과목명: 시스템프로그래밍

담당 교수명: 박운상

<<Assignment 2>>

**서강대학교 컴퓨터공학과**

**20161662 허남규**

Table of Contents

[1. 프로그램 개요 3](#_Toc511156507)

[2. 프로그램 설명 3](#_Toc511156508)

[2.1. Main 함수 3](#_Toc511156509)

[2.1.1.ParsedCommand [parser.h] 및 Interpreter [interpreter.h] 모듈 3](#_Toc511156510)

[2.1.2.활용: Main 함수 주요 기능 설명 3](#_Toc511156511)

[2.1.3.Flowchart 4](#_Toc511156512)

[2.2. Assemble 명령 처리 4](#_Toc511156513)

[2.2.1.Assemble.h, asm\_helper.h 모듈 4](#_Toc511156514)

[2.2.2.Assemble 명령 처리 구조 4](#_Toc511156515)

[2.2.3.Asm\_helper.h 모듈의 활용 6](#_Toc511156516)

[3. 모듈 정의 6](#_Toc511156517)

[3.1. dir [dir.h] 7](#_Toc511156518)

[함수 설명 7](#_Toc511156519)

[3.2. hashtable [hashtable.h] 7](#_Toc511156520)

[3.2.1.함수 설명 7](#_Toc511156521)

[3.3. help [help.h] 8](#_Toc511156522)

[3.3.1.함수 설명 8](#_Toc511156523)

[3.4. history [history.h] 8](#_Toc511156524)

[3.4.1.함수 설명 8](#_Toc511156525)

[3.5. memory [memory.h] 9](#_Toc511156526)

[3.5.1.함수 설명 9](#_Toc511156527)

[3.6. asm\_helper.h 10](#_Toc511156528)

[3.6.1.Function Explanations 10](#_Toc511156529)

[3.7. assemble.h 11](#_Toc511156530)

[3.7.1.Function Explanations 11](#_Toc511156531)

[3.8. file.h 12](#_Toc511156532)

[3.8.1.Function Explanations 12](#_Toc511156533)

[3.9. generic\_dict 12](#_Toc511156534)

[3.9.1.Function Explanations 13](#_Toc511156535)

[3.10. interpreter.h 15](#_Toc511156536)

[3.10.1.Usage 15](#_Toc511156537)

[3.10.2.Function Explanations 15](#_Toc511156538)

[3.11. parser.h 16](#_Toc511156539)

[3.11.1.Function Explanations 16](#_Toc511156540)

[3.12. register.h 16](#_Toc511156541)

[3.12.1.Function Explanations 16](#_Toc511156542)

[3.13. reserved.h 16](#_Toc511156543)

[3.14. symbol.h 17](#_Toc511156544)

[3.14.1.(Some) Function Explanations 17](#_Toc511156545)

[4. 전역 변수 17](#_Toc511156546)

[4.1. struct \_Global G 내 전역 변수 17](#_Toc511156547)

[4.1.1.History \*history 17](#_Toc511156548)

[4.1.2.HashTable table; 17](#_Toc511156549)

[4.1.3.Block \*block; 17](#_Toc511156550)

[4.1.4.ReservedDict \*reserved; 18](#_Toc511156551)

[5. 특이 사항 18](#_Toc511156552)

[5.1. Inline 함수 18](#_Toc511156553)

[5.2. Macro 정의를 이용한 모듈 테스트 18](#_Toc511156554)

[5.3. void 포인터를 활용한 generic 자료 구조 18](#_Toc511156555)

[5.4. 가독성 및 주석 관련 19](#_Toc511156556)

[6. 코드 19](#_Toc511156557)

# 프로그램 개요

이번 프로젝트의 최종 목표는 Linux OS 상에서 C 표준 라이브러리 함수를 기반으로 SIC/XE 가상 머신을 구현하는 것입니다. 저번 프로젝트에서는 명령어 입출력, 메모리 조작 및 출력, 그리고 연산자 mnemonic을 opcode 값으로 변환하는 테이블 등 SIC/XE 가상 머신의 기반이 되는 요소들을 구현했습니다.

이번 프로젝트에서는 그것의 연장선 상에서 SIC/XE 아키텍쳐의 어셈블리 파일을 컴파일해서 lst와 obj 파일을 생성하는 명령어와 그 결과 만들어진 symbol table을 출력하거나 asm, lst, obj 등의 파일을 출력하는 명령어를 구현했습니다.

# 프로그램 설명

## Main 함수

### ParsedCommand [parser.h] 및 Interpreter [interpreter.h] 모듈

구현하는 명령어의 개수가 늘어남에 따라 main 함수가 너무 길어지는 것을 우려하여 이번 프로젝트에서는 사용자의 명령을 명령어 (command operator)와 인자 (arguments)로 따로 파싱해서 저장하는 ParsedCommand 모듈과 함께, 파싱된 명령어를 분석해서 그에 대응하는 함수를 알아서 호출하는 Interpreter 모듈을 제작했습니다..

### 활용: Main 함수 주요 기능 설명

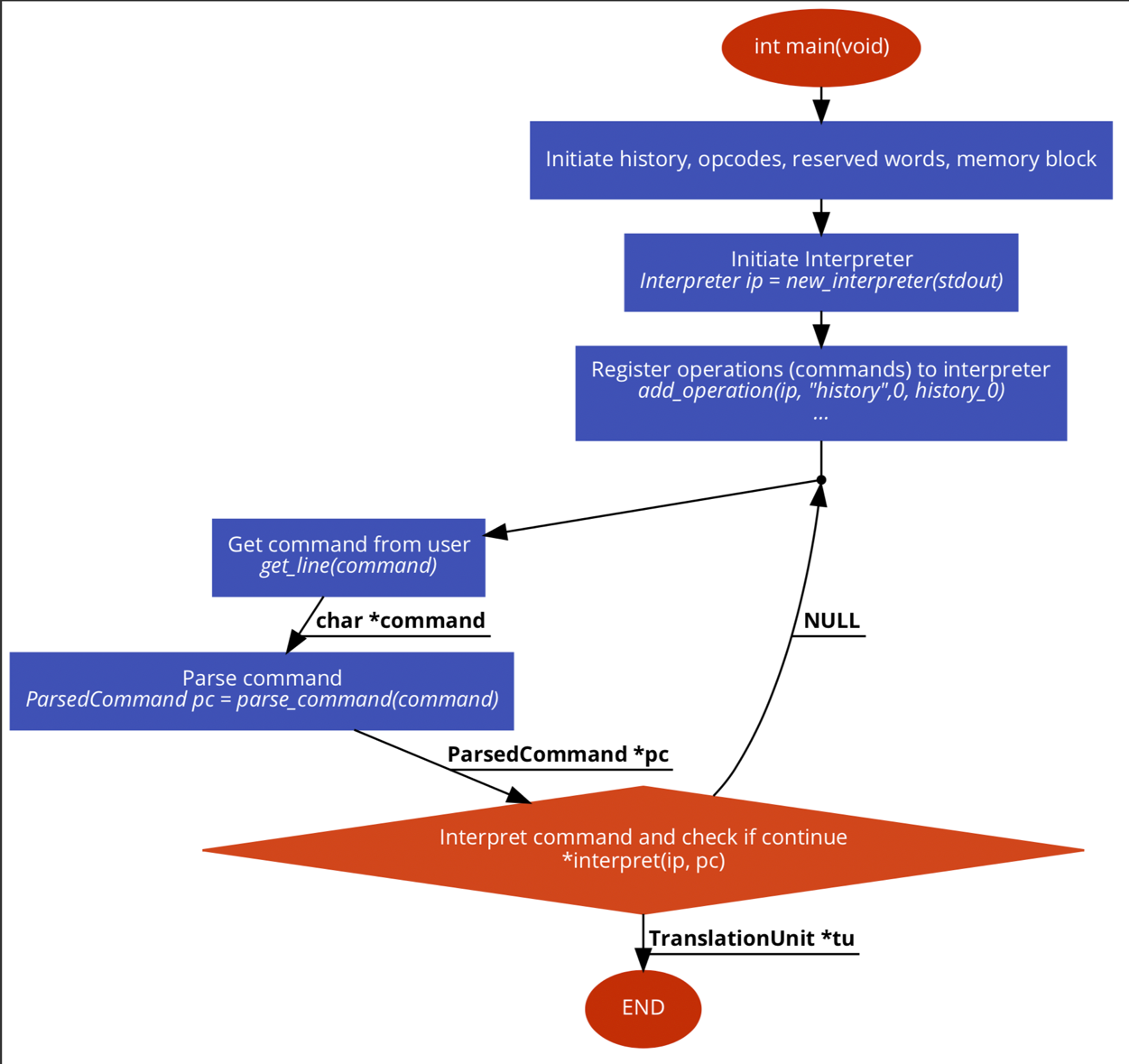
Interpreter 설정: 먼저, Interpreter 구조체를 초기화한 후, <명령어+인자 개수, 호출 함수> 쌍을 add\_operation 함수로 넘겨 interpreter에 추가합니다. 예를 들어

add\_operation((Interpreter\*) ip, “dump”, 2, dump\_function);

위와 같이 add\_operation 함수를 호출할 경우, “dump argument0 argument1”와 같은 사용자 입력을 파싱하여 interpreter에 넘기면 dump\_function함수가 호출됩니다. 이때 사용자가 입력된 인자도 함수에 넘겨집니다.

사용자 입력 받기: While 반복문 내에서 사용자의 입력을 받습니다. 그리고 해당 입력을 인자로 parse\_command 함수를 호추랗여, 입력을 명령어와 인자로 구분하여 저장한 ParsedCommand 구조체를 반환받습니다. 이때 해당 ParsedCommand를 interpreter와 함께 interpret 함수로 넘기면, interpreter가 해당 명령을 처리하게 됩니다.

### Flowchart



## Assemble 명령 처리

### Assemble.h, asm\_helper.h 모듈

Assemble 명령에서 처리해야 하는 케이스 (operator, directive, operation 종류 등)가 상당히 많습니다. 그래서 주요 기능을 처리하는 assemble.h 모듈과 함께, assemble.h 모듈의 implementation (assemble.c)의 readability를 향상시키고 문제를 소단위로 분리하여 코드 오류를 최소화하고자 asm\_helper.h라는 헬퍼 모듈을 따로 만들었습니다.

### Assemble 명령 처리 구조

#### Translation 과정

asm 파일을 처리할 때, 먼저 모든 asm 코드를 모두 해석해서 TranslationUnit이라는 구조체 내에 SicStatement라는 구조체의 linked-list로 저장합니다. 이를 translation 과정이라 부르겠습니다. SIC/XE 구문은 각각 SicStatement에 하나씩 1:1 대응되어 저장됩니다. 이때 SIC/XE 구문은 instruction과 함께 START, END, BASE, BYTE, RESB 등의 directive 줄을 포함합니다.

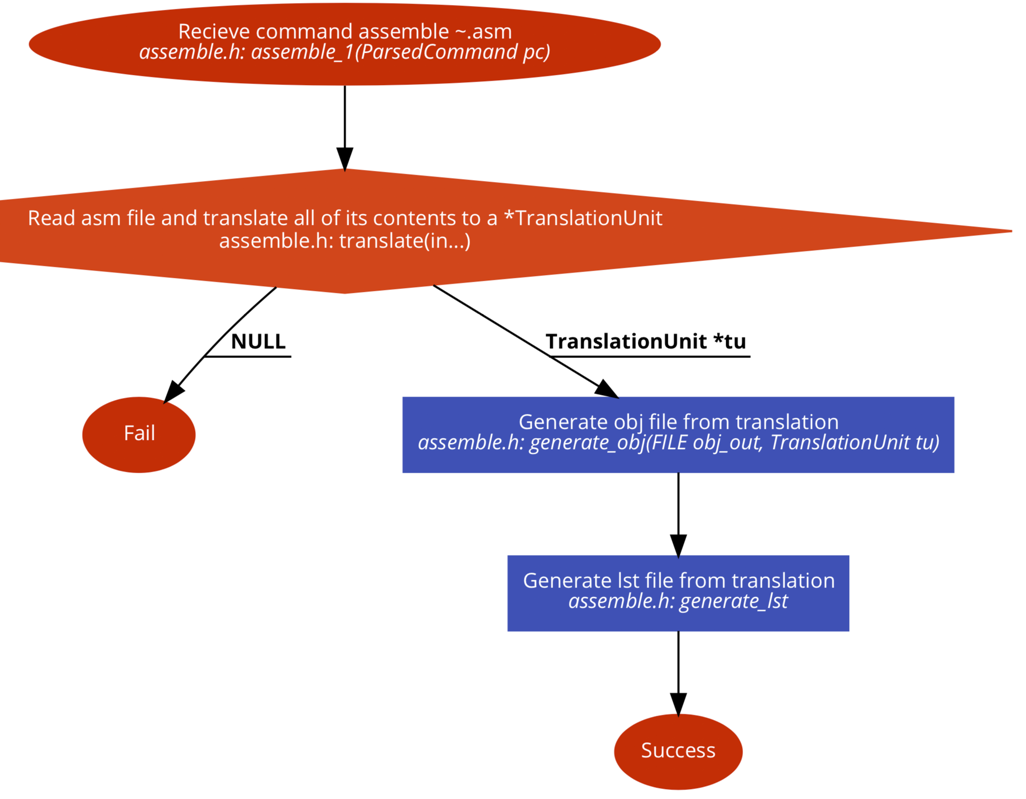
Translation 과정에서 SicStatement 구조체에는 점차 많은 정보가 쌓이게 되며, translation이 끝나면 obj 및 lst 파일 생성에 필요한 모든 정보가 SicStatement에 담기게 됩니다.

이때 SP교재에 제시된 pass 1, pass 2 과정은 assemble.c 파일 내의 translate 함수에서 호출하는 read\_asm, assign\_addresses / fill\_symbol\_table, validate\_statements함수에 나뉘어져서 이루어집니다.

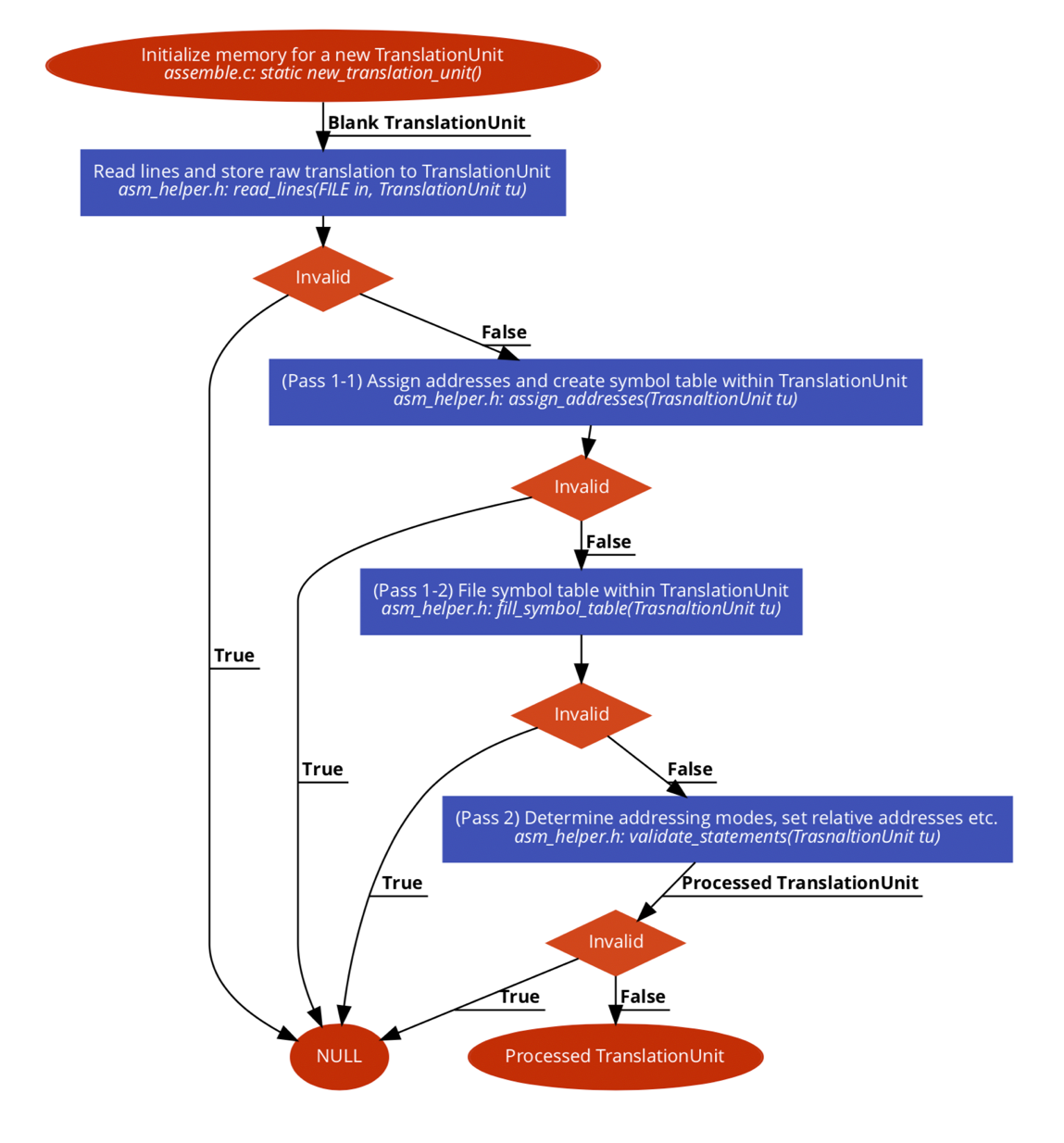
#### Object, Listing 파일 생성

Translation 과정에서 SIC/XE 코드를 모두 SicStatement 형태로 변환을 했기 때문에 이를 이용한 .obj 및 .lst 파일 생성 과정은 비교적 간단합니다. 이는 assemble.c 파일 내의 generate\_obj, generate\_lst 함수에서 처리됩니다.

