ASM\_SRC));

strcpy(parseResult->source, source);

memset(parseResult->label, '\0', ASM\_LEN);

memset(parseResult->inst, '\0', ASM\_LEN);

memset(parseResult->operand[0], '\0', ASM\_LEN);

memset(parseResult->operand[1], '\0', ASM\_LEN);

parseResult->hasLabel = false;

parseResult->indexing = false;

parseResult->operandCnt = 0;

parseResult->byteSize = 0;

parseResult->objCode = 0;

parseResult->type = INST;

parseResult->direcName = NOTDR;

parseResult->errorCode = OK;

parseResult->next = NULL;

// add space front and back of commas

j = 0;

for(i = 0; source[i]; i++) {

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

strcpy(tmp + j, " , ");

j += 3;

}

else

tmp[j++] = source[i];

}

tok = strtok(tmp, delim);

if(tok[0] == '.') { // comment found

parseResult->type = COMMENT;

parseResult->format = NONE;

return parseResult;

}

else if(!strcmp(tok, ",")) { // comma should not be present

setError(parseResult, INSTRUCTION);

return parseResult;

}

// look for directives

for(i = 1; i < 8; i++)

if(!strcmp(tok, directives[i])) { // pseudo instruction found

parseResult->type = PSEUDO;

parseResult->format = NONE;

parseResult->direcName = i;

strcpy(parseResult->inst, tok);

break;

}

if(parseResult->type != PSEUDO) { // if not a pseudo instruction

bucket = bucketSearch(tok + (tok[0] == '+' ? 1 : 0) );

// if first field is a label

if(!bucket) {

if(symTableSearch(tok)) { // symbol already exists

setError(parseResult, SYMBOL);

return parseResult;

}

else { // new symbol found, add to SYMTAB and move on

parseResult->hasLabel = true;

strcpy(parseResult->label, tok);

tok = strtok(NULL, delim);

// missing instruction

if(!tok) {

setError(parseResult, INSTRUCTION);

return parseResult;

}

bucket = bucketSearch(tok + (tok[0] == '+' ? 1 : 0) );

if(!strcmp(tok, ",")) { // comma should not be present

setError(parseResult, INSTRUCTION);

return parseResult;

}

}

}

// look for directives

for(i = 1; i < 8; i++)

if(!strcmp(tok, directives[i])) { // pseudo instruction found

parseResult->type = PSEUDO;

strcpy(parseResult->inst, tok);

parseResult->format = NONE;

parseResult->direcName = i;

break;

}

// if field is an instruction

if(bucket) {

parseResult->type = INST;

parseResult->format = bucket->format + 1; // get format from hash table

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

// not a format 3/4 instruction

if(bucket->format != f34) {

setError(parseResult, INSTRUCTION);

return parseResult;

}

// found a format 4 instruction

parseResult->format = format4;

}

strcpy(parseResult->inst, tok);

}

// invalid instruction

else if(parseResult->type != PSEUDO){

setError(parseResult, INSTRUCTION);

return parseResult;

}

}

// tokenize the operand field part

tok = strtok(NULL, delim); // first operand

if(bucket && ((bucket->operandCnt && !tok) || (!bucket->operandCnt && tok))) {

setError(parseResult, OPERAND);

return parseResult;

}

else if(bucket && !bucket->operandCnt)

return parseResult;

if(tok && !strcmp(tok, ",")) { // comma should not be present

setError(parseResult, OPERAND);

return parseResult;

}

strcpy(parseResult->operand[0], tok);

parseResult->operandCnt = 1;

tok = strtok(NULL, delim); // expected a comma if there's second operand

if(tok && strcmp(tok, ",")) { // comma should be present

setError(parseResult, OPERAND);

return parseResult;

}

if(tok) {

tok = strtok(NULL, delim); // second operand

if(!tok || !strcmp(tok, ",")) {

setError(parseResult, OPERAND);

return parseResult;

}

strcpy(parseResult->operand[1], tok);

parseResult->operandCnt = 2;

