**과목명: 시스템프로그래밍**

**2 분반**

**<<Project #1>>**

**서강대학교 컴퓨터공학부**

**20151550**

**박태준**

목 차

1. **프로그램 개요**
2. **프로그램 설명**
   1. 프로그램 흐름도
3. **모듈 정의**
   1. 정의한 모듈에 대해 간략적인 설명
4. **전역 변수 정의**
5. **코드 설명**

**1.프로그램 개요**

이 프로그램은 SIC/XE 머신을 구현하기 위한 전 단계 작업이라고 할 수 있다. 모든 명령을 수행할 셸(shell) 프롬프트와 컴파일을 통해서 만들어진 object 코드가 적재되고 실행될 메모리공간, 그리고 string 형식의 mnemonic을 opcode 값으로 변환하는 OPCODE 테이블과 관련 명령어들을 출력하는 프로그램이다. 프로그램이 동작시키는 셸은 총 9개의 명령을 수행할 수 있다.

**2.프로그램 설명**

2.1 프로그램 흐름도

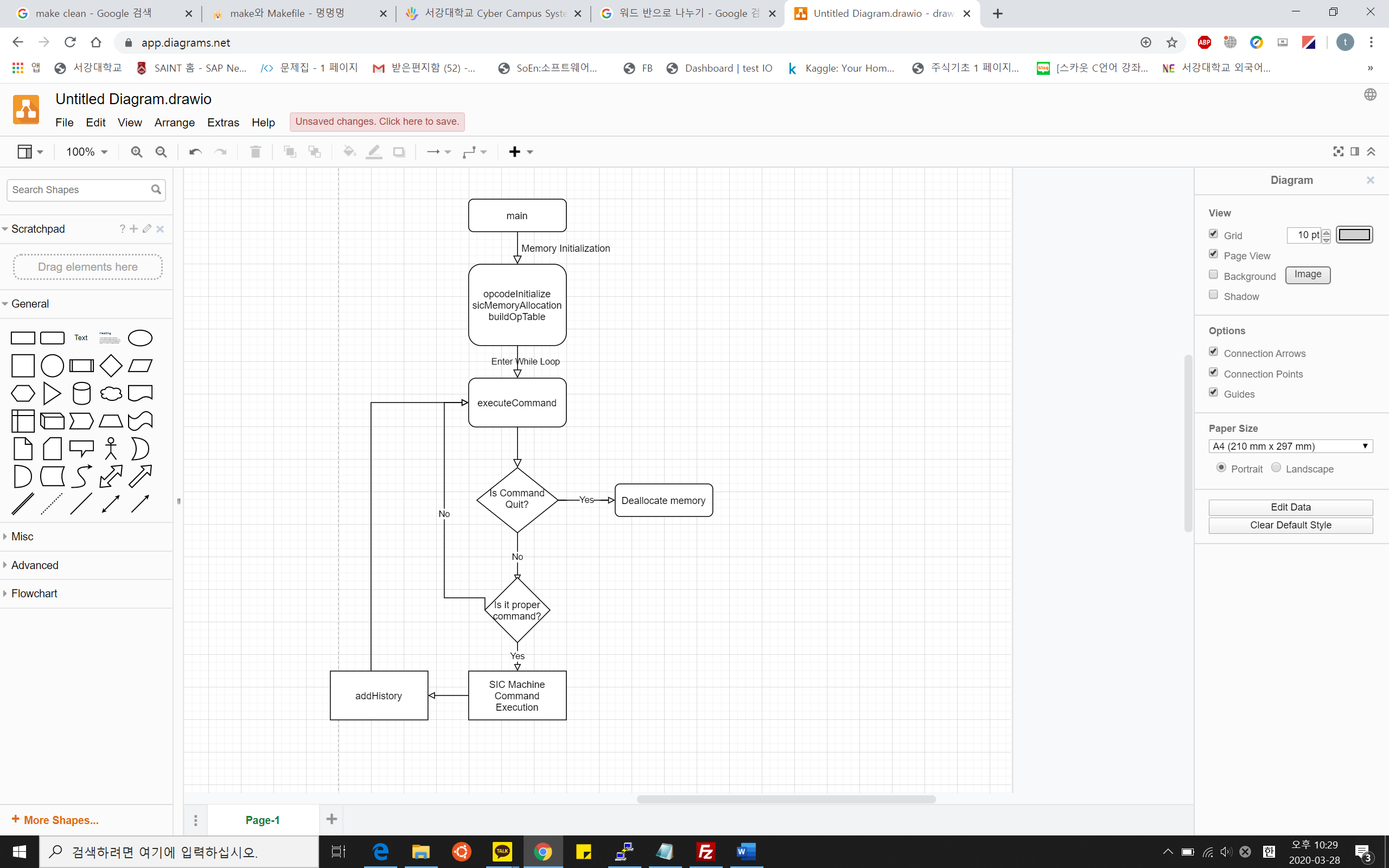


그림 1. Main 함수부터 Command Line 흐름도

위 그림은 main 함수로 시작해서 execution 이전의 과정인 셸을 설명하는 흐름도이다. 셸의 명령어 실행은 SIC Machine Command Execution으로 함축하고, 후에 설명한다.

1. 메인 함수가 시작되면 전역 변수들에 대한 메모리 초기화가 진행된다.
2. 이후 while loop를 진행하며 shell prompt의 역할을 수행한다.
3. executeCommand 함수에는 유저가 입력한 명령문이 입력되고, 이를 validation 하는 과정이 진행된다.
4. Command가 quit일 경우 무한 루프를 종료하고, 할당했던 메모리를 해제 후 프로그램을 종료한다.
5. 3번의 validation하는 과정에서 command가 잘못된 것을 확인할 경우 다시 새로운 command를 입력받는다.
6. Proper command의 경우, Command Execution을 진행하고, 해당 command를 히스토리에 저장한다.
7. 다시 executeCommand를 수행한다.