#### Flowchart: Assemble 전체 과정



#### Flowchart: Assemble 내 Translation 과정



### Asm\_helper.h 모듈의 활용

assemble.c 파일 내 assemble 과정의 implementation은 대부분 asm\_helper.h에서 정의된 헬퍼 함수 호출로 이루어져 있습니다. 그래서 assemble.c에서는 assemble 과정의 흐름을 전반적으로 확인할 수 있고, 실제 asm\_helper.c 내의 구현 사항도 static function으로 나누어져 있기 때문에 헬퍼 함수의 구현도 비교적 쉽게 흐름을 파악할 수 있습니다. 이 static function들 또한 asm\_helper.c 내 주석으로 상세히 설명되어 있습니다.

# 모듈 정의

이 프로그램은 OOP 설계 원칙에 따라 모든 기능을 데이터와 그에 대한 함수로 모듈화하여 만들었습니다. 구현 언어가 C 언어로 제한되어 있기 때문에 클래스를 사용하는 대신 각각의 모듈을 파일로 분리했습니다. 파일로 분리된 모듈 내에는 구조체와 new, free 함수, 그리고 해당 구조체 타입을 첫 인자로 갖는 기타 함수를 통해 OOP 방법론을 수용하였습니다.

아래 각각의 모듈에 대해 핵심 구조체와 함수에 관한 설명을 추가했습니다. 내부적인 구현과 변수, 지역 함수 등을 제외하고 헤더에 정의된 요소에 관해서만 서술하였습니다.

**일부 모듈 설명은 영어로 되어 있습니다.**

**Note.** 모듈 간의 독립성을 향상시키고 모듈 단위의 테스트를 쉽게 진행하기 위해, 모든 출력 기능은 output stream을 이용하는 형태로 구현했습니다.

## dir [dir.h]

현재 디렉토리를 출력하기 위한 함수를 담고 있는 모듈입니다.

### 함수 설명

void fprint\_dir(FILE \*out);

*out* output stream으로 현재 디렉토리를 출력합니다.

## hashtable [hashtable.h]

연산자 mnemonic – opcode에 해당하는 key – value 쌍을 저장하기 위한 해쉬 테이블을 제공하는 모듈입니다.

### 함수 설명

typedef struct \_HashTable { … } HashTable;

해쉬 테이블 데이터를 담기 위한 구조체입니다.

**Implementation Note.** 내부 데이터는 <<데이터 쌍을 담은 TableNode로 이루어진 linked-list>에 대한 reference를 담은 BucketNode의 배열>로 저장됩니다.

HashTable \*new\_hash\_table();

필요한 메모리를 할당받아 새로운 해쉬 테이블을 초기화하여 반환합니다.

void free\_hash\_table(HashTable \*table);

해쉬 테이블에 쓰인 메모리를 해제합니다.

void add\_to\_hash\_table(HashTable \*table, char \*key, Value value);

해쉬 테이블에 새로운 key – value 쌍을 추가합니다.

int find\_from\_hash\_table(HashTable \*table, char \*key, Value \*value);

해쉬 테이블 내에서 특정 key를 찾아 그와 연결된 값을 pass-by-reference로 반환하고 성공 여부를 반환합니다.

void fprint\_hash\_table(FILE \*out, HashTable \*table);

해쉬 테이블을 지정한 output stream에 출력합니다.

## help [help.h]

이 프로그램에서 사용할 수 있는 명령어에 관한 도움말 출력 기능을 제공하는 모듈입니다.

### 함수 설명

void fprint\_help(FILE \*out);

도움말을 지정된 output stream에 출력합니다.

## history [history.h]

문자열을 저장하여 지금까지 저장된 문자열을 출력하는 기록을 제공하는 모듈로, 이번 프로그램에서는 사용자 입력 기록을 출력하는 history 명령어 구현에 쓰입니다.

### 함수 설명

typedef struct \_History { … } History;

문자열 기록 데이터를 담는 구조체입니다.

**Implementation Note.** 내부 데이터는 크기가 정해진 문자열 (문자 배열)의 linked-list로 저장됩니다.

History \*new\_history();

필요한 메모리를 할당받아 새로운 문자열 기록을 초기화하여 반환합니다.

void free\_history(History \*history);

문자열 기록에 쓰인 메모리를 해제합니다.

void add\_history(History \*history, char \*string);

기록에 새로운 문자열을 추가합니다.

void fprint\_history(FILE \*out, History \*history);

기록된 문자열을 지정된 output stream에 모두 출력합니다.

int has\_history(History \*history);

기록된 문자열이 있는지 반환합니다.

## memory [memory.h]

정해진 크기 (1MB)에 해당하는 메모리를 담을 수 있는 메모리 블록 (Block)에 대한 읽기, 쓰기, 초기화, 출력 기능을 제공하는 모듈입니다.

### 함수 설명

typedef struct \_Block { … } Block;

1MB에 해당하는 메모리를 저장하는 구조체입니다.

**Implementation Note.** 내부 데이터는 unsigned char 배열로 저장됩니다.

Block \*new\_memory\_block();

필요한 메모리를 할당받아 새로운 메모리 블록을 초기화하여 반환합니다.

// void free\_memory\_block();

메모리 블록은 별다른 내부 작업 없이 free 함수를 이용하여 메모리를 해제할 수 있습니다.

void set\_memory(Block \*block, int location, unsigned char value);

메모리 블록 내 지정된 주소에 새로운 값을 저장합니다.

void fill\_memory(Block \*block, int start, int end, unsigned char value);

메모리 블록 내 시작 주소부터 끝 주소까지 (inclusive) 새로운 값을 저장합니다.

char \*dump\_memory(Block \*block, int start, int end);

메모리 블록 내 시작 주소부터 끝 주소까지 dump 형식으로 출력된 문자열을 반환합니다.

끝 주소 인자로 -1 값이 넘겨질 경우 끝 주소를 지정하지 않은 것으로 간주하여 시작 주소부터 160개의 값을 출력합니다.

시작 주소와 끝 주소 인자로 모두 -1 값이 넘겨질 경우, 이전까지 출력한 주소 다음부터 160개의 값을 출력합니다.

이때 반환되는 문자열은 동적 할당된 것이므로 사용 후 꼭 메모리를 해제해야 합니다.

void reset\_memory(Block \*block);

메모리 블록 내의 값을 모두 0으로 초기화합니다.

## asm\_helper.h

This module helps assemble.h with the extrenuous assembly process of asm

files. Why? Because there are just too many functions and too much code. In a nutshell, this is our assembly process:

Read the assembly file, process it and save the translated SIC/XE code as a TranslationUnit. Invidiual SIC/XE statements are saved within a TranslationUnit as a linked-list of SicStatement's - which contain all the information needed to generate the .lst and .obj files.

### Function Explanations

int read\_asm(FILE \*out, FILE \*in, TranslationUnit \*tu);

Read all SIC/XE statements from input stream. = Pass 1

This reads a non-comment line from asm source code and parses labels, reserved words (operators or directives) and optionally operands and n/i/e flags. The parsed results are stored in the SicStatementNode. Also validate operator-operand compatability and determines the size of the instruction. Also assigns line number of SicStatements

- Does count START, BASE, NOBASE directives as SicStatement's.

- Does not validate symbols, address-offsets etc.

Print errors to output stream

Returns success as boolean

int fill\_symbol\_table(FILE \*out, TranslationUnit \*tu);

Reads the sic statements fills the symbol table (all within tu)

Print errors to output stream

Returns success as boolean

int validate\_statements(FILE \*out, TranslationUnit \*tu);

Validates the statements in tu. This assigns the p, b, e flag and checks if p or b relative offsets can be used for non-extended instructions. Also assigns the offset or absolute address that'll be used in the actual instruction. Also sets the base addresses.

Print errors to output stream

Returns success as boolean

void fprint\_lst\_line(FILE \*out, SicStatement \*statement, HashTable \*opcodes);

Prints the validated statement as a line in an lst file.

void fprint\_statement\_in\_hex(FILE \*out, SicStatement \*statement, HashTable \*opcodes);

Prints the hex representation of the instruction stated by the validated SicStatement.

void fprint\_header\_record(FILE \*out, TranslationUnit \*tu);

Prints the header record for translation unit tu.

void fprint\_text\_record(FILE \*out, TranslationUnit \*tu);

Prints the text record for translation unit tu.

void fprint\_end\_record(FILE \*out, TranslationUnit \*tu);

Prints the end record for translation unit tu.

void fprint\_modificaiton\_record(FILE \*out, TranslationUnit \*tu);

Prints the modification record for translation unit tu.

## assemble.h

This is where the assembly happens.

In a nutshell, this is our assembly process: Read the assembly file, process it and save the translated SIC/XE code as a TranslationUnit. Individual SIC/XE statements are saved within a TranslationUnit as a linked-list of SicStatement's - which contain all the information needed to generate the .lst and .obj files.

Note that most of the implementations involve calling helper functions from asm\_helper.h.

### Function Explanations

int assemble\_1(FILE \*out, ParsedCommand \*pc);

Assembles pc->argument[0] (\*.asm) and generates an obj file and an lst file.

Return success as boolean.

int symbol\_0(FILE \*out, ParsedCommand \*pc);

Prints out the symbol table of the previously assembled SIC/XE program.

Return TRUE (since there's no way to fail..?)

TranslationUnit \*translate(FILE \*out, FILE \*in, HashTable \*opcodes, ReservedDict \*reserved);

Translate the given asm file stream \*in\* into a TranslationUnit.

Return the resulting TranslationUnit on success.

Return NULL on fail.

void free\_translation\_unit(TranslationUnit \*tu);

Deallocates the translation unit

void generate\_obj(FILE \*out, TranslationUnit \*tu);

Generates object code from the TranslationUnit and print it to the \*out\* stream.

void generate\_lst(FILE \*out, TranslationUnit \*tu);

Generates a listing record from the TranslationUnit and print it to the \*out\* stream.

void print\_symlist(FILE \*out, TranslationUnit \*tu);

Print the list of symbols defined in the TranslationUnit and print it to the \*out\* stream.

## file.h

A module for printing the current directory and printing the contents of files.

### Function Explanations

void fprint\_dir(FILE \*out);

Print the current directory to output stream

int fprint\_file(FILE \*out, char \*filename);

Print the contents of 'filename' to output stream

## generic\_dict

This module provides a generic dictionary (internally, a hash table) that stores values as references, pointed by void pointers. To implement a dictionary that holds a custom type, just typedef Dict to the dictionary of that type (like typedef Dict StringDict;) and wrap the following dictionary functions to handle that type. All of the hash table related gizmo's are implemented within this module!

### Function Explanations

Dict \*new\_dict(int size);

Allocates and returns a new dictionary with a bucket size of \*size\*

void free\_dict(Dict \*dict, void (\*free\_value) (void \*value));

Frees the dictonary structure, and frees the invidiuals values of your custom type using a custom free function (that you provide ;)

void add\_to\_dict(Dict \*dict, char \*key, void \*value);

Adds a key-value pair to the dictionary.

int find\_from\_dict(Dict \*dict, char \*key, void \*\*value);

Finds a value cooresponing to \*key\* within \*dict\*. If found, it is assigned to \*value.

Return success as boolean.

int dict\_contains(Dict \*dict, char \*key);

Return whether if \*key\* exists within \*dict\*.

void fprint\_dict(FILE \*out, Dict \*dict, void (\*print\_value) (FILE \*out, void \*value));

Print dictionary in the following format to the \*out\* stream.

0: [KEY,VALUE] -> [KEY,VALUE]...

The VALUE part is printed using the print function \*print\_value\*. Yes, you provide that too :) Because, we don't know the type of \*value!

void fprint\_dict\_values(FILE \*out, Dict \*dict, void (\*print\_value) (FILE \*out, void \*value));

Print dictionary in the following format to the \*out\* stream.

\tKEY\tVALUE\n

\tKEY\tVALUE\n

...

Again, the VALUE part is printed using the print function \*print\_value\*. Of course, you provide that too :)

void ignore\_dict\_values(void \*value);

Use this function in free\_dict when your values do not require freeing. For example, references to statically allocated strings. This srsly doesn't do anything with the passed \*value\*.

generic\_list

This module provides a generic linked-list that stores values as references, pointed by void pointers.

To implement a linked-list that holds a custom type, just typedef List to the list of that type (like typedef List StringList;) and wrap the following linked-list functions to handle that type. All of the linked-list related gizmo's are implemented within this module!

List \*new\_list();

Allocates and returns a new linked-list.

void add\_to\_list(List \*list, void \*value);

Adds a node containing \*value\* to \*list\*.

void free\_list(List \*list, void (\*free\_value) (void \*value));

Frees the linked-list structure, and frees the invidiuals values of your custom type using a custom free function (that you provide ;)

void fprint\_list(FILE \*out, List \*list, void (\*print\_value) (FILE \*out, void \*value));

Print dictionary in the following format to the \*out\* stream.

VALUE -> VALUE -> ... -> VALUE\n

The VALUE part is printed using the print function \*print\_value\*. Yes, you provide that too :) Because, we don't know the type of \*value!

void ignore\_list\_values(void \*value);

Use this function in free\_list when your values do not require freeing. For example, references to statically allocated strings. This srsly doesn't do anything with the passed \*value\*.

## interpreter.h

This module is provides a clean interface to interpret commands (ParsedCommand's) and run the appropriate function to execute that command.

### Usage

Create a new Interpreter and add interpretable operations (OperationNode's) to that Interpreter. Each operation has an operator string, the number of arguments for that operation, and the function to call to execute that operation.

Once you've set up the Interpreter, you can let the Interpreter interpret the user's command (converted to ParsedCommand). Just call the function interpret(Interpreter ip, ParsedCommand pc), and the appropriate function will be called with the user's arguments.

### Function Explanations

Interpreter \*new\_interpreter(FILE \*out);

Allocates and returns a new interpreter.

void free\_interpreter(Interpreter \*ip);

Deallocates the interpreter and all of the statements assigned to it.

void add\_operation(Interpreter \*ip, char \*operator, int argument\_count, void \*function);

Add an operation to interpret with \*ip\*. For example, add <"command", 2, func> to \*ip\*, and the func function will be called when the user inputs something like "command argument1 argument2".

int interpret(Interpreter \*ip, ParsedCommand \*pc);

Pass the user input to \*ip\* as a ParsedCommand \*pc\* and the interpretor will 'magically' interpret that command and call the function within the corresponding operation.

Return FALSE is there is no such operation.

int interpret\_and\_free(Interpreter \*ip, ParsedCommand \*pc);

Call interpret with \*ip\* and \*pc\* and free \*pc\* - since there is usually no need to use it after it has been interpreted.

## parser.h

This modules provides the definition of struct \_ParsedCommand which stores invidiual tokens (command + arguments) of a user's command. It also provides the parse\_command function that automatically parses a user command string and generates a ParsedCommand struct.

Terminology

command: the original one-line string inputted by user

token: each word within command

operator: the actual "command word" part of the command

argument: the arguments to the operation. (all tokens excluding operation)

### Function Explanations

ParsedCommand \*parse\_command(char \*command, int \*error\_code);

Generate and return ParsedCommand from a user input string (\*command\*).

On parse fail, save error code in \*error\_code\* and return NULL.

Note that this ParsedCommand is dynamically allocated.

void free\_parsed\_command(ParsedCommand \*parsed);

Deallocate the givne ParsedCommand

## register.h

This module provides register enum definition and a convinience function to get a string represenation of a register enum.

### Function Explanations

const char \*register\_to\_string(Register reg);

Return the string representation of register enum. Note that this returns a pointer to a statically allocated string literal.

## reserved.h

To parse asm files, we need a dictionary of reserved words, including operators and directives like START, END, BASE etc. This module provides the ReservedDict type that you can use to read a record file of reserved words and interpret them during asm assembly.

The functions are basically wrapper fuctions for the generic\_dict module, so the function explanations will be omitted

## symbol.h

This module provides the SymbolDict type that you can use to store and retrive symbols and their corresponding instruction addresses.

The functions are just wrapper functions for the generic dict type, so most function explanation will be omitted.

### (Some) Function Explanations

void fprint\_symbols(FILE \*out, SymbolDict \*dict);

Prints the symbols in \*dict\* in descending order, based on the string of the symbol.