}

// check operand format for instructions

if(parseResult->type == INST) {

switch(parseResult->format) {

case format2:

// first operand must be a Register

if(strlen(parseResult->operand[0]) != 1 || !isRegister(parseResult->operand[0][0])) {

setError(parseResult, OPERAND);

return parseResult;

}

// second operand for SHIFTL/R must be a number

if(!strncmp(parseResult->inst, "SHIFT", 5)) {

for(i = 0; parseResult->operand[1][i]; i++)

if(!isdigit(parseResult->operand[1][i])) {

setError(parseResult, OPERAND);

return parseResult;

}

}

// second operand must be a Register

else if(bucket->operandCnt != 1){

if(strlen(parseResult->operand[1]) != 1 || !isRegister(parseResult->operand[1][0])) {

setError(parseResult, OPERAND);

return parseResult;

}

}

break;

case format3:

case format4:

// one non-register operand or second operand is not X (indexing)

if((strlen(parseResult->operand[0]) == 1 && isRegister(parseResult->operand[0][0])) || (strlen(parseResult->operand[1]) && strcmp(parseResult->operand[1], "X")) ) {

setError(parseResult, OPERAND);

return parseResult;

}

if(strlen(parseResult->operand[1]))

parseResult->indexing = true;

break;

default:

break;

}

}

return parseResult;

}

// check for register name

bool isRegister(char reg) {

int i;

for(i = 0; i < 7; i++)

if(registers[i] == reg)

return true;

return false;

}

// set information about error in node

void setError(ASM\_SRC\* parsedResult, ASM\_ERROR error) {

parsedResult->type = ERROR;

parsedResult->format = NONE;

parsedResult->errorCode = error;

}

// add new symbol to SYMTAB

void symTableAdd(char\* symbol, int address) {

SYMBOL\_ENTRY\* cur = symTable;

SYMBOL\_ENTRY\* newEntry = (SYMBOL\_ENTRY\*) malloc(sizeof(SYMBOL\_ENTRY));

strcpy(newEntry->symbol, symbol);

newEntry->address = address;

newEntry->next = NULL;

// if SYMTAB is empty

if(!symTable) {

symTable = newEntry;

return;

}

// add in a non ascending order

if(strcmp(symTable->symbol, symbol) < 0) {

newEntry->next = symTable;

symTable = newEntry;

return;

}

// while next symbol is bigger in alphabetical order

while(cur->next && strcmp(cur->next->symbol, symbol) > 0)

cur = cur->next;

newEntry->next = cur->next;

cur->next = newEntry;

}

SYMBOL\_ENTRY\* symTableSearch(char\* symbol) {

SYMBOL\_ENTRY\* cur = symTable;

while(cur) {

// if a match is found

if(!strcmp(symbol, cur->symbol))

return cur;

cur = cur->next;

}

return NULL;

}

// free SYMTAB

void symTableFree() {

SYMBOL\_ENTRY \*cur, \*next;

cur = symTable;

while(cur) {

next = cur->next;

free(cur);

cur = next;

}

symTable = NULL;

}

## Makefile

20161577.out: main.o cmdProc.o shell.o memory.o hash.o assembler.o

gcc -Wall -o 20161577.out main.o cmdProc.o shell.o memory.o hash.o assembler.o -lm

@echo "\n>>> To execute, type ./20161577.out\n"

main.o: 20161577.h 20161577.c

gcc -Wall -c -o main.o 20161577.c -lm

cmdProc.o: 20161577.h cmdProc.c

gcc -Wall -c -o cmdProc.o cmdProc.c -lm

shell.o: 20161577.h shell.c

gcc -Wall -c -o shell.o shell.c -lm

memory.o: 20161577.h memory.c

gcc -Wall -c -o memory.o memory.c -lm

hash.o: 20161577.h hash.c

gcc -Wall -c -o hash.o hash.c -lm

assembler.o: 20161577.h assembler.c

gcc -Wall -c -o assembler.o assembler.c -lm

clean:

-rm \*.o

-rm \*.lst

-rm \*.obj

-rm 20161577.out