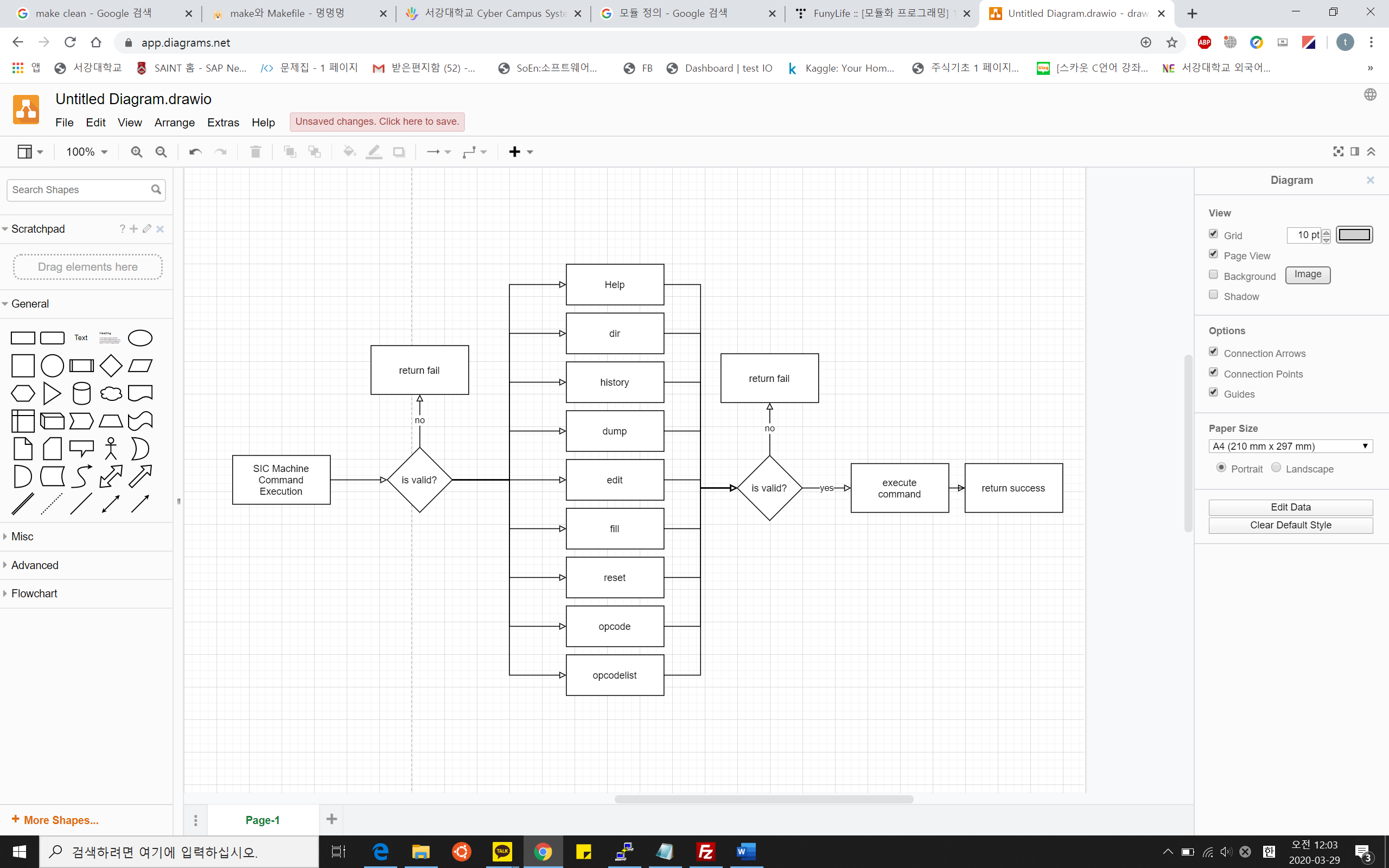


그림 2. SIC Machine Command Execution 실행 과정

위 그림은 그림 1.에서 생략한 SIC Machine Command Execution의 과정을 설명한다.

SIC Machine Command Execution은 명령어의 유효성을 판단하고 실행하는 과정을 개략적으로 설명한다.

1. Command를 통해 들어온 명령이 유효한지 (셸에 속한 명령어인지) 확인한다.
2. 유효하지 않을 경우 fail을 리턴한다.
3. 유효한 명령어의 경우, 각 명령어에 따라서 명령을 수행한다.
4. 이때 명세서의 조건에 부합하지 않는다면 fail을 리턴한다.
5. 명세서의 조건에도 부합한다면, 명령을 수행하는데 오류가 없다고 판단, 이를 수행한다.
6. Success를 리턴한다.

3.모듈 정의

3.1 정의한 모듈.

이 프로그램은 명세서에 정의된 명령어들을 수행하는 Execute Command, 루프를 돌며 string을 분해하고 Execute Command에 명령어 및 요소를 전달하는 getOrder로 크게 두 가지로 나눌 수 있다. 2. 프로그램 흐름도의 첫 번째 흐름도가 getOrder, 두 번째 흐름도가 Execute Command라고 할 수 있다.

또한 Command들의 경우, 각 Command마다 Argument들의 유효성을 검사하는 Valid…Arg 함수들과 (ex. validFillArg) 유효성 검사 후 직접 명령을 수행하는 execute…(ex. excuteReset) 함수 부분으로 나눌 수 있다.