# 전역 변수

이 프로그램은 전역 변수를 최소한으로 사용하도록 설계되었습니다. 이는 모듈 간 독립성을 유지하여 충돌을 방지하고 모듈 단위의 테스트를 쉽게 할 수 있기 위함입니다.

공유가 불가피한 요소는 모두 global.h에 extern으로 선언된 struct \_Global 타입의 변수 G 안에 모두 저장했습니다. 전역 변수 사용으로 인해 발생할 수 있는 오류를 최소화하기 위해 이와 같은 방법으로 전역 변수를 묶었습니다.

## struct \_Global G 내 전역 변수

### History \*history

사용자 입력 명령어 기록을 관리하기 위해 [history.h] 모듈에서 사용하는 구조체

### HashTable table;

Opcode 테이블을 저장하기 위해 [hashtable.h] 모듈에서 사용하는 구조체

### Block \*block;

메모리 관련 명령어로 조작하는 메모리 공간의 데이터를 담은 구조체

### ReservedDict \*reserved;

Assemble 과정에서 필요한 operator와 directive 등의 reserved word를 사전 (해쉬테이블) 형태로 담은 구조체

# 특이 사항

## Inline 함수

main.c 및 다른 모듈 내에는 readability 향상을 위한 몇가지 지역 함수가 있습니다. 이들 중 일부는 컴파일러 최적화를 위해 static inline으로 선언했습니다. 해당 기능은 C99 표준에서 소개된 기능이지만 실행 환경인 cspro의 gcc 버젼 (5.4.0)의 기본 C언어 컴파일 표준이 GNU11이기 때문에 별다른 컴파일 옵션을 추가하지 않았습니다.

## Macro 정의를 이용한 모듈 테스트

모든 모듈의 module\_name.c 파일에는 모듈 테스트를 위한 main 함수가 정의되어 있습니다. 이는TEST라는 매크로 정의를 선언해야만 컴파일 단위에 포함이 되도록 IFDEF, ENDIF 전처리문으로 둘러쌓여 있습니다. 해당 main 함수를 이용해서 모듈을 테스트하려면 gcc 컴파일 명령어에 -D TEST 옵션을 추가해야 합니다. 이 경우 main 기호가 다중으로 선언되는 문제로 인해 다른 모듈과 링크를 하여 테스트할 수 없습니다. 모듈 단위의 테스트가 목적이기 때문에 위와 같은 방법으로 전처리기를 사용했습니다.

## void 포인터를 활용한 generic 자료 구조

이번 프로그램을 설계하면서linked-list와 해시 테이블을 다양한 데이터 타입에 대해 구현할 필요가 있었습니다. 자료 구조의 조작과 관련된 코드를 중복해서 작성하기보다는 자료 구조 구현 논리를 재사용하기 위해 값 (value)을 void\* 타입으로 저장하는 타입-무관한 generic 자료 구조 모듈을 구현했습니다. 그리고 자료 구조에 담아야하는 자료형에 따라 해당 모듈을 wrapping하는 식으로 custom 타입의 자료 구조 모듈을 추가적으로 구현했습니다.

Generic linked-list는 [generic\_list.h], generic 해시테이블은 [generic\_dict.h]에 정의되어 있고, 구현 및 사용 관련 사항은 주석으로 상세히 설명되어 있습니다. 아래 자료 구조는 모두 위 generic 자료 구조를 바탕으로 구현되었습니다.

* [hashtable.h] 모듈 내의 opcode 테이블
* [interpreter.h]에 정의된 Interpreter 내의 명령어 리스트
* [asm\_helper.h]에 정의된 TranslationUnit 내의 SicStatement 리스트
* [symbol.h] 모듈 내의 symbol 테이블
* [reserved.h] 모듈 내의 reserved word 테이블

## 가독성 및 주석 관련

이번 프로그램에서는 주석의 필요성을 최소화하기 위해 logic을 모듈 단위로 나누고, 긴 코드는 static 함수로 나누어 함수 이름을 descriptive하게 정의하는 등 많은 refactoring 작업을 통해 가독성을 최대화하였습니다. 그리하여 대부분의 주석은 \*.h 파일 내 함수 정의 뒤의 함수 설명에 국한되어 있으며, \*.c 파일 내 implementation code에서는 주석을 추가하지 않았습니다.

# 코드

/\*

\* asm\_helper.h

\*

\* This module helps assemble.h with the extrenuous assembly process of asm

\* files. Why? Because there are just too many functions and too much code.

\*

\*/

#ifndef ASM\_HELPER\_H

#define ASM\_HELPER\_H

#include "reserved.h"

#include "symbol.h"

#include "hashtable.h"

#include "generic\_list.h"

#include "register.h"

#include <stdio.h>

#define SIC\_MNEMONIC\_LENGTH 10

#define SIC\_LABEL\_LENGTH 7

#define SIC\_MAX\_BYTE\_LENGTH 4096

typedef enum \_AddressingMode {

IMMEDIATE, INDIRECT, SIMPLE,

} AddressingMode;

typedef struct \_SicStatement {

unsigned int address;

unsigned int instruction\_size;

int line\_number;

char label[SIC\_LABEL\_LENGTH];

int base; // -1 for NOBASE

char p, b, e;

ReservedType type;

char \*reserved\_string;

union {

struct { // memory operand or end

AddressingMode mode;

char x;

char is\_literal;

int adjusted\_address;

struct {

char label[SIC\_LABEL\_LENGTH];

unsigned int literal;

};

};

struct { // single register

Register r;

};

struct { // register pair

Register r1;

Register r2;

};

struct { // byte

char is\_b; // TRUE: X'0E1', FALSE: C'HELLO'

char is\_odd\_b; // TRUE: X'203', FALSE: X'2034'

int ulength; // The number of hexes in X'\*'

union {

char \*bytes; // The string within C'\*'

unsigned char \*ubytes; // The array of bytes within X'\*'

};

};

struct { // word

unsigned int word;

};

struct { // reserved size

int rsize;

};

struct {

int start\_address;

};

} operands;

} SicStatement;

#define SicStatementList List

typedef struct \_TranslationUnit {

HashTable \*opcodes;

ReservedDict \*reserved;

SicStatementList \*statements;

SymbolTable \*symbols;

} TranslationUnit;

SicStatementList \*new\_statement\_list();

void free\_statement\_list(SicStatementList \*list);

int read\_asm(FILE \*out, FILE \*in, TranslationUnit \*tu);

/\*

\* Read all SIC/XE statements from input stream. = Pass 1

\*

\* This reads a non-comment line from asm source code and parses labels,

\* reserved words (operators or directives) and optionally operands and

\* n/i/e flags. The parsed results are stored in the SicStatementNode.

\* Also validate operator-operand compatability and determines the size

\* of the instruction. Also assigns line number of SicStatements

\*

\* - Does count START, BASE, NOBASE directives as SicStatement's.

\* - Does not validate symbols, address-offsets etc.

\*

\* Print errors to output stream

\*

\* Returns success as boolean

\*/

int assign\_addresses(FILE \*out, TranslationUnit \*tu);

int fill\_symbol\_table(FILE \*out, TranslationUnit \*tu);

/\*

\* Reads the sic statements fills the symbol table (all within tu)

\*

\* Print errors to output stream

\*

\* Returns success as boolean

\*/

int validate\_statements(FILE \*out, TranslationUnit \*tu);

/\*

\* Validates the statements in tu. This assigns the p, b, e flag and checks

\* if p or b relative offsets can be used for non-extended instructions. Also

\* assigns the offset or absolute address that'll be used in the actual

\* instruction. Also sets the base addresses.

\*

\* Print errors to output stream

\*

\* Returns success as boolean

\*/

void fprint\_lst\_line(FILE \*out, SicStatement \*statement, HashTable \*opcodes);

/\*

\* Prints the validated statement as a line in an lst file.

\*/

void fprint\_statement\_in\_hex(FILE \*out, SicStatement \*statement,

HashTable \*opcodes);

/\*

\* Prints the hex representation of the instruction stated by the

\* validated SicStatement.

\*/

void fprint\_header\_record(FILE \*out, TranslationUnit \*tu);

/\*

\* Prints the header record for translation unit tu.

\*/

void fprint\_text\_record(FILE \*out, TranslationUnit \*tu);

/\*

\* Prints the text record for translation unit tu.

\*/

void fprint\_end\_record(FILE \*out, TranslationUnit \*tu);

/\*

\* Prints the end record for translation unit tu.

\*/

void fprint\_modificaiton\_record(FILE \*out, TranslationUnit \*tu);

/\*

\* Prints the modification record for translation unit tu.

\*/

#endif

/\*

\* assemble.h

\*

\* This is where the assembly happens.

\*

\* In a nutshell, this is our assembly process:

\* Read the assembly file, process it and save the translated SIC/XE code

\* as a TranslationUnit. Invidiual SIC/XE statements are saved within a

\* TranslationUnit as a linked-list of SicStatement's - which contain all

\* the information needed to generate the .lst and .obj files.

\*

\* Note that most of the implementations involve calling helper functions from

\* asm\_helper.h.

\*/

#ifndef ASSEMBLE\_H

#define ASSEMBLE\_H

#include "hashtable.h"

#include "reserved.h"

#include "symbol.h"

#include "asm\_helper.h"

#include "parser.h"

int assemble\_1(FILE \*out, ParsedCommand \*pc);

/\*

\* Assembles pc->argument[0] (\*.asm) and generates an obj file and an lst

\* file.

\*

\* Return success as boolean.

\*/

int symbol\_0(FILE \*out, ParsedCommand \*pc);

/\*

\* Prints out the symbol table of the previously assembled SIC/XE program.

\*

\* Return TRUE (since there's no way to fail..?)

\*/

TranslationUnit \*translate(

FILE \*out, FILE \*in, HashTable \*opcodes, ReservedDict \*reserved);

/\*

\* Translate the given asm file stream \*in\* into a TranslationUnit.

\*

\* Return the resulting TranslationUnit on success.

\* Return NULL on fail.

\*/

void free\_translation\_unit(TranslationUnit \*tu);

/\*

\* Deallocates the translation unit

\*/

void generate\_obj(FILE \*out, TranslationUnit \*tu);

/\*

\* Generates object code from the TranslationUnit and print it to the \*out\*

\* stream.

\*/

void generate\_lst(FILE \*out, TranslationUnit \*tu);

/\*

\* Generates a listing record from the TranslationUnit and print it to the

\* \*out\* stream.

\*/

void print\_symlist(FILE \*out, TranslationUnit \*tu);

/\*

\* Print the list of symbols defined in the TranslationUnit and print it to

\* the \*out\* stream.

\*/

#endif

/\*

\* file.h

\*

\* A module for printing the current directory and printing the

\* contents of files.

\*/

#ifndef FILE\_H

#define FILE\_H

#include <stdio.h>

void fprint\_dir(FILE \*out);

/\* Description

\* Print the current directory to output stream

\*/

int fprint\_file(FILE \*out, char \*filename);

/\* Description

\* Print the contents of 'filename' to output stream

\*/

#endif

#ifndef GENERIC\_DICT\_H

#define GENERIC\_DICT\_H

#include <stdio.h>

/\*

\* generic\_dict

\*

\* This module provides a generic dictionary (internally, a hash table) that

\* stores values as references, pointed by void pointers.

\*

\* To implement a dictionary that holds a custom type, just typedef Dict to

\* the dictionary of that type (like typedef Dict StringDict;) and wrap the

\* following dictionary functions to handle that type. All of the hash table

\* related gizmo's are implemented within this module!

\*/

typedef struct \_BucketNode {

char \*key;

void \*value;

struct \_BucketNode \*link;

} BucketNode;

typedef struct \_DictNode {

BucketNode \*head;

BucketNode \*last;

} DictNode;

typedef struct \_Dict {

int size;

DictNode \*nodes;

} Dict;

Dict \*new\_dict(int size);

/\*

\* Allocates and returns a new dictionary with a bucket size of \*size\*

\*/

void free\_dict(Dict \*dict, void (\*free\_value) (void \*value));

/\*

\* Frees the dictonary structure, and frees the invidiuals values of your

\* custom type using a custom free function (that you provide ;)

\*/

void add\_to\_dict(Dict \*dict, char \*key, void \*value);

/\*

\* Adds a key-value pair to the dictionary.

\*/

int find\_from\_dict(Dict \*dict, char \*key, void \*\*value);

/\*

\* Finds a value cooresponing to \*key\* within \*dict\*. If found, it is assigned

\* to \*value.

\*

\* Return success as boolean.

\*/

int dict\_contains(Dict \*dict, char \*key);

/\*

\* Return whether if \*key\* exists within \*dict\*.

\*/

void fprint\_dict(FILE \*out, Dict \*dict, void (\*print\_value) (FILE \*out, void \*value));

/\*

\* Print dictionary in the following format to the \*out\* stream.

\* 0: [KEY,VALUE] -> [KEY,VALUE]...

\*

\* The VALUE part is printed using the print function \*print\_value\*.

\* Yes, you provide that too :) Because, we don't know the type of \*value!

\*/

void fprint\_dict\_values(FILE \*out, Dict \*dict,

void (\*print\_value) (FILE \*out, void \*value));

/\*

\* Print dictionary in the following format to the \*out\* stream.

\* \tKEY\tVALUE\n

\* \tKEY\tVALUE\n

\* ...

\*

\* Again, the VALUE part is printed using the print function \*print\_value\*.

\* Of course, you provide that too :)

\*/

void ignore\_dict\_values(void \*value);

/\*

\* Use this function in free\_dict when your values do not require freeing.

\* For example, references to statically allocated strings. This srsly doesn't

\* do anything with the passed \*value\*.

\*/

#endif

/\*

\* generic\_list

\*

\* This module provides a generic linked-list that stores values as

\* references, pointed by void pointers.

\*

\* ~~Because we have to implement like 15 frigging different linked-lists

\* during this project mann.~~

\*

\* To implement a linked-list that holds a custom type, just typedef List to

\* the list of that type (like typedef List StringList;) and wrap the

\* following linked-list functions to handle that type. All of the linked-list

\* related gizmo's are implemented within this module!

\*/

#ifndef GENERIC\_LIST\_H

#define GENERIC\_LIST\_H

#include <stdio.h>

typedef struct \_LinkedNode {

void \*value;

struct \_LinkedNode \*link;

} LinkedNode;

typedef struct \_List {

LinkedNode \*head;

LinkedNode \*last;

} List;

List \*new\_list();

/\*

\* Allocates and returns a new linked-list.

\*/

void add\_to\_list(List \*list, void \*value);

/\*

\* Adds a node containing \*value\* to \*list\*.

\*/

void free\_list(List \*list, void (\*free\_value) (void \*value));

/\*

\* Frees the linked-list structure, and frees the invidiuals values of your

\* custom type using a custom free function (that you provide ;)

\*/

void fprint\_list(FILE \*out, List \*list, void (\*print\_value) (FILE \*out, void \*value));

/\*

\* Print dictionary in the following format to the \*out\* stream.

\* VALUE -> VALUE -> ... -> VALUE\n

\*

\* The VALUE part is printed using the print function \*print\_value\*.

\* Yes, you provide that too :) Because, we don't know the type of \*value!

\*/

void ignore\_list\_values(void \*value);

/\*

\* Use this function in free\_list when your values do not require freeing.

\* For example, references to statically allocated strings. This srsly doesn't

\* do anything with the passed \*value\*.

\*/

#endif

#include "history.h"

#include "hashtable.h"

#include "memory.h"

#include "reserved.h"

/\*

\* global.h

\*

\* A horrible module that makes global variables just slightly more safe!

\*/

typedef struct \_Global {

History \*history;

HashTable \*table;

Block \*block;

ReservedDict \*reserved;

} Global;

extern Global G;

#ifndef HASHTABLE\_H

#define HASHTABLE\_H

#include "generic\_dict.h"

#include <stdio.h>

typedef struct \_Opcode {

unsigned char raw;

} Opcode;

#define HashTable Dict

HashTable \*new\_hash\_table();

/\* Description

\* Initiate memory for a valid HashTable.

\*/

void free\_hash\_table(HashTable \*table);

/\* Description

\* Free memory and destory HashTable.

\*/

void add\_to\_hash\_table(HashTable \*table, char \*key, unsigned char opcode);

/\* Description

\* Add a key-value entry to HashTable.

\*

\* Arguments

\* char \*key: key as a string.

\* Value value: the value to store. To change the fields of the value later

\* on, change the definition of struct Value

\*/

int find\_from\_hash\_table(HashTable \*table, char \*key, unsigned char \*opcode);

/\* Description

\* Find the value associated with key from table

\*

\* Arguments

\* char \*key: key as string

\* Value \*value: return found value to reference

\*

\* Return

\* A boolean value for whether the key exists

\*/

void fprint\_hash\_table(FILE \*out, HashTable \*table);

/\* Description

\* Prints the hash-table to a human-readable form to an output stream

\*/

#endif

#ifndef HELP\_H

#define HELP\_H

#include <stdio.h>

void fprint\_help(FILE \*out);

/\* Description

\* Print a help dialog to file stream.

\*/

#endif

/\*

\* history.h

\*

\* This module provides a functions related to storing and priting the

\* users command history. Basically, this is for the hi[story] command.

\*/

#ifndef HISTORY\_H

#define HISTORY\_H

#include "main.h"

#include <stdio.h>

typedef struct \_HistoryNode {

struct \_HistoryNode \*next;

char string[COMMAND\_LENGTH];

} HistoryNode;

typedef struct \_History {

HistoryNode \*head; // fake head

HistoryNode \*last;

} History;

/\* Description

\* Stores command history as a linked-list.

\*/

History \*new\_history();

/\* Description

\* Initiate memory for a valid history record.

\*/

void free\_history(History \*history);

/\* Description

\* Free memory and destroy history record.

\*/

void add\_history(History \*history, char \*string);

/\* Description

\* Add a previous command to a history record.

\*

\* Arguments

\* char \*string: the command string

\*/

void fprint\_history(FILE \*out, History \*history);

/\* Description

\* Print the history to an output stream.

\*/

int has\_history(History \*history);

/\* Description

\* Return whether there are any commands in the history record.

\*/

#endif

/\*

\* interpreter.h

\*

\* This module is provides a clean interface to interpret commands

\* (ParsedCommand's) and run the appropriate function to execute that command.

\*

\* Usage

\* Create a new Interpreter and add interpretable operations (OperationNode's)

\* to that Interpreter. Each operation has an operator string, the number

\* of arguments for that operation, and the function to call to execute that

\* operation.

\*

\* IMPORTANT: The signature of the function must be

\* int f(FILE \*out, ProcessedCommand \*pc);

\* This will not be explicitly checked during complication nor run-time.

\*

\* Once you've set up the Interpreter, you can let the Interpreter interpret

\* the user's command (converted to ParsedCommand). Just call the function

\* interpret(Interpreter ip, ParsedCommand pc), and the appropriate function

\* will be called with the user's arguments.

\*/

#ifndef INTERPRETER\_H

#define INTERPRETER\_H

#include "parser.h"

typedef struct \_OperationNode {

char operator[TOKEN\_LENGTH];

int argument\_count;

int (\*function)(FILE\*, ParsedCommand\*);

struct \_OperationNode \*link;

} OperationNode;

/\*

\* Linked-list node for storing the interpretable operations. Note that

\* the module treats the same operator with different number of arguments

\* as separate 'Operation's.

\*/

typedef struct \_Interpreter {

OperationNode \*head;

OperationNode \*last;

FILE \*output\_stream;

} Interpreter;

Interpreter \*new\_interpreter(FILE \*out);

/\*

\* Allocates and returns a new interpreter.

\*/

void free\_interpreter(Interpreter \*ip);

/\*

\* Deallocates the interpreter and all of the statements assigned to it.

\*/

void add\_operation(Interpreter \*ip, char \*operator,

int argument\_count, void \*function);

/\*

\* Add an operation to interpret with \*ip\*. For example, add <"command",

\* 2, func> to \*ip\*, and the func function will be called when the user

\* inputs something like "command argument1 argument2".

\*/

int interpret(Interpreter \*ip, ParsedCommand \*pc);

/\*

\* Pass the user input to \*ip\* as a ParsedCommand \*pc\* and the interpretor

\* will 'magically' interpret that command and call the function within

\* the corresponding operation.

\*

\* Return FALSE is there is no such operation.

\*/

int interpret\_and\_free(Interpreter \*ip, ParsedCommand \*pc);

/\*

\* Call interpret with \*ip\* and \*pc\* and free \*pc\* - since there is usually

\* no need to use it after it has been interpreted.

\*/

#endif

#ifndef MAIN\_H

#define MAIN\_H

#define COMMAND\_LENGTH 120

#endif

#define BLOCK\_SIZE 1048576

#define MAX\_DUMP\_LENGTH 1000000

#ifndef MEMORY\_H

#define MEMORY\_H

#include <stdio.h>

typedef struct \_Block {

unsigned char data[BLOCK\_SIZE];

int current;

} Block;

/\* Description

\* A memory block that stores 1 megabyte of unsigned chars.

\*

\* Members

\* unsigned char data[]:

\* int current: the current position used for dump operations

\*/

Block \*new\_memory\_block();

/\* Description

\* Initiate memory for a valid memory block.

\*/

void set\_memory(Block \*block, int location, unsigned char value);

/\* Description

\* Find the value associated with key from table.

\*

\* Arguments

\* char \*key: key as string

\* Value \*value: return found value to reference

\*

\* Return

\* A boolean value for whether the key exists

\*/

void fill\_memory(Block \*block, int start, int end, unsigned char value);

/\* Description

\* Fill the memory block with a given value from start to end (inclusive).

\*/

void dump\_memory(FILE \*out, Block \*block, int start, int end);

/\* Description

\* Print the values within the block from start to end (inclusive) in a

\* dump format to output stream.

\*

\* Return

\* A dynamically allocated string (make sure to free it)

\*/

void reset\_memory(Block \*block);

/\* Description

\* Fill the memory block with 0s.

\*/

#endif

/\*

\* parser.h

\*

\* This modules provides the definition of struct \_ParsedCommand which

\* stores invidiual tokens (command + arguments) of a user's command.

\* It also provides the parse\_command function that automatically

\* parses a user command string and generates a ParsedCommand struct.

\*

\* Terminology

\* command: the original one-line string inputted by user

\* token: each word within command

\* operator: the actual "command word" part of the command

\* argument: the arguments to the operation. (all tokens excluding operation)

\*

\* Note that TOKEN\_COUNT is the maximum number of arguments + 1

\*/

#ifndef PARSER\_H

#define PARSER\_H

#include "main.h"

#define INVALID\_FORMAT\_ERROR 1

#define TOO\_MANY\_ARGUMENTS\_ERROR 2

#define COMMAND\_TOO\_LONG\_ERROR 3

#define ARGUMENT\_TOO\_LONG\_ERROR 4

#define TOKEN\_COUNT 4

#define ARGUMENT\_COUNT 3

#define TOKEN\_LENGTH 40

typedef struct \_ParsedCommand {

char original\_command[COMMAND\_LENGTH];

char tokenized\_command[COMMAND\_LENGTH];

char \*operator;

char \*arguments[ARGUMENT\_COUNT];

int argument\_count;

} ParsedCommand;

ParsedCommand \*parse\_command(char \*command, int \*error\_code);

/\*

\* Generate and return ParsedCommand from a user input string (\*command\*).

\*

\* On parse fail, save error code in \*error\_code\* and return NULL.

\*

\* Note that this ParsedCommand is dynamically allocated.

\*/

void free\_parsed\_command(ParsedCommand \*parsed);

/\*

\* Deallocate the givne ParsedCommand

\*/

#endif

/\*

\* register.h

\*

\* This module provides register enum definition and a convinience function

\* to get a string represenation of a register enum.

\*/

#ifndef REGISTER\_H

#define REGISTER\_H

typedef enum \_Register {

A = 0, X = 1, L = 2, B = 3,

S = 4, T = 5, PC = 8, SW = 9,

} Register;

const char \*register\_to\_string(Register reg);

/\*

\* Return the string representation of register enum.

\* Note that this returns a pointer to a statically allocated string literal.

\*/

#endif

/\*

\* reserved.h

\*

\* To parse asm files, we need a dictionary of reserved words, including

\* operators and directives like START, END, BASE etc. This module provides

\* the ReservedDict type that you can use to read a record file of reserved

\* words and interpret them during asm assembly.

\*

\* The functions below are basically wrapper fuctions for the generic\_dict

\* module, so they are pretty self-explanatory.

\*/

#ifndef RESERVED\_H

#define RESERVED\_H

#include "generic\_dict.h"

#define RESERVED\_OPERATION\_FLAG 8

typedef enum \_ReservedType {

MEM\_OPERATION = 8,

REG\_OPERATION = 9,

REG\_PAIR\_OPERATION = 10,

REG\_N\_OPERATION = 11,

N\_OPERATION = 12,

BLANK\_OPERATION = 13,

BYTE = 0,

WORD = 1,

RBYTE = 2,

RWORD = 3,

START = 4,

END = 5,

BASE = 6,

NOBASE = 7

} ReservedType;

typedef struct \_ReservedValue {

ReservedType type;

} ReservedValue;

typedef Dict ReservedDict;

ReservedDict \*new\_reserved\_dict();

void free\_reserved\_dict(ReservedDict \*dict);

void add\_to\_reserved\_dict(ReservedDict \*dict, char \*key, ReservedType type);

int find\_reserved\_type(ReservedDict \*dict, char \*key, ReservedType \*type);

#endif

/\*

\* symbol.h

\*

\* This module provides the SymbolDict type that you can use to store and

\* retrive symbols and their corresponding instruction addresses.

\*

\* The functions are just wrapper functions for the generic dict type, so

\* they are pretty self explanatory.

\*/

#ifndef SYMBOL\_H

#define SYMBOL\_H

#include "generic\_dict.h"

#include <stdio.h>

#define BASE\_SIZE 10

#define SYMBOL\_LENGTH 6

typedef struct \_Symbol {

unsigned int address;

} Symbol;

typedef Dict SymbolDict;

typedef SymbolDict SymbolTable;

SymbolDict \*new\_symbol\_dict();

void add\_to\_symbol\_dict(SymbolDict \*dict, char \*string, unsigned int address);

int find\_symbol\_address(SymbolDict \*dict, char \*string, unsigned int \*address);

void free\_symbol\_dict(SymbolDict \*dict);

void fprint\_symbols(FILE \*out, SymbolDict \*dict);

/\*

\* Prints the symbols in \*dict\* in descending order, based on the string of

\* the symbol.

\*/

#define new\_symbol\_table new\_symbol\_dict

#define add\_to\_symbol\_table add\_to\_symbol\_dict

#define free\_symbol\_table free\_symbol\_dict

#endif

/\*

\* asm\_helper.c

\*

\* This source file contains a lot of complicated static functions to assist

\* with the implementation of functions from asm\_helper.h. (assister functions

\* for helper functions)

\*

\* The assister functions themselves are very complicated, so each of them are

\* extensively documented. Enjoy!

\*/

#include "asm\_helper.h"

#include "assemble.h"

#include "hashtable.h"

#include "generic\_list.h"

#include "symbol.h"

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <string.h>

#include <assert.h>

static int sscan\_label\_and\_reserved(

char \*\*asm\_line\_cursor, SicStatement \*statement,

ReservedDict \*reserved);

/\*

\* Pass 1 Step 1

\*

\* Scan for an optional label and reserved word (operator etc.) from asm line.

\* Save the results to SicStatement.

\*

\* Ignore preceding whitespaces, and move the string cursor to the leading

\* whitespace after the last character.

\*

\* Return success as boolean.

\*/

static int sscan\_operand(char \*\*asm\_line\_cursor, SicStatement \*statement);

/\*

\* Pass 1 Step 2

\*

\* Scans for an appropriate operand corresponding to the ReservedType (the

\* type of operator or reserved word) following the string cursor.

\*

\* Ignore preceding whitespaces, and move the string cursor to the leading

\* whitespace after the last character.

\*

\* Return success as boolean.

\*/

#define READ\_LINE\_INVALID -1

#define READ\_LINE\_SUCCESS 0

#define READ\_LINE\_EOF 1

#define READ\_LINE\_BLANK 2

static int read\_line(FILE \*in, char \*\*string);

/\*

\* Read one line from input stream and save to \*string unless it is a blank

\* line. Save NULL to \*string if it is a comment.

\*

\* Return Value

\* -1: invalid

\* 0: successful

\* 1: eof

\* 2: blank line

\*/

static SicStatement \*add\_statement(SicStatementList \*list);

/\*

\* Allocate ListNode and add new SicStatement to SicStatementList

\*/

static void free\_statement(void \*s\_void);

/\*

\* Deallocate SicStatement. Passed to generic List free function.

\*/

#define SSCAN\_WORD\_BUFFER\_SIZE 33

#define SSCAN\_WORD\_NO\_COMMA 0

#define SSCAN\_WORD\_REQUIRE\_COMMA 1

static char \*sscan\_word(char \*\*cursor, int require\_comma\_separator);

/\*

\* Scans for a word (non-whitespace-separated string) following the character

\* pointed by the string cursor and moves the cursor to the following

\* whitespace or NULL character.

\*

\* If require\_comma\_separator is set to TRUE, make sure that there is a

\* comma between the current cursor and that integer.

\*

\* Returns the word as a dynamically allocated string.

\*

\* If there is no word after the cursor or if the comma separator requirement

\* is not met, return NULL.

\*/

#define SSCAN\_NO\_COMMA 0

#define SSCAN\_REQUIRE\_COMMA 1

#define SSCAN\_INT\_NONE -1

#define SSCAN\_INT\_NEGATIVE -2

static int sscan\_int(char \*\*cursor, int require\_comma\_separator);

/\*

\* Scans for an int (actually, non-negative integer) following

\* the character pointed by the string cursor, and moves the cursor to the

\* whitespace following that integer.

\*

\* If require\_comma\_separator is set to TRUE, make sure that there is a

\* comma between the current cursor and that integer.

\*

\* Returns the scanned int value.

\*

\* If there is no non-neg int after the cursor or if the comma separator

\* requirement is not met, return NULL.

\*/

static int sscan\_register(

char \*\*cursor, Register \*reg, int require\_comma\_separator);

/\*

\* Scans for a register like X, PC... following the character pointed by the

\* string cursor, ignoring whitespaces. This moves the cursor to the

\* whitespace following that integer.

\*

\* If require\_comma\_separator is set to TRUE, make sure that there is a

\* comma between the current cursor and that register.

\*

\* On success, set \*reg to the cooresponding enum.

\*

\* Returns success as boolean. If there is no non-neg int after the cursor

\* or if the comma separator requirement is not met, return FALSE.

\*/

static int read\_memory\_operand(char \*word, SicStatement \*statement);

/\*

\* Scans for a memory operand (including addressing mode flags, immediate

\* values, and labels) within string \*word\*.

\*

\* On success, save the results to statement->operands.

\*

\* Returns success as boolean. If we fail to read a memmory operand,

\* return FALSE.

\*/

static int sscan\_bytes(char \*\*cursor, SicStatement \*statement);

/\*

\* Scans for a BYTE operand like X'0F9' or C'Hello World!' following the

\* string cursor, ignoring whitespaces. On success, this moves the cursor to

\* the end of the string.

\*

\* On success, save the results to statement->operands.

\*

\* Returns success as boolean. If we fail to read a BYTE operand or there

\* are preceeding whitespaces, return FALSE.

\*/

static int read\_reserved(

char \*word, SicStatement \*statement, ReservedDict \*reserved);

/\*

\* Check if \*word\* is a reserved word. If so, save the reserved

\* word to statement->reserved\_string and set (ReservedType) statement->type

\* to the corresponding enum and return TRUE. If not, return FALSE.

\*/

static int sscan\_operand(char \*\*asm\_line\_cursor, SicStatement \*statement);

/\*

\* Scans for an appropriate operand corresponding to the ReservedType (the

\* type of operator or reserved word) following the string cursor.

\*

\* Return success as boolean.

\*/

static void strcat\_byte(char \*string, SicStatement \*statement, int raw);

/\*

\* Concat the in-obj (raw) or in-asm format of BYTE stored in statement

\* to \*string\*.

\*/

static void fflush\_text\_record\_line(FILE \*out, HashTable \*opcodes,

LinkedNode \*start\_node, LinkedNode \*end\_node);

/\*

\* Flushes the obj representation of the instructions contained in

\* the SicStatements saved within the (LinkedList) SicStatementList, starting

\* from the start\_node (inclusive) until the end\_node (exclusive).

\*

\* The content is flushed to the \*out\* stream.

\*/

static inline int is\_whitespace(char c); // self explanatory

static inline int is\_whitespace\_or\_comma(char c); // self explanatory

static inline int isupperxdigit(char c); // self explanatory

static inline void skip\_whitespaces(char \*\*cursor);

/\*

\* Moves string cursor to the first following non-whitespace character.

\* Basically, skip all the preceding whitespaces.

\*/

static inline int skip\_whitespaces\_and\_comma(char \*\*cursor);

/\*

\* Moves string cursor to the first following non-whitespace character,

\* including exactly one comma. Used for skipping a comma-included whitespace,

\* like the one between BYTE and X in "BYTE , X"

\*

\* Return 0 if there are more or less than 1 comma until the next

\* non-whitespace character.

\*/

static inline int is\_reserved(char \*word, ReservedDict \*reserved);

/\*

\* Check if the \*reserved\* dict contains \*word".