다른 한편으로, SIC Machine의 메모리에 접근하는 명령어인 dump, edit, fill, reset 명령어와, command 기록에 접근하는 history, opcode와 관련된 opcode, opcodelist, 그외에 따로 동작하는 dir, help, quit까지 4가지 종류로 분리할 수 있다.

3.2 사용자 정의 상수

Help, dir, quit, history, dump, edit, fill, reset, opcode, opcodelist

명령어를 대신하는 상수를 선언하였다.

Success, fail

함수 실행의 성공, 실패 유무를 이용해 예외처리를 하기 위해 선언하였다.

commandLen

사용자가 입력하는 커맨드의 최대 길이를 제한하기 위해 선언하였다.

hasComma, noComma

사용자가 쉼표를 입력해야 하는 양식이 존재하고, 하지 않아야 하는 양식이 존재한다.

각각의 예외처리에 대해서 대응하기 위해 위와 같은 상수를 미리 선언하였다.

memSize

1MB의 SIC machine 메모리를 효율적으로 저장하기 위해 선언하였다. 1MB = 16 \* memSize이다.

3.3 사용자 정의 자료 구조 설명

Typedef struct hist{

Char command[50]

Struct hist \*next;

}historyLink

Linked List를 활용해서 사용자의 Command 기록을 저장하는 자료 구조. 유효한 Command만 저장하는데 사용된다.

Typedef struct opc{

Char op[10]

Int format[2]

Int opCode

Struct opc\* next

}

Opcode에 대한 Hashtable의 자료구조. Opcode.txt에 명시된 opcode의 정보는, opcode의 이름, 대응되는 숫자, format의 종류(최대 2개)를 제공한다. 이를 다 저장하기 위해 위와 같은 자료구조를 만들었다.