\*/

#ifdef TEST

#undef TEST

#include "generic\_list.c"

#include "generic\_dict.c"

#include "symbol.c"

#include "reserved.c"

#include "hashtable.c"

int main(void) {

return 0;

}

#endif

SicStatementList \*new\_statement\_list() {

return (SicStatementList\*) new\_list();

}

static void free\_statement(void \*s\_void) {

SicStatement \*s = (SicStatement\*) s\_void;

if (s->reserved\_string) free(s->reserved\_string);

if (s->type == BYTE && !s->operands.is\_b && s->operands.bytes)

free(s->operands.bytes);

if (s->type == BYTE && s->operands.is\_b && s->operands.ubytes)

free(s->operands.ubytes);

free(s);

}

void free\_statement\_list(SicStatementList \*list) {

free\_list(list, free\_statement);

}

int read\_asm(FILE \*out, FILE \*in, TranslationUnit \*tu) {

char \*asm\_line;

int res;

int line\_number;

for (line\_number = 1; ; ++line\_number) {

res = read\_line(in, &asm\_line);

if (asm\_line) {

char \*cursor = asm\_line;

SicStatement \*s = add\_statement(tu->statements);

s->line\_number = line\_number \* 5;

if (!sscan\_label\_and\_reserved(&cursor, s, tu->reserved)) {

fprintf(out,

"error: an error occured on line %d\n", line\_number);

fprintf(out, ">> %s\n", asm\_line);

return 0;

}

if (!sscan\_operand(&cursor, s)) {

fprintf(out,

"error: an error occured on line %d\n", line\_number);

fprintf(out, ">> %s\n", asm\_line);

return 0;

}

free(asm\_line);

} else {

if (res == READ\_LINE\_EOF) return 1;

if (res == READ\_LINE\_BLANK) continue;

if (res == READ\_LINE\_INVALID) {

fprintf(out, "error: could not read line number %d\n", line\_number);

return 0;

}

}

}

return 1;

}

int assign\_addresses(FILE \*out, TranslationUnit \*tu) {

int address = 0;

int address\_set = 0;

for (LinkedNode \*node = tu->statements->head->link;

node; node = node->link) {

SicStatement \*s = (SicStatement\*) node->value;

if (s->type == START) {

address = s->operands.start\_address;

address\_set = 1;

break;

}

}

if (!address\_set) {

fprintf(out, "error: there is no START directive\n"); return 0;

}

for (LinkedNode \*node = tu->statements->head->link;

node; node = node->link) {

SicStatement \*s = (SicStatement\*) node->value;

s->address = address;

switch (s->type) {

case START:

case END:

case BASE:

case NOBASE:

s->instruction\_size = 0; break;

case BYTE:

s->instruction\_size = strlen(s->operands.bytes); break;

case WORD:

s->instruction\_size = 3; break;

case RBYTE:

case RWORD:

s->instruction\_size = s->operands.rsize; break;

case REG\_OPERATION:

case REG\_PAIR\_OPERATION:

s->instruction\_size = 2; break;

case MEM\_OPERATION:

if (s->e) s->instruction\_size = 4;

else s->instruction\_size = 3;

break;

case BLANK\_OPERATION:

s->instruction\_size = 3; break;

default:

fprintf(out, "error: operator unsupported\n");

return 0;

}

address += s->instruction\_size;

}

return 1;

}

int fill\_symbol\_table(FILE \*out, TranslationUnit \*tu) {

for (LinkedNode \*node = tu->statements->head->link;

node; node = node->link) {

SicStatement \*s = (SicStatement\*) node->value;

if (\*s->label) {

if (dict\_contains(tu->symbols, s->label)) {

fprintf(out, "error: duplicate symbol on line %d\n", s->line\_number);

return 0;

} else {

add\_to\_symbol\_table(tu->symbols, s->label, s->address);

}

}

}

return 1;

}

int validate\_statements(FILE \*out, TranslationUnit \*tu) {

int base = -1;

for (LinkedNode \*node = tu->statements->head->link;

node; node = node->link) {

SicStatement \*s = (SicStatement\*) node->value;

unsigned int absolute\_address;

s->base = base;

switch (s->type) {

case START: case END: case BYTE: case WORD: case RBYTE:

case RWORD: case REG\_OPERATION: case REG\_PAIR\_OPERATION:

case BLANK\_OPERATION:

break;

case BASE:

if (!find\_symbol\_address(

tu->symbols, s->operands.label, &absolute\_address)) {

fprintf(out, "error: undefined symbol on line %d\n", s->line\_number);

return 0;

}

base = absolute\_address;

break;

case NOBASE:

base = -1;

break;

case MEM\_OPERATION:

if (s->operands.is\_literal) {

absolute\_address = s->operands.literal;

if (!s->e) {

if (absolute\_address > 4096) {

fprintf(out, "error: requires extended instruction on line %d\n",

s->line\_number);

return 0;

}

}

s->operands.adjusted\_address = absolute\_address;

} else {

if (!find\_symbol\_address(

tu->symbols, s->operands.label, &absolute\_address)) {

fprintf(out, "error: undefined symbol on line %d\n", s->line\_number);

return 0;

}

if (s->e) s->operands.adjusted\_address = absolute\_address;

else {

int pc = absolute\_address -

(s->address + s->instruction\_size);

int b = absolute\_address - s->base;

if (-2048 <= pc && pc <= 2047) {

s->p = 1; s->b = 0;

s->operands.adjusted\_address = pc;

} else if (s->base != -1 && 0 <= b && b <= 4096) {

s->p = 0; s->b = 1;

s->operands.adjusted\_address = b;

} else {

fprintf(out, "error: requires extended instruction on line %d\n",

s->line\_number);

return 0;

}

}

}

break;

default:

return 0;

}

}

return 1;

}

static void fflush\_text\_record\_line(FILE \*out, HashTable \*opcodes,

LinkedNode \*start\_node, LinkedNode \*end\_node) {

for (LinkedNode \*node = start\_node; node != end\_node; node = node->link)

fprint\_statement\_in\_hex(out, (SicStatement\*) node->value, opcodes);

}

#define TEXT\_RECORD\_SIZE 0x20

void fprint\_text\_record(FILE \*out, TranslationUnit \*tu) {

LinkedNode \*line\_start\_node;

int address = 0;

int line\_start\_address = 0;

int length = 0;

line\_start\_node = tu->statements->head->link;

for (LinkedNode \*node = line\_start\_node; node; node = node->link) {

SicStatement \*s = (SicStatement\*) node->value;

int size = s->instruction\_size;

if (size) {

int is\_reserved = (s->type == RWORD || s->type == RBYTE);

int newline = length != 0 && length + size >= TEXT\_RECORD\_SIZE;

if ((is\_reserved && length) || newline ) {

fprintf(out, "T");

fprintf(out, "%06X", line\_start\_address);

fprintf(out, "%02X", length);

fflush\_text\_record\_line(

out, tu->opcodes, line\_start\_node, node);

fprintf(out, "\n");

line\_start\_address = address;

line\_start\_node = node;

length = 0;

}

if (!is\_reserved) length += size;

address += size;

}

}

if (length) {

fprintf(out, "T");

fprintf(out, "%06X", line\_start\_address);

fprintf(out, "%02X", length);

fflush\_text\_record\_line(out, tu->opcodes, line\_start\_node, NULL);

fprintf(out, "\n");

}

}

void fprint\_end\_record(FILE \*out, TranslationUnit \*tu) {

fprintf(out, "E");

for (LinkedNode \*node = tu->statements->head->link; node;

node = node->link) {

SicStatement \*statement = (SicStatement\*) node->value;

if (statement->type == END) {

unsigned int exec\_address;

find\_symbol\_address(tu->symbols,

statement->operands.label, &exec\_address);

fprintf(out, "%06X", exec\_address);

fprintf(out, "\n");

return;

}

}

}

void fprint\_modificaiton\_record(FILE \*out, TranslationUnit \*tu) {

for (LinkedNode \*node = tu->statements->head->link; node;

node = node->link) {

SicStatement \*statement = (SicStatement\*) node->value;

if (statement->e) {

if (!statement->operands.is\_literal) {

fprintf(out, "M");

fprintf(out, "%06X", statement->address + 1);

fprintf(out, "%02X", 5);

fprintf(out, "\n");

}

}

}

}

void fprint\_lst\_line(FILE \*out, SicStatement \*statement, HashTable \*opcodes) {

fprintf(out, "%3d", statement->line\_number);

if (statement->type == BASE || statement->type == NOBASE)

fprintf(out, " %4s", "");

else

fprintf(out, " %04X", statement->address);

fprintf(out, " %-6s", \*statement->label ? statement->label : "");

fprintf(out, " ");

if (statement->e) fprintf(out, "+");

else fprintf(out, " ");

fprintf(out, "%-9s", statement->reserved\_string);

char key = ' ';

if (statement->type == MEM\_OPERATION) {

if (statement->operands.mode == INDIRECT) key = '@';

else if (statement->operands.mode == IMMEDIATE) key = '#';

}

fprintf(out, " ");

fprintf(out, "%c", key);

int length;

char \*operand;

switch (statement->type) {

case MEM\_OPERATION:

if (statement->operands.is\_literal) {

fprintf(out, "%-12d", statement->operands.literal);

} else {

length = strlen(statement->operands.label);

operand = malloc(sizeof(char) \* (length + 3));

strcpy(operand, statement->operands.label);

if (statement->operands.x) strcat(operand, ",X");

fprintf(out, "%-12s", operand);

free(operand);

}

break;

case REG\_OPERATION:

fprintf(out, "%-12s", register\_to\_string(statement->operands.r));

break;

case REG\_PAIR\_OPERATION:

operand = malloc(sizeof(char) \* 10);

strcpy(operand, register\_to\_string(statement->operands.r1));

strcat(operand, ",");

strcat(operand, register\_to\_string(statement->operands.r2));

fprintf(out, "%-12s", operand);

free(operand);

break;

case BYTE:

if (statement->operands.is\_b) {

length = statement->operands.ulength;

operand = malloc(sizeof(unsigned char) \* (length \* 2 + 4));

strcpy(operand, "X'");

strcat\_byte(operand, statement, 0);

strcat(operand, "'");

} else {

length = strlen(statement->operands.bytes);

operand = malloc(sizeof(char) \* (length \* 2 + 4));

strcpy(operand, "C'");

strcat\_byte(operand, statement, 0);

strcat(operand, "'");

}

fprintf(out, "%-12s", operand);

free(operand);

break;

case WORD:

fprintf(out, "%-12d", statement->operands.word);

break;

case RBYTE:

fprintf(out, "%-12d", statement->operands.rsize);

break;

case RWORD:

fprintf(out, "%-12d", statement->operands.rsize / 3);

break;

case START:

fprintf(out, "%-12d", statement->operands.start\_address);

break;

case END:

fprintf(out, "%-12s", statement->operands.label);

break;

case BASE: case NOBASE: case BLANK\_OPERATION:

fprintf(out, "%12s", "");

break;

case REG\_N\_OPERATION: case N\_OPERATION:

assert(0 && "unsupported operator");

break;

}

fprint\_statement\_in\_hex(out, statement, opcodes);

}

void fprint\_statement\_in\_hex(FILE \*out, SicStatement \*statement,

HashTable \*opcodes) {

unsigned char opcode;

if (statement->type & RESERVED\_OPERATION\_FLAG) {

if (!find\_from\_hash\_table(opcodes, statement->reserved\_string,

&opcode)) {

assert(0 && "couldn't find operation");

}

}

char buffer[SIC\_MAX\_BYTE\_LENGTH] = { 0 };

switch (statement->type) {

case MEM\_OPERATION:

switch (statement->operands.mode) {

case IMMEDIATE: opcode = opcode | 1; break;

case INDIRECT: opcode = opcode | 2; break;

case SIMPLE: opcode = opcode | 3; break;

}

fprintf(out, "%02X", opcode);

int address = statement->operands.adjusted\_address;

if (statement->e) {

address = address | (1 << 20); // set e flag

// set x flog

if (statement->operands.x) address = address | (1 << 23);

fprintf(out, "%06X", address);

} else {

if (statement->p) {

int signed\_bit = address < 0;

if (address < 0) {

address = address & ((1 << 11) - 1);

}

address = address | (signed\_bit << 11);

address = address | (1 << 13); // set p flag

} else if (statement->b) {

address = address | (1 << 14); // set b flag

}

// set x flog

if (statement->operands.x) address = address | (1 << 15);

fprintf(out, "%04X", address);

}

break;

case REG\_OPERATION:

fprintf(out, "%02X", opcode);

fprintf(out, "%1X", statement->operands.r);

fprintf(out, "0");

break;

case REG\_PAIR\_OPERATION:

fprintf(out, "%02X", opcode);

fprintf(out, "%1X", statement->operands.r1);

fprintf(out, "%1X", statement->operands.r2);

break;

case BLANK\_OPERATION:

opcode = opcode | 3; // SIMPLE mode by default

fprintf(out, "%02X", opcode);

fprintf(out, "0000");

break;

case REG\_N\_OPERATION: case N\_OPERATION:

assert(0 && "unsupported operator");

break;

case BYTE:

strcat\_byte(buffer, statement, 1);

fprintf(out, "%s", buffer);

break;

case WORD:

fprintf(out, "%04X", statement->operands.word);

break;

default:

break;

}

}

void fprint\_header\_record(FILE \*out, TranslationUnit \*tu) {

fprintf(out, "H");

int last\_address = 0;

for (LinkedNode \*node = tu->statements->head->link; node;

node = node->link) {

SicStatement \*statement = (SicStatement\*) node->value;

if (statement->type == START) {

fprintf(out, "%-6s", statement->label);

fprintf(out, "%06X", statement->operands.start\_address);

}

last\_address = statement->address;

}

fprintf(out, "%06X", last\_address);

fprintf(out, "\n");

}

#define SSCAN\_WORD\_BUFFER\_SIZE 33

#define SSCAN\_WORD\_NO\_COMMA 0

#define SSCAN\_WORD\_REQUIRE\_COMMA 1

static char \*sscan\_word(char \*\*cursor, int require\_comma\_separator) {

char buffer[SSCAN\_WORD\_BUFFER\_SIZE];

char \*buffer\_cursor = buffer;

char \*buffer\_end = buffer + SSCAN\_WORD\_BUFFER\_SIZE;

if (require\_comma\_separator) {

if (!skip\_whitespaces\_and\_comma(cursor))

return 0;

} else skip\_whitespaces(cursor);

for (; !is\_whitespace\_or\_comma(\*\*cursor) && \*\*cursor; (\*cursor)++, buffer\_cursor++) {

if (buffer\_cursor == buffer\_end - 1)

return NULL;

\*buffer\_cursor = \*\*cursor;

}

\*buffer\_cursor = '\0';

if (buffer\_cursor == buffer) return NULL;

char \*string = malloc(sizeof(char) \* (buffer\_cursor - buffer + 1));

strcpy(string, buffer);

return string;

}

#define SSCAN\_NO\_COMMA 0

#define SSCAN\_REQUIRE\_COMMA 1

#define SSCAN\_INT\_NONE -1

#define SSCAN\_INT\_NEGATIVE -2

static int sscan\_int(char \*\*cursor, int require\_comma\_separator) {

int n, offset;

if (require\_comma\_separator) {

if (!skip\_whitespaces\_and\_comma(cursor))

return SSCAN\_INT\_NONE;

} else skip\_whitespaces(cursor);

if (sscanf(\*cursor, "%d%n", &n, &offset) != 1) {

return SSCAN\_INT\_NONE;

}

if (n < 0) return SSCAN\_INT\_NEGATIVE;

(\*cursor) += offset;

if (is\_whitespace\_or\_comma(\*\*cursor)) return SSCAN\_INT\_NONE; // trailing chars

return n;

}

static int sscan\_register(

char \*\*cursor, Register \*reg, int require\_comma\_separator) {

char \*word = sscan\_word(cursor, require\_comma\_separator);

if (!word) return 0;

char \*strings[8] = {

"A", "X", "L", "B", "S", "T", "PC", "SW" };

Register registers[8] = {

A, X, L, B, S, T, PC, SW };

for (int i = 0; i < 8; ++i) {

if (!strcmp(strings[i], word)) {

\*reg = registers[i];

free(word);

return 1;

}

}

free(word);

return 0;

}

static int read\_memory\_operand(char \*word, SicStatement \*statement) {

switch (\*word) {

case '@': statement->operands.mode = INDIRECT; word++; break;

case '#': statement->operands.mode = IMMEDIATE; word++; break;

default: statement->operands.mode = SIMPLE;

}

int literal = -1;

if (statement->operands.mode == IMMEDIATE)

// may be literal if immediate

literal = sscan\_int(&word, SSCAN\_NO\_COMMA);

if (literal == SSCAN\_INT\_NEGATIVE) return 0;

if (literal == SSCAN\_INT\_NONE) {

if (strlen(word) >= SIC\_LABEL\_LENGTH) return 0;

strcpy(statement->operands.label, word);

statement->operands.is\_literal = 0;

} else {

statement->operands.literal = literal;

statement->operands.is\_literal = 1;

}

return 1;

}

static inline int isupperxdigit(char c) {

return toupper(c) == c && isxdigit(c);

}

static int sscan\_bytes(char \*\*cursor, SicStatement \*statement) {

int is\_bytes; // or chars

skip\_whitespaces(cursor);

switch (\*((\*cursor)++)) {

case 'X': is\_bytes = 1; break;

case 'C': is\_bytes = 0; break;

default: return 0;

}

if (\*((\*cursor)++) != '\'') return 0;

char bytes[SIC\_MAX\_BYTE\_LENGTH] = { 0 };

unsigned char ubytes[SIC\_MAX\_BYTE\_LENGTH] = { 0 };

int index = 0;

statement->operands.is\_b = is\_bytes;

if (is\_bytes) {

int odd = 1;

while (\*\*cursor != '\'') {

if (!\*\*cursor) return 0;

if (isupperxdigit(\*\*cursor)) {

if (index == SIC\_MAX\_BYTE\_LENGTH - 1) return 0;

int n; sscanf((\*cursor)++, "%1X", &n);

if (odd) ubytes[index] += 16 \* n;

else ubytes[index++] += n;

odd = !odd;

} else {

return 0;

}

}

// if odd == TRUE, the number of hexes is even

if (!odd) index++;

statement->operands.is\_odd\_b = !odd;

} else {

while (\*\*cursor != '\'') {

if (!\*\*cursor) return 0;

if (index == SIC\_MAX\_BYTE\_LENGTH - 1) return 0;

bytes[index++] = \*((\*cursor)++);

}

bytes[index] = '\0';

}

if (\*((\*cursor)++) != '\'') return 0;

skip\_whitespaces(cursor);

if (\*\*cursor != '\0') return 0;

if (is\_bytes) {

statement->operands.ubytes = malloc(sizeof(unsigned char));

statement->operands.ulength = index;

for (int i = 0; i < index; ++i)

statement->operands.ubytes[i] = ubytes[i];

} else {

statement->operands.bytes = malloc(sizeof(char) \* (strlen(bytes) + 1));

strcpy(statement->operands.bytes, bytes);

}

return 1;

}

static int sscan\_operand(char \*\*asm\_line\_cursor, SicStatement \*statement) {

// In all asm lines, the reserved token is preceded by an appropriate

// operand for the ReservedType within statement

char \*first\_word;

char \*second\_word;

int n;

switch (statement->type) {

case MEM\_OPERATION:

first\_word = sscan\_word(asm\_line\_cursor, SSCAN\_WORD\_NO\_COMMA);

if (!first\_word) return 0;

if (!read\_memory\_operand(first\_word, statement)) {

free(first\_word); return 0;

}

// no ,X

skip\_whitespaces(asm\_line\_cursor);

if (\*\*asm\_line\_cursor == '\0') {

statement->operands.x = 0; break;

}

// must have , X

second\_word = sscan\_word(asm\_line\_cursor, SSCAN\_WORD\_REQUIRE\_COMMA);

if (!second\_word) return 0;

if (strcmp(second\_word, "X")) {

free(second\_word);

return 0;

}

statement->operands.x = 1;

free(second\_word);

break;

case REG\_OPERATION:

if (!sscan\_register(asm\_line\_cursor,

&statement->operands.r, SSCAN\_NO\_COMMA))

return 0;

break;

case REG\_PAIR\_OPERATION:

if (!sscan\_register(asm\_line\_cursor,

&statement->operands.r1, SSCAN\_NO\_COMMA))

return 0;

if (!sscan\_register(asm\_line\_cursor,

&statement->operands.r2, SSCAN\_REQUIRE\_COMMA))

return 0;

break;

case BLANK\_OPERATION: break;

case BYTE:

if (!sscan\_bytes(asm\_line\_cursor, statement))

return 0;

break;

case WORD:

if ((n = sscan\_int(asm\_line\_cursor, SSCAN\_NO\_COMMA)) < 0)

return 0;

statement->operands.word = n;

break;

case RBYTE:

if ((n = sscan\_int(asm\_line\_cursor, SSCAN\_NO\_COMMA)) < 0)

return 0;

statement->operands.rsize = n;

break;

case RWORD:

if ((n = sscan\_int(asm\_line\_cursor, SSCAN\_NO\_COMMA)) < 0)

return 0;

statement->operands.rsize = n \* 3;

break;

case START:

if ((n = sscan\_int(asm\_line\_cursor, SSCAN\_NO\_COMMA)) < 0) {

return 0;

}

statement->operands.start\_address = n;

break;

case END:

first\_word = sscan\_word(asm\_line\_cursor, SSCAN\_WORD\_NO\_COMMA);

if (first\_word) {

if (strlen(first\_word) >= SIC\_LABEL\_LENGTH) return 0;

strcpy(statement->operands.label, first\_word);

free(first\_word);

} else statement->operands.label[0] = '\0';

break;

case BASE:

first\_word = sscan\_word(asm\_line\_cursor, SSCAN\_WORD\_NO\_COMMA);

if (strlen(first\_word) >= SIC\_LABEL\_LENGTH) return 0;

strcpy(statement->operands.label, first\_word);

free(first\_word);

break;

case NOBASE: break;

default:

printf("Unsupported operand\n");

return 0;

}

// check for trailing text

skip\_whitespaces(asm\_line\_cursor);

return \*\*asm\_line\_cursor == '\0';

}

static inline int is\_reserved(char \*word, ReservedDict \*reserved) {

if (\*word == '+') return dict\_contains(reserved, word + 1);

else return dict\_contains(reserved, word);

}

static int read\_reserved(

char \*word, SicStatement \*statement, ReservedDict \*reserved) {

if (\*word == '+') {

// this instruction is extended so this must be a memory operator

if (find\_reserved\_type(reserved, word + 1, &statement->type)) {

if (statement->type == MEM\_OPERATION) {

statement->reserved\_string = malloc(

sizeof(char) \* (strlen(word) - 1));

strcpy(statement->reserved\_string, word + 1);

statement->e = 1;

free(word);

return 1;

}

}

}

else {

if (find\_reserved\_type(reserved, word, &statement->type)) {

statement->reserved\_string = word;

statement->e = 0;

return 1;

}

}

return 0;

}

static int sscan\_label\_and\_reserved(

char \*\*asm\_line\_cursor, SicStatement \*statement,

ReservedDict \*reserved) {

// All asm lines start with an optional label and a required

// reserved token like an operand or START, RBYTE etc.

char \*first\_word = sscan\_word(asm\_line\_cursor, SSCAN\_WORD\_NO\_COMMA);

if (!first\_word) return 0;

if (is\_reserved(first\_word, reserved)) {

// we found a reserved token, so there is no label

return read\_reserved(first\_word, statement, reserved);

}

// else -> save the label

if (strlen(first\_word) >= SIC\_LABEL\_LENGTH) return 0;

strcpy(statement->label, first\_word);

free(first\_word);

char \*second\_word = sscan\_word(asm\_line\_cursor, SSCAN\_WORD\_NO\_COMMA);

if (!second\_word) return 0;

// now we need to find a reserved token

if (is\_reserved(second\_word, reserved)) {

return read\_reserved(second\_word, statement, reserved);

}

free(second\_word);

return 0;

}

static SicStatement \*add\_statement(SicStatementList \*list) {

SicStatement \*statement = calloc(sizeof(SicStatement), 1);

statement->reserved\_string = NULL;

statement->operands.bytes = NULL;

statement->p = 0;

statement->b = 0;

statement->e = 0;

add\_to\_list(list, (void\*) statement);

return statement;

}

static void strcat\_byte(char \*string, SicStatement \*statement, int raw) {

if (statement->operands.is\_b) {

char hex[3];

for (int i = 0; i < statement->operands.ulength; ++i) {

if (!raw && statement->operands.is\_odd\_b &&

i == statement->operands.ulength - 1) {

sprintf(hex, "%1X", statement->operands.ubytes[i] / 16);

} else {

sprintf(hex, "%02X", statement->operands.ubytes[i]);

}

strcat(string, hex);

}

} else {

if (raw) {

for (char \*c = statement->operands.bytes; \*c; ++c) {

char hex[3];

sprintf(hex, "%02X", \*c);

strcat(string, hex);

}

} else {

strcat(string, statement->operands.bytes);

}

}

}

static inline int is\_whitespace(char c) {

return c == ' ' || c == '\t';

}

static inline int is\_whitespace\_or\_comma(char c) {

return c == ' ' || c == '\t' || c == ',';

}

static int read\_line(FILE \*in, char \*\*string) {

int is\_start\_of\_line = 1;

int is\_comment\_part = 0;

int c;

char buffer[4096] = { 0 };

char \*buffer\_cursor = buffer;

char \*buffer\_end = buffer + 1024;

if (feof(in)) {

\*string = NULL;

return READ\_LINE\_EOF;

}

while (1) {

c = fgetc(in);

if (c == '\n' || c == EOF) {

if (is\_start\_of\_line) {

\*string = NULL;

return READ\_LINE\_BLANK;

} else {

\*string = malloc(sizeof(char) \* (strlen(buffer) + 1));

strcpy(\*string, buffer);

return READ\_LINE\_SUCCESS;

}

}

if (is\_comment\_part) {

continue;

} else {

if (c == '.') {

is\_comment\_part = 1; continue;

}

if (is\_start\_of\_line && !is\_whitespace(c)) is\_start\_of\_line = 0;

if (!is\_start\_of\_line) {

if (buffer\_cursor == buffer\_end) return READ\_LINE\_INVALID;

\*buffer\_cursor++ = c;

}

}

}

}

static inline void skip\_whitespaces(char \*\*cursor) {

while (is\_whitespace(\*\*cursor) && \*\*cursor) {

(\*cursor) += 1;

}

}

static inline int skip\_whitespaces\_and\_comma(char \*\*cursor) {

int passed\_comma = 0;

while (\*\*cursor && is\_whitespace\_or\_comma(\*\*cursor)) {

if (\*\*cursor == ',') {

if (passed\_comma) return 0;

else passed\_comma = 1;

}

(\*cursor) += 1;

}

return passed\_comma;

}

#include <stdio.h>

#include <stdlib.h>

#include "assemble.h"

#include "hashtable.h"

#include "reserved.h"

#include "symbol.h"

#include "asm\_helper.h"

#include "global.h"

#include "string.h"

/\* Recall these definitions

typedef struct \_TranslationUnit {

HashTable \*opcodes;

ReservedDict \*reserved;

SicStatementList \*statements;

SymbolTable \*symbols;

} TranslationUnit;

\*/

static TranslationUnit \*tu = NULL;

int assemble\_1(FILE \*out, ParsedCommand \*pc) {

if (tu) {

free\_translation\_unit(tu);

tu = NULL;

}

FILE \*in;

if (!(in = fopen(pc->arguments[0], "r"))) {

fprintf(out, "error: could not open file for assembly\n");

return 0;

}

tu = translate(out, in, G.table, G.reserved);

fclose(in);

if (!tu) return 0;

char lst[1000];

char obj[1000];

strcpy(lst, pc->arguments[0]);

strcpy(obj, pc->arguments[0]);

strcpy(lst + strlen(pc->arguments[0]) - 4, ".lst");

strcpy(obj + strlen(pc->arguments[0]) - 4, ".obj");

FILE \*lst\_out = fopen(lst, "w");

FILE \*obj\_out = fopen(obj, "w");

generate\_lst(lst\_out, tu);

generate\_obj(obj\_out, tu);

fclose(lst\_out);

fclose(obj\_out);

fprintf(out, "output file : [%s], [%s]\n", lst, obj);

return 1;

}

int symbol\_0(FILE \*out, ParsedCommand \*pc) {

if (tu)

fprint\_symbols(out, tu->symbols);

else

fprintf(out, "there is no valid symbol table to print out.\n");

return 1;

}

static TranslationUnit \*new\_translation\_unit(

FILE \*in, HashTable \*opcodes, ReservedDict \*reserved) {

TranslationUnit \*tu = malloc(sizeof(TranslationUnit));

tu->opcodes = opcodes;

tu->reserved = reserved;

tu->statements = new\_statement\_list();

tu->symbols = new\_symbol\_table();

return tu;

}

TranslationUnit \*translate(

FILE \*out, FILE \*in, HashTable \*opcodes, ReservedDict \*reserved) {

TranslationUnit \*tu = new\_translation\_unit(in, opcodes, reserved);

if (!read\_asm(out, in, tu)) return NULL;

if (!assign\_addresses(out, tu)) return NULL;

if (!fill\_symbol\_table(out, tu)) return NULL;

if (!validate\_statements(out, tu)) return NULL;

return tu;

}

void free\_translation\_unit(TranslationUnit \*tu) {

free\_statement\_list(tu->statements);

free\_symbol\_table(tu->symbols);

free(tu);

}

void generate\_obj(FILE \*out, TranslationUnit \*tu) {

fprint\_header\_record(out, tu);

fprint\_text\_record(out, tu);

fprint\_modificaiton\_record(out, tu);

fprint\_end\_record(out, tu);

}

void generate\_lst(FILE \*out, TranslationUnit \*tu) {

for (LinkedNode \*node = tu->statements->head->link; node;

node = node->link) {

fprint\_lst\_line(out, (SicStatement\*) node->value, tu->opcodes);

fprintf(out, "\n");

}

}

#include "file.h"

#include <stdio.h>

#include <dirent.h>

#include <sys/stat.h>

#ifdef TEST

// Manual unit test for this module

// Refer to the README for testing instructions

int main(void) {

printf(">> Print dir:\n");

fprint\_dir(stdout);

printf(">> Print file.h:\n");

if (!fprint\_file(stdout, "file.h"))

printf("Error opening file!\n");

printf(">> Print hoho.h:\n");

if (!fprint\_file(stdout, "hoho.h"))

printf("Error opening file!\n");

}

#endif

void fprint\_dir(FILE \*out) {

struct dirent \*entry;

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

// For each file in current directory

while ((entry = readdir(directory))) {

char \*filename = entry->d\_name;

struct stat buffer;

stat(filename, &buffer);

if (filename[0] == '.') continue;

if (entry->d\_type == DT\_DIR) {

fprintf(out, "%s/\t", filename);

continue;

}

fprintf(out, "%s", filename);

if (entry->d\_type == DT\_DIR) fprintf(out, "\\");

else if (buffer.st\_mode & S\_IXUSR) fprintf(out, "\*");

fprintf(out, "\t");

}

fprintf(out, "\n");

}

int fprint\_file(FILE \*out, char \*filename) {

FILE \*file;

// Open file and return failure if there was a problem

if (!(file = fopen(filename, "r")))

return 0;

if (feof(file))

return 0;

// Print all characters from 'file' to output stream

char c;

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

fprintf(out, "%c", c);

return 1;

}

#include "generic\_dict.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <assert.h>

/\* Recall these definitions

typedef struct \_BucketNode {

char \*key;

void \*value;

struct \_BucketNode \*link;

} BucketNode;

typedef struct \_DictNode {

BucketNode \*head;

BucketNode \*last;

} DictNode;

typedef struct \_Dict {

int size;

DictNode \*nodes;

} Dict;

\*/

#ifdef TEST

void print\_string(FILE \*out, void \*string) {

fprintf(out, "%s", (char\*) string);

}

int main(void) {

Dict \*dict = new\_dict(20);

add\_to\_dict(dict, "hello", "Yeah, hi");

add\_to\_dict(dict, "bye", "Cya");

char \*string;

assert(find\_from\_dict(dict, "hello", (void\*) &string));

assert(!strcmp("Yeah, hi", string));

assert(find\_from\_dict(dict, "bye", (void\*) &string));

assert(!strcmp("Cya", string));

assert(dict\_contains(dict, "hello"));

assert(dict\_contains(dict, "bye"));

assert(!dict\_contains(dict, "hoho"));

assert(!dict\_contains(dict, "Hello"));

free\_dict(dict, ignore\_dict\_values);

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

printf("Automatic tests successful!\n");

{

printf("\n");

Dict \*dict = new\_dict(20);

dict = new\_dict(20);

add\_to\_dict(dict, "hello", "yeah, hi");

add\_to\_dict(dict, "bye", "Cya");

printf("Print function\n");

fprint\_dict(stdout, dict, print\_string);

free\_dict(dict, ignore\_dict\_values);

}

return 0;

}

#endif

Dict \*new\_dict(int size) {

Dict \*dict= malloc(sizeof(Dict));

dict->size = size;

dict->nodes = malloc(sizeof(DictNode) \* size);

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

dict->nodes[i].head = malloc(sizeof(BucketNode));

dict->nodes[i].head->key = NULL;

dict->nodes[i].head->value = NULL;

dict->nodes[i].head->link = NULL;

dict->nodes[i].last = dict->nodes[i].head;

}

return dict;

}

static inline int hash(char \*key, int table\_size) {

unsigned int hash\_value = 0;

for (; \*key; ++key) {

hash\_value = hash\_value << 1;

hash\_value ^= \*key;

}

return hash\_value % table\_size;

}

void free\_dict(Dict \*dict, void (\*free\_value) (void \*value)) {

for (int i = 0; i < dict->size; ++i) {

BucketNode \*next;

for (BucketNode \*node = dict->nodes[i].head; node; node = next) {

next = node->link;

free\_value(node->value);

free(node->key);

free(node);

}

}

free(dict->nodes);

free(dict);

}

void add\_to\_dict(Dict \*dict, char \*key, void \*value) {

int h = hash(key, dict->size);

BucketNode \*node = malloc(sizeof(BucketNode));

node->key = malloc(sizeof(char) \* (strlen(key) + 1));

strcpy(node->key, key);

node->link = NULL;

node->value = value;

dict->nodes[h].last->link = node;

dict->nodes[h].last = node;