3.4 함수 설명

Int checkComma(char\* str)

Str 이라는 스트링에서, 마지막 원소가 ‘,’인지 판단한다.

맞을 경우 success를, 아닌 경우 fail을 리턴한다.

Int stringToHex(char\* str, int needComma)

Str을 16진수의 숫자로 변환하는 함수이다. 다만 변환하기 전에 변환가능한지 검사한다.

이때 checkComma를 사용하는데, needComma를 이용해 해당 스트링이 comma가 필요한지 아닌지를 검사한다. 이는 dump start, end와 같이 일부 format에 쉼표가 들어가기 때문이다.

Int checkFirstToken(char \*firstTok)

firstTok이란 명령어로 들어오는 스트링 중 띄어쓰기로 잘랐을 때 가장 첫 번째 요소를 의미한다. 즉, 명령어를 검사하는 함수이다. 입력된 스트링이 명세서의 명령어에 해당할 경우 해당 명령어의 상수 값을 리턴한다. 명령을 받지 않았거나 없는 명령어를 입력할 경우 fail을 리턴한다.

Int checkSecondToken(char \*secTok, int firstTok)

위의 언급한 바와 같이, 스트링을 잘랐을 때 두 번째 요소를 의미한다. 두 번째 요소는 어느 정도 특별함이 존재하는데, 일부 명령어는 두 번째 요소가 있으면 안되는 경우가 있기 때문이다. 이에 따라서 명령어의 종류별로 나눠 해당 명령어의 두 번째 요소 조건 여부를 판단한다. 맞지 않을 경우 fail을, 맞을 경우 success를 리턴한다.

Int executeCommand(int command, char\* Arg2, char\* Arg3, char\* Arg4, char\* Arg5, historyLink\* Hist, opcode\* Op[])

명령이 존재하는 경우에만 실행되는 함수다. 명령의 상수 번호와 토큰으로 잘린 요소들, 작업을 위해 History의 링크드리스트와 Opcode의 해시 테이블의 포인터를 파라미터로 가져온다. 각 명령에 대해서 실행시키는 함수인 “execute …” 함수들을 부르는 함수이다. 일부 함수들의 경우 함수를 직접 실행시키기 전에 요소들의 유효성을 검사하는 “valid …” 함수들을 부르기도 한다. 이를 통과하지 못할 경우 fail을 리턴하고, 모두 성공할 경우 success가 리턴된다. executeCommand에서 success가 리턴되는 경우, 성공한 명령어를 기록하는 addHistory 함수가 실행된다.

Void addHistory(historyLink\* head, char\* record)

위의 executeCommand가 success를 리턴할 때만 실행된다. 성공한 명령들을 저장하는 historyLink들을 파라미터로 가져와 record의 스트링을 저장한다. 이때 새로운 메모리를 할당하여 Linked List의 형태로 데이터의 가장 끝 chain에 새 데이터를 저장한다.

Int executeHelp()

Help 명령어를 수행하는 함수이다.

Int executeDir()

Dir 명령어를 수행하는 함수이다. Dirent, sts/stat 라이브러리를 활용해 디렉토리일 경우 ‘/’를 붙이고, 실행가능한 함수의 경우 ‘\*’을 뒤에 붙여서 출력하는 함수이다. 그 외에 현재 폴더 내의 파일을 모두 출력한다.

Int executeHistory(historyLink\* head)

이전 addHistory와 달리 History의 명령을 수행하는 함수이다. History의 기록을 확인하는 함수이다. 이때 Linked List를 Head부터 끝까지 탐색하며 출력한다.

Int validDumpArg(char\* Arg2, char\* Arg3, char\* Arg4)

dump명령을 수행하는 executeDump를 수행하기 전에 요소들의 유효성을 검사하기 위한 함수이다. 해당 함수가 fail을 리턴할 경우 executeCommand가 그대로 fail을 리턴하기에 유효하지 않은 요소가 들어올 경우 history에 추가되지 않는다. 요소가 모두 유효한 경우 dump에 들어갈 요소의 수를 리턴한다. 리턴 값의 범위는 0,1,2 이다.