}

int find\_from\_dict(Dict \*dict, char \*key, void \*\*value) {

int h = hash(key, dict->size);

for (BucketNode \*node = dict->nodes[h].head->link; node; node = node->link)

if (!(strcmp(key, node->key))) {

\*value = node->value;

return 1;

}

return 0;

}

int dict\_contains(Dict \*dict, char \*key) {

int h = hash(key, dict->size);

for (BucketNode \*node = dict->nodes[h].head->link; node; node = node->link)

if (!(strcmp(key, node->key))) {

return 1;

}

return 0;

}

void fprint\_dict(FILE \*out, Dict \*dict, void (\*print\_value) (FILE \*out, void \*value)) {

for (int i = 0; i < dict->size; ++i) {

fprintf(out, "%d : ", i);

for (BucketNode \*node = dict->nodes[i].head->link; node; node = node->link) {

fprintf(out, "[%s,", node->key);

print\_value(out, node->value);

fprintf(out, "]");

if (node->link) fprintf(out, " -> ");

}

fprintf(out, "\n");

}

}

void ignore\_dict\_values(void \*value) {

(void) value; // suppress unused warning

}

#include "generic\_list.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <assert.h>

/\* Recall these definitions

typedef struct \_LinkedNode {

void \*value;

struct \_LinkedNode \*link;

} LinkedNode;

typedef struct \_List {

LinkedNode \*head;

LinkedNode \*last;

} List;

\*/

#ifdef TEST

void print\_string(FILE \*out, void \*string) {

fprintf(out, "%s", (char\*) string);

}

int main(void) {

List \*list = new\_list();

add\_to\_list(list, "Hello");

add\_to\_list(list, "Yo");

assert(!strcmp(list->head->link->value, "Hello"));

assert(!strcmp(list->head->link->link->value, "Yo"));

free\_list(list, ignore\_list\_values);

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

printf("Automatic tests successful!\n");

{

printf("\n");

List \*list = new\_list();

add\_to\_list(list, "Hello");

add\_to\_list(list, "Yo");

fprint\_list(stdout, list, print\_string);

free\_list(list, ignore\_list\_values);

}

return 0;

}

#endif

List \*new\_list() {

List \*list = malloc(sizeof(List));

LinkedNode \*node = malloc(sizeof(LinkedNode));

node->link = NULL;

node->value = NULL;

list->head = node;

list->last = list->head;

return list;

}

void add\_to\_list(List \*list, void \*value) {

LinkedNode \*node = malloc(sizeof(LinkedNode));

node->value = value;

node->link = NULL;

list->last->link = node;

list->last = node;

}

void free\_list(List \*list, void (\*free\_value) (void \*value)) {

LinkedNode \*next;

for (LinkedNode \*node = list->head; node; node = next) {

next = node->link;

if (node != list->head) free\_value(node->value);

free(node);

}

}

void fprint\_list(FILE \*out, List \*list, void (\*print\_value) (FILE \*out, void \*value)) {

for (LinkedNode \*node = list->head->link; node; node = node->link) {

print\_value(out, node->value);

if (node->link)

printf(" -> ");

}

printf("\n");

}

void ignore\_list\_values(void \*value) {

// suppress unused warning

(void) value;

}

#include "hashtable.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define OPCODE\_TABLE\_SIZE 20

/\* Recall these definitions

typedef struct \_Opcode {

unsigned char raw;

} Opcode;

#define HashTable Dict;

\*/

#ifdef TEST

#undef TEST

#include "generic\_dict.c"

// Manual unit test for this module

// Refer to the README for testing instructions

int main(void) {

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

HashTable \*table = new\_hash\_table();

add\_to\_hash\_table(table, "Hello!", 12);

add\_to\_hash\_table(table, "Hoho!", 23);

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

char ho[3] = { 0 };

ho[0] = 32 + i;

ho[1] = 32 + i;

add\_to\_hash\_table(table, ho, i);

}

printf("Print hash table\n");

fprint\_hash\_table(stdout, table);

printf("\n");

printf("Find non-exisitng\n");

unsigned char opcode;

if (find\_from\_hash\_table(table, "Non", &opcode))

printf("> Found something..? FAIL\n");

else

printf("> Yes! We couldn't find anything\n");

printf("\n");

printf("Find Hello!\n");

if (find\_from\_hash\_table(table, "Hello!", &opcode))

printf("> Found this: %02X\n", opcode);

else

printf("> Couldn't find anything..? FAIL\n");

free\_hash\_table(table);

}

return 0;

}

#endif

HashTable \*new\_hash\_table() {

Dict \*dict = new\_dict(OPCODE\_TABLE\_SIZE);

return dict;

}

void free\_hash\_table(HashTable \*table) {

free\_dict(table, free);

}

void add\_to\_hash\_table(HashTable \*table, char \*key, unsigned char opcode) {

Opcode \*value = malloc(sizeof(Opcode));

value->raw = opcode;

add\_to\_dict(table, key, (void\*) value);

}

int find\_from\_hash\_table(HashTable \*table, char \*key, unsigned char \*opcode) {

Opcode \*value;

if (find\_from\_dict(table, key, (void\*\*) &value)) {

\*opcode = value->raw; return 1;

} else return 0;

}

void fprint\_opcode(FILE \*out, void \*opcode\_void) {

Opcode \*opcode = (Opcode\*) opcode\_void;

fprintf(out, "%02X", opcode->raw);

}

void fprint\_hash\_table(FILE \*out, HashTable \*table) {

fprint\_dict(out, table, fprint\_opcode);

}

#include "help.h"

#include <stdio.h>

#include <stdlib.h>

#ifdef TEST

// Manual unit test for this module

// Refer to the README for testing instructions

int main(void) {

fprint\_help(stdout);

return 0;

}

#endif

void fprint\_help(FILE \*out) {

const int HELP\_COUNT = 13;

const char \*HELP\_STRINGS[] = {

"h[elp]", "d[ir]",

"q[uit]", "hi[story]",

"du[mp] [start, end]", "e[dit] address, value",

"f[ill] start, end, value", "reset",

"opcode mnemonic", "opcodelist",

"assemble filename", "type filename",

"symbol"

};

for (int i = 0; i < HELP\_COUNT; ++i)

fprintf(out, "%s\n", HELP\_STRINGS[i]);

}

#include "history.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

/\* Recall these definitions

#define COMMAND\_LENGTH 120

typedef struct \_HistoryNode {

struct \_HistoryNode \*next;

char string[HISTORY\_LENGTH];

} HistoryNode;

typedef struct \_History {

HistoryNode \*head; // fake head

HistoryNode \*last;

} History;

\*/

#ifdef TEST

// Manual unit test for this module

// Refer to the README for testing instructions

int main(void) {

for (int i = 0; i < 20; ++i) {

History \*history = new\_history();

add\_history(history, "Hello!");

add\_history(history, "Yo!");

add\_history(history, "Hello!");

add\_history(history, "Yo!");

add\_history(history, "Hello!");

add\_history(history, "Yo!");

fprint\_history(stdout, history);

free\_history(history);

}

return 0;

}

#endif

// Make sure to free after use!

History \*new\_history() {

History \*history = malloc(sizeof(History));

history->head = malloc(sizeof(HistoryNode));

history->head->next = NULL;

history->last = history->head;

return history;

}

void free\_history(History \*history) {

HistoryNode \*next;

for (HistoryNode \*node = history->head; node; node = next) {

next = node->next;

free(node);

}

free(history);

}

void add\_history(History \*history, char \*string) {

HistoryNode \*new\_node = malloc(sizeof(HistoryNode));

strcpy(new\_node->string, string);

new\_node->next = NULL;

history->last->next = new\_node;

history->last = new\_node;

}

void fprint\_history(FILE \*out, History \*history) {

int i = 1;

for (HistoryNode \*node = history->head->next; node; node = node->next, ++i)

fprintf(out, "%d\t%s\n", i, node->string);

}

int has\_history(History \*history) {

return history->head != history->last;

}

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <assert.h>

#include "interpreter.h"

/\* Note these declarations

typedef struct \_OperationNode {

char operator[TOKEN\_LENGTH];

int argument\_count;

int (\*function)(FILE\*, ParsedCommand\*);

struct \_OperationNode \*link;

} OperationNode;

typedef struct \_Interpreter {

OperationNode \*head;

OperationNode \*last;

FILE \*output\_stream;

} Interpreter;

\*/

#ifdef TEST

#undef TEST

#include "parser.c"

int f1(FILE \*out, ParsedCommand \*pc) {

return 1;

}

int f2(FILE \*out, ParsedCommand \*pc) {

return 2;

}

int f3(FILE \*out, ParsedCommand \*pc) {

return 3;

}

int main(void) {

Interpreter \*ip = new\_interpreter(stdout);

ParsedCommand \*pc;

int error\_code;

add\_operation(ip, "hey", 0, f1);

add\_operation(ip, "ho", 1, f2);

add\_operation(ip, "ho", 3, f3);

pc = parse\_command("hey \t", &error\_code);

assert(interpret\_and\_free(ip, pc) == 1);

pc = parse\_command("ho ajif \t", &error\_code);

assert(interpret\_and\_free(ip, pc) == 2);

pc = parse\_command("ho \t fds, \t d2", &error\_code);

assert(interpret\_and\_free(ip, pc) == 0);

pc = parse\_command("ho \t fds, \t d2 , \t ji", &error\_code);

assert(interpret\_and\_free(ip, pc) == 3);

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

printf("Automatic tests successful!\n");

free\_interpreter(ip);

return 0;

}

#endif

Interpreter \*new\_interpreter(FILE \*out) {

Interpreter \*ip = malloc(sizeof(Interpreter));

ip->head = malloc(sizeof(OperationNode));

ip->last = ip->head;

ip->head->link = NULL;

ip->output\_stream = out;

return ip;

}

void free\_interpreter(Interpreter \*ip) {

OperationNode \*next;

for (OperationNode \*node = ip->head; node; node = next) {

next = node->link;

free(node);

}

free(ip);

}

void add\_operation(Interpreter \*ip, char \*operator,

int argument\_count, void \*function) {

OperationNode \*node = malloc(sizeof(OperationNode));

strcpy(node->operator, operator);

node->argument\_count = argument\_count;

node->function = function;

node->link = NULL;

ip->last->link = node;

ip->last = node;

}

int interpret(Interpreter \*ip, ParsedCommand \*pc) {

// Assert that there were operations added to the interpreter

assert (ip->head->link && "empty interpreter?");

FILE \*out = ip->output\_stream;

for (OperationNode \*node = ip->head->link; node; node = node->link) {

int match = 1;

match = match && !strcmp(node->operator, pc->operator);

match = match && (node->argument\_count == pc->argument\_count);

if (match) return node->function(out, pc);

}

// Print error if there were no commands found

fprintf(out, "error: no command %s that accepts %d argument%s.\n",

pc->operator, pc->argument\_count,

pc->argument\_count == 1 ? "" : "s");

return 0;

}

int interpret\_and\_free(Interpreter \*ip, ParsedCommand \*pc) {

int result = interpret(ip, pc);

free(pc);

return result;

}

#include "main.h"

#include "file.h"

#include "hashtable.h"

#include "help.h"

#include "history.h"

#include "memory.h"

#include "parser.h"

#include "interpreter.h"

#include "global.h"

#include "assemble.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <assert.h>

#define OPCODE\_FILE "opcode.txt"

#define RESERVED\_FILE "reserved.txt"

#define PROMPT "sicsim> "

static inline int hex\_to\_uint(char \*string, unsigned int \*value);

// Convert hex string to uint and save to value. Return success as boolean.

static inline int get\_line(char \*string);

// Scan one line to string and return whether it is the appropriate length.

static void load\_hash\_table(HashTable \*table, FILE \*in);

// Load HashTable for the given opcode input file and save to table.

static void load\_reserved\_dict(ReservedDict \*dict, FILE \*in);

static int quit\_0(FILE \*out, ParsedCommand \*pc);

static int history\_0(FILE \*out, ParsedCommand \*pc);

static int dir\_0(FILE \*out, ParsedCommand \*pc);

static int dump\_0(FILE \*out, ParsedCommand \*pc);

static int dump\_1(FILE \*out, ParsedCommand \*pc);

static int dump\_2(FILE \*out, ParsedCommand \*pc);

static int edit\_2(FILE \*out, ParsedCommand \*pc);

static int fill\_3(FILE \*out, ParsedCommand \*pc);

static int help\_0(FILE \*out, ParsedCommand \*pc);

static int reset\_0(FILE \*out, ParsedCommand \*pc);

static int opcodelist\_0(FILE \*out, ParsedCommand \*pc);

static int opcode\_1(FILE \*out, ParsedCommand \*pc);

static int type\_1(FILE \*out, ParsedCommand \*pc);

Global G;

int main(void) {

G.history = new\_history();

G.table = new\_hash\_table();

G.block = new\_memory\_block();

G.reserved = new\_reserved\_dict();

FILE \*opcode\_in;

if (!(opcode\_in = fopen(OPCODE\_FILE, "r")))

printf("error opening %s. continuing without opcodes\n", OPCODE\_FILE);

else {

load\_hash\_table(G.table, opcode\_in);

fclose(opcode\_in);

}

FILE \*reserved\_in;

if (!(reserved\_in = fopen(RESERVED\_FILE, "r")))

printf("error opening %s. continuing without reserved tokens\n",

RESERVED\_FILE);

else {

load\_reserved\_dict(G.reserved, reserved\_in);

fclose(reserved\_in);

}

Interpreter \*ip = new\_interpreter(stdout);

add\_operation(ip, "history", 0, history\_0);

add\_operation(ip, "hi", 0, history\_0);

add\_operation(ip, "q", 0, quit\_0);

add\_operation(ip, "quit", 0, quit\_0);

add\_operation(ip, "dir", 0, dir\_0);

add\_operation(ip, "d", 0, dir\_0);

add\_operation(ip, "edit", 2, edit\_2);

add\_operation(ip, "e", 2, edit\_2);

add\_operation(ip, "du", 0, dump\_0);

add\_operation(ip, "du", 1, dump\_1);

add\_operation(ip, "du", 2, dump\_2);

add\_operation(ip, "dump", 0, dump\_0);

add\_operation(ip, "dump", 1, dump\_1);

add\_operation(ip, "dump", 2, dump\_2);

add\_operation(ip, "help", 0, help\_0);

add\_operation(ip, "h", 0, help\_0);

add\_operation(ip, "f", 3, fill\_3);

add\_operation(ip, "fill", 3, fill\_3);

add\_operation(ip, "reset", 0, reset\_0);

add\_operation(ip, "opcodelist", 0, opcodelist\_0);

add\_operation(ip, "opcode", 1, opcode\_1);

add\_operation(ip, "type", 1, type\_1);

add\_operation(ip, "assemble", 1, assemble\_1);

add\_operation(ip, "symbol", 0, symbol\_0);

char command[COMMAND\_LENGTH];

ParsedCommand \*pc;

while (1) {

printf("%s", PROMPT);

if (!get\_line(command)) {

printf("error: command is too long\n");

continue;

}

int error\_code;

if (!(pc = parse\_command(command, &error\_code))) {

switch (error\_code) {

case TOO\_MANY\_ARGUMENTS\_ERROR:

printf("error: too many arguments\n"); break;

case INVALID\_FORMAT\_ERROR:

printf("error: format is invalid\n"); break;

default:

printf("error: could not parse command\n");

break;

}

continue;

}

if (interpret(ip, pc)) {

int is\_history = 0;

is\_history = is\_history || !strcmp(pc->operator, "history");

is\_history = is\_history || !strcmp(pc->operator, "hi");

if (!is\_history) add\_history(G.history, command);

}

free(pc);

}

return 0;

}

static inline int hex\_to\_uint(char \*string, unsigned int \*value) {

int length = strlen(string);

char format\_string[20];

int used;

sprintf(format\_string, "%%%dX%%n", length);

/\* Note that %n assigns the number of characters used for sscanf.

\* Also note that sscanf returns the number of fields whose values were

\* assigned - excluding %n

\*/

return sscanf(string, format\_string, value, &used) && used == length;

}

static inline int get\_line(char \*string) {

char \*p = string;

do {

if (p - string == COMMAND\_LENGTH - 1) return 0;

\*p = getchar();

} while (\*(p++) != '\n');

\*(p - 1) = 0; // places null-char at \n

return 1;

}

static void load\_hash\_table(HashTable \*table, FILE \*in) {

unsigned int opcode;

char mnemonic[100];

char operand\_count[100];

while (fscanf(in, "%02X %s %s", &opcode, mnemonic, operand\_count) == 3) {

assert(opcode < 256);

add\_to\_hash\_table(table, mnemonic, opcode);

}

}

static void load\_reserved\_dict(ReservedDict \*dict, FILE \*in) {

char string[100];

int type;

while (fscanf(in, "%s %d", string, &type) == 2) {

add\_to\_reserved\_dict(dict, string, (ReservedType) type);

}

}

static int history\_0(FILE \*out, ParsedCommand \*pc) {

int print = has\_history(G.history);

add\_history(G.history, pc->original\_command);

if (print) fprint\_history(stdout, G.history);

return 1;

}

static int quit\_0(FILE \*out, ParsedCommand \*pc) {

free(G.block);

free\_history(G.history);

free\_hash\_table(G.table);

exit(0);

return 1;

}

static int dir\_0(FILE \*out, ParsedCommand \*pc) {

fprint\_dir(out);

return 1;

}

static int dump\_0(FILE \*out, ParsedCommand \*pc) {

dump\_memory(out, G.block, -1, -1);

return 1;

}

static int dump\_1(FILE \*out, ParsedCommand \*pc) {

unsigned int start;

if (!hex\_to\_uint(pc->arguments[0], &start) || start >= BLOCK\_SIZE) {

printf("error: invalid start address\n");

return 0;

}

dump\_memory(out, G.block, (int) start, -1);

return 1;

}

static int dump\_2(FILE \*out, ParsedCommand \*pc) {

unsigned int start;

if (!hex\_to\_uint(pc->arguments[0], &start) || start >= BLOCK\_SIZE) {

printf("error: invalid start address\n");

return 0;

}

unsigned int end;

if (!hex\_to\_uint(pc->arguments[1], &end) || end >= BLOCK\_SIZE || end < start) {

printf("error: invalid end address\n");

return 0;

}

dump\_memory(out, G.block, (int) start, (int) end);

return 1;

}

static int edit\_2(FILE \*out, ParsedCommand \*pc) {

unsigned int address;

if (!hex\_to\_uint(pc->arguments[0], &address) || address >= BLOCK\_SIZE) {

fprintf(out, "error: invalid address\n");

return 0;

}

unsigned int value;

if (!hex\_to\_uint(pc->arguments[1], &value) || value >= 256) {

fprintf(out, "error: invalid value\n");

return 0;

}

set\_memory(G.block, address, value);

return 1;

}

static int fill\_3(FILE \*out, ParsedCommand \*pc) {

unsigned int start;

if (!hex\_to\_uint(pc->arguments[0], &start) || start >= BLOCK\_SIZE) {

fprintf(out, "error: invalid start address\n");

return 0;

}

unsigned int end;

if (!hex\_to\_uint(pc->arguments[1], &end) || end >= BLOCK\_SIZE || end < start) {

fprintf(out, "error: invalid end address\n");

return 0;

}

unsigned int value;

if (!hex\_to\_uint(pc->arguments[2], &value) || value >= 256) {

fprintf(out, "error: invalid value\n");

return 0;

}

fill\_memory(G.block, start, end ,value);

return 1;

}

static int help\_0(FILE \*out, ParsedCommand \*pc) {

fprint\_help(out);

return 1;

}

static int reset\_0(FILE \*out, ParsedCommand \*pc) {

reset\_memory(G.block);

return 1;

}

static int opcode\_1(FILE \*out, ParsedCommand \*pc) {

unsigned char opcode;

if (!find\_from\_hash\_table(G.table, pc->arguments[0], &opcode)) {

fprintf(out, "Couldn't find opcode for %s\n", pc->arguments[0]);

return 0;

}

printf("opcode is %02X\n", opcode);

return 1;

}

static int opcodelist\_0(FILE \*out, ParsedCommand \*pc) {

fprint\_hash\_table(out, G.table);

return 1;

}

static int type\_1(FILE \*out, ParsedCommand \*pc) {

fprint\_file(out, pc->arguments[0]);

return 1;

}

#include "memory.h"

#include <assert.h>

#include <stdio.h>

#include <stdlib.h>

/\* Recall this definition

struct Block {

unsigned char data[BLOCK\_SIZE];

int current;

}

\*/

#ifdef TEST

// Manual unit test for this module

// Refer to the README for testing instructions

int main(void) {

Block \*block = new\_memory\_block();

for (int i = 0; i < BLOCK\_SIZE; ++i)

set\_memory(block, i, i % 256);

printf("Dump Test\n");

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

dump\_memory(stdout, block, -1, -1);

printf("\nDump Range & Offset Test\n");

dump\_memory(stdout, block, 23, 48);

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

dump\_memory(stdout, block, -1, -1);

printf("\nOverflow Test\n");

dump\_memory(stdout, block, BLOCK\_SIZE - 4, -1);

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

dump\_memory(stdout, block, -1, -1);

printf("\nRange & Overflow Test\n");

dump\_memory(stdout, block, 0xFFFD7, 0xFFFE0);

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

dump\_memory(stdout, block, -1, -1);

printf("\nReset Test\n");

reset\_memory(block);

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

dump\_memory(stdout, block, -1, -1);

free(block);

return 0;

}

#endif

// Make sure to free after use!

Block \*new\_memory\_block() {

Block \*block = malloc(sizeof(Block));

for (int i = 0; i < BLOCK\_SIZE; ++i)

block->data[i] = '\0';

block->current = 0;

return block;

}

void set\_memory(Block \*block, int location, unsigned char value) {

block->data[location] = value;

}

void fill\_memory(Block \*block, int start, int end, unsigned char value) {

// Inclusive, Namgyu's exception rules

if (start < 0) start = 0;

if (BLOCK\_SIZE <= end) end = BLOCK\_SIZE - 1;

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

block->data[i] = value;

}

static inline void fprint\_digit(FILE \*out, int i) {

int digit\_34 = i / 16 % 256;

int digit\_12 = i / 4096; assert(digit\_12 < 256);

fprintf(out, "%02X%02X0 ", digit\_12, digit\_34);

}

static inline void fprint\_data\_hex(FILE \*out, Block \*block,

int start, int end, int base\_index) {

for (int i = 0; i < 16; ++i) {

int index = base\_index + i;

int value = block->data[index];

if (start <= index && index <= end)

fprintf(out, "%02X ", value);

else

fprintf(out, " ");

}

}

static inline void fprint\_data\_char(FILE \*out, Block \*block,

int start, int end, int base\_index) {

const int X20 = 32;

const int X7E = 126;

for (int i = 0; i < 16; ++i) {

int index = base\_index + i;

int value = block->data[index];

if (index < start || end < index || value < X20 || X7E < value)

fprintf(out, ".");

else

fprintf(out, "%c", value);

}

}

// Make sure to free after use!

void dump\_memory(FILE \*out, Block \*block, int start, int end) {

if (start == -1) start = block->current; else assert(start >= 0);

if (end == -1) end = start + 159;

if (end >= BLOCK\_SIZE) end = BLOCK\_SIZE - 1;

block->current = end + 1;

if (block->current == BLOCK\_SIZE) block->current = 0;

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

fprint\_digit(out, i);

fprint\_data\_hex(out, block, start, end, i);

fprintf(out, "; ");

fprint\_data\_char(out, block, start, end, i);

fprintf(out, "\n");

}

}

void reset\_memory(Block \*block) {

for (int i = 0; i < BLOCK\_SIZE; ++i) block->data[i] = 0;

block->current = 0;

}

#include "parser.h"

#include <stdio.h>

#include <stdlib.h>

#include <assert.h>

#include <string.h>

/\* Note these definitions

#define INVALID\_FORMAT\_ERROR 1

#define TOO\_MANY\_ARGUMENTS\_ERROR 2

#define COMMAND\_TOO\_LONG\_ERROR 3

#define ARGUMENT\_TOO\_LONG\_ERROR 4

#define COMMAND\_LENGTH 120

#define TOKEN\_COUNT 4

#define ARGUMENT\_COUNT 3

#define TOKEN\_LENGTH 40

typedef struct \_ParsedCommand {

char original\_command[COMMAND\_LENGTH];

char tokenized\_command[COMMAND\_LENGTH];

char \*operator;

char \*arguments[ARGUMENT\_COUNT];

int argument\_count;

} ParsedCommand;

\*/

static inline int is\_comma(char c);

// Return whether c is a comma. Functionalized for readability.

static inline int is\_normal\_letter(char c);

// Return whether c is a normal letter (not comma, space, newline, or tab).

static int validate\_and\_tokenize(char \*string, int \*token\_count, char \*\*tokens);

// Return whether the comma sequence is valid and remove commas.

static int validate\_and\_remove\_commas(char \*string);

// Return whether the spacing and number of command tokens are valid and

// save the (number of) tokens to token(s)(\_count).

#ifdef TEST

int main(void) {

ParsedCommand \*pc;

int error\_code;

pc = NULL;

pc = parse\_command("ho hoho, koko, mo", &error\_code);

assert(pc && "command should be valid");

pc = NULL;

pc = parse\_command("comma is not here!", &error\_code);

assert(!pc && "command shouldn't be valid");

assert(error\_code == INVALID\_FORMAT\_ERROR);

pc = NULL;

pc = parse\_command("comma is, not there!", &error\_code);

assert(!pc && "command shouldn't be valid");

assert(error\_code == INVALID\_FORMAT\_ERROR);

pc = NULL;

pc = parse\_command("too many, arguments, you, know", &error\_code);

assert(!pc && "command shouldn't be valid");

assert(error\_code == TOO\_MANY\_ARGUMENTS\_ERROR);

pc = NULL;

pc = parse\_command("yolo \t ho, \t ko, \t yo ", &error\_code);

assert(pc && "command should be valid");

assert(!strcmp(pc->operator, "yolo"));

assert(!strcmp(pc->arguments[0], "ho"));

assert(!strcmp(pc->arguments[1], "ko"));

assert(!strcmp(pc->arguments[2], "yo"));

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

printf("Automatic tests successful!\n");

char \*test\_string = "yolo \t ho, \t ho, \t yo";

printf("\nParsing %s\n", test\_string);

pc = parse\_command("yolo \t ho, \t ko, \t yo ", &error\_code);

printf("operator: %s\n", pc->operator);

for (int i = 0; i < pc->argument\_count; ++i)

printf("argument%d: %s\n", i, pc->arguments[i]);

return 0;

}

#endif

ParsedCommand \*parse\_command(char \*command, int \*error\_code) {

ParsedCommand \*pc = malloc(sizeof(ParsedCommand));

strcpy(pc->original\_command, command);

strcpy(pc->tokenized\_command, command);

if (!validate\_and\_remove\_commas(pc->tokenized\_command)) {

free(pc); \*error\_code = INVALID\_FORMAT\_ERROR; return NULL;

}

// Get all tokens and assign to operator + arguments within pc

int token\_count;

char \*tokens[TOKEN\_COUNT];

if (!validate\_and\_tokenize(pc->tokenized\_command, &token\_count, tokens)) {

free(pc); \*error\_code = TOO\_MANY\_ARGUMENTS\_ERROR; return NULL;

}

pc->argument\_count = token\_count - 1;

pc->operator = tokens[0];

for (int i = 1; i < token\_count; ++i)

pc->arguments[i - 1] = tokens[i];

return pc;

}

static inline int is\_comma(char c) {

return c == ',';

}

static inline int is\_normal\_letter(char c) {

if (c == ' ') return 0;

if (c == '\t') return 0;

if (c == ',') return 0;

if (c == '\n') return 0;

return 1;

}

static int validate\_and\_remove\_commas(char \*string) {

int word\_index = 0;

int was\_letter = 0;

int comma\_avail = 0;

for (; \*string; ++string) {

if (is\_normal\_letter(\*string)) {

if (!was\_letter) {

word\_index++;

if (comma\_avail > 0) return 0;

if (word\_index >= 2) comma\_avail = 1;

}

was\_letter = 1;

} else {

was\_letter = 0;

if (is\_comma(\*string)) {

\*string = ' ';

comma\_avail--;

}

if (comma\_avail < 0) return 0;

}

}

return 1;

}

static int validate\_and\_tokenize(char \*string, int \*token\_count, char \*\*tokens) {

\*token\_count = 0;

int waiting\_for\_word = 1;

for (; \*string; ++string) {

if (is\_normal\_letter(\*string)) {

if (waiting\_for\_word) {

if (\*token\_count == TOKEN\_COUNT) return 0;

tokens[(\*token\_count)++] = string;

waiting\_for\_word = 0;

}

} else {

\*string = '\0';

waiting\_for\_word = 1;

}

}

return 1;

}

#include "register.h"

#include <stdlib.h>

const char \*register\_to\_string(Register reg) {

static char \*strings[8] = {

"A", "X", "L", "B", "S", "T", "PC", "SW" };

static Register registers[8] = {

A, X, L, B, S, T, PC, SW };

for (int i = 0; i < 8; ++i) {

if (registers[i] == reg) {

return strings[i];

}

}

return NULL;

}

#include "reserved.h"

#include "generic\_dict.h"

#include <stdlib.h>

/\* Recall these definitions

typedef enum \_ReservedType {

MEM\_OPERATION = 8,

REG\_OPERATION = 9,

REG\_PAIR\_OPERATION = 10,

REG\_N\_OPERATION = 11,

N\_OPERATION = 12,

BYTE = 0,

WORD = 1,

RBYTE = 2,

RWORD = 3,

START = 4,

END = 5,

BASE = 6,

NOBASE = 7

} ReservedType;

typedef struct \_ReservedValue {

ReservedType type;

} ReservedValue;

typedef Dict ReservedDict;

\*/

static const int RESERVED\_DICT\_SIZE = 40;

ReservedDict \*new\_reserved\_dict() {

ReservedDict \*dict = new\_dict(RESERVED\_DICT\_SIZE);

return dict;

}

void free\_reserved\_dict(ReservedDict \*dict) {

free\_dict(dict, free);

}

void add\_to\_reserved\_dict(ReservedDict \*dict, char \*key, ReservedType type) {

ReservedValue \*value = malloc(sizeof(ReservedValue));

value->type = type;

add\_to\_dict(dict, key, value);

}

int find\_reserved\_type(ReservedDict \*dict, char \*key, ReservedType \*type) {

ReservedValue \*value;

if (find\_from\_dict(dict, key, (void\*\*) &value)) {

\*type = value->type;

return 1;

} else {

return 0;

}

}

#include "symbol.h"

#include "generic\_dict.h"

#include <stdio.h>

#include <stdlib.h>

#include <assert.h>

#include <string.h>

/\* Recall these definitions

#define BASE\_SIZE 10

#define SYMBOL\_LENGTH 6

typedef struct \_Symbol {

unsigned int address;

} Symbol;

typedef Dict SymbolDict;

\*/

static const int SYMBOL\_DICT\_SIZE = 40;

#ifdef TEST

#undef TEST

#include "generic\_dict.c"

#define TEST

int main(void) {

{

SymbolDict \*dict = new\_symbol\_dict();

add\_to\_symbol\_dict(dict, "Hello", 1023);

add\_to\_symbol\_dict(dict, "Yo", 2039);

unsigned int address;

assert(find\_symbol\_address(dict, "Hello", &address));

assert(address == 1023);

assert(find\_symbol\_address(dict, "Yo", &address));

assert(address == 2039);

free\_symbol\_dict(dict);

}

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

printf("Automatic tests successful!\n");

#ifdef EXPLICIT

{

SymbolDict \*dict = new\_symbol\_dict();

add\_to\_symbol\_dict(dict, "Hello", 512);

add\_to\_symbol\_dict(dict, "Yo", 1024);

add\_to\_symbol\_dict(dict, "Yolo", 2048);

add\_to\_symbol\_dict(dict, "Abs", 1536);

add\_to\_symbol\_dict(dict, "Hoho", 256);

fprint\_symbols(stdout, dict);

free\_symbol\_dict(dict);

}

#endif

return 0;

}

#endif

SymbolDict \*new\_symbol\_dict() {

return new\_dict(SYMBOL\_DICT\_SIZE);

}

void add\_to\_symbol\_dict(SymbolDict \*dict, char \*string, unsigned int address) {

Symbol \*symbol = malloc(sizeof(Symbol));

symbol->address = address;

add\_to\_dict(dict, string, symbol);

}

int find\_symbol\_address(SymbolDict \*dict, char \*string, unsigned int \*address) {

Symbol \*symbol;

if (find\_from\_dict(dict, string, (void\*\*) &symbol)) {

\*address = symbol->address;

return 1;

} else

return 0;

}

void free\_symbol\_dict(SymbolDict \*dict) {

free\_dict(dict, free);

}

// Note that typedef and structs are static by default

typedef struct \_SymbolPair {

char \*string;

unsigned int address;

} SymbolPair;

static int compare\_symbol\_pairs(const void \*a, const void \*b) {

return - strcmp(((SymbolPair\*)a)->string, ((SymbolPair\*)b)->string);

}

// Note that this function is reliant on the implementation of Dict

void fprint\_symbols(FILE \*out, SymbolDict \*dict) {

int count = 0;

for (int i = 0; i < dict->size; ++i) {

DictNode list = dict->nodes[i];

for (BucketNode \*node = list.head->link; node; node = node->link) {

count ++;

}

}

SymbolPair \*pairs = malloc(sizeof(SymbolPair) \* count);

int index = 0;

for (int i = 0; i < dict->size; ++i) {

DictNode list = dict->nodes[i];

for (BucketNode \*node = list.head->link; node; node = node->link) {

pairs[index].string = node->key;

pairs[index++].address = ((Symbol\*)node->value)->address;

}

}

qsort(pairs, count, sizeof(SymbolPair), compare\_symbol\_pairs);

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

SymbolPair pair = pairs[i];

fprintf(out, "\t%s\t%04X\n", pair.string, pair.address);

}

}