Int executeDump(char\* Arg2, char\* Arg3, int argNum)

dump명령을 수행하는 함수이다. Dump 명령은 출력이 필요하기 때문에 dumpNoArg, dumpWithArgs라는 출력 함수를 사용해 출력한다. 이 함수는 dump에 들어온 요소의 개수에 따라서 출력 함수를 골라주고, 요소들을 모두 16진수로 변환시켜 넣는 함수이다. 또한 16진수로 바꿨을 때, 명세서의 요구사항에 반하는 숫자가 있는지 검사한다.

Int dumpNoArg(),dumpWithArgs(int h1, int h2)

Dump의 명령어의 요소의 개수에 따라 실행되는 출력함수

Int validEditArg(char\* Arg2, char\* Arg3, char\* Arg4)

Edit 명령어에 들어오는 요소들의 유효성을 검사하는 함수이다.

Int executeEdit(char\* Arg2, char\* Arg3)

validEditArg를 통해 유효성 검사가 완료되면 실행되는 함수이다. 실제 SIC 머신의 메모리에 접근해 값을 변환시킨다. 그러나 명세서의 조건에 어긋나는 경우 변환시키지 않고 fail을 리턴한다.

Int validFillArg(char\* Arg2, char\* Arg3, char\* Arg4, char\* Arg5)

Fill 명령어는 3개의 요소가 필요하기에 이를 검사한다.

Int executeFill(char\* Arg2, char\* Arg3, char\* Arg4)

validFillArg를 통과한 뒤 실행된다. SIC 머신의 메모리에 접근해 start부터 end까지의 값을 변환시킨다. 그러나 명세서의 조건에 어긋나는 값을 가지고 있다면 변환시키지 않고 fail을 리턴한다.

Int executeReset()

1MB메모리를 모두 0으로 리셋하는 함수이다.

Int validaOpcodeArg(char\* Arg2, char\* Arg3)

Opcode 명령어에 들어오는 요소의 유효성을 검사한다. 그러나 hash table을 이용하면 스트링의 불일치는 자연스럽게 확인할 수 있다. 따라서 Arg2가 NULL이거나, Arg3가 존재하는 경우만 fail을 리턴한다.

Int executeOpcode(char\* Arg2, opcode\* Op[])

Hash Function을 이용해서 인덱스를 정하고, 해당 인덱스 번 째의 해시 테이블에 접근해 NULL이 나올 때까지 opcode를 검사한다. 만약 해당 opcode가 존재할 경우 값을 리턴 해주고 없을 경우 없다는 메시지를 출력한다.

Int executeOpcodelist(opcode\* op[])

Opcodelist 명령어를 수행하는 함수이다.

0번째부터 19번째 해시 테이블을 모두 순회하며 속해있는 모든 opcode를 출력해준다.

Int getOrder(historyLink\* histHead, opcode\* opHead[])

커맨드라인의 가장 핵심 함수이다. 사용자로부터 명령어의 string을 받고, 이를 토큰으로 나눠준다. 토큰으로 나눈 뒤, checkFirstToken을 이용해 명령어를 판단한다. 이후 executeCommand를 실행시켜 명령을 실행시킨다. 또한 commandCode를 리턴하면서 commandCode가 quit일 경우 프로그램을 종료하도록 시그널을 보내는 역할을 한다.

Void freeHistory, sicMemoryAllocation, sicMemoryFree, opcodeInitialize, opcodeFree, buildOpTable

전역 변수에 해당하는 History 기록들, 1MB의 메모리, Opcode의 데이터를 저장하는 변수들을 초기화하거나 메모리를 해제해주는 함수이다.

Int hashFunc(char \*command)

Hash table을 만들기 위해서 사용하는 해시 함수이다.

1. command의 모든 character의 ascii 값을 더한다.

2. 이를 20으로 나눈 나머지 값을 리턴한다.

Int main()

메인 함수의 경우 초기화를 하는 함수를 부르는 것, 종료시 메모리를 해제하는 함수를 부르는 것, while 문 내에서 getOrder를 실행하며 getOrder가 quit 값을 보낼 때까지 실행하는 역할을 한다.

4.전역 변수 정의

sicMemory : sic 머신 구현을 위해 1 Mbyte의 메모리를 할당한 Character형 배열. 편의를 위해 [65536][16] 크기의 2차원 배열의 형태를 가지고 있다.

dumpAddress : dump가 실행될 때 start, end argument가 없는 경우 10줄만 출력하며, 그 다음 dump에서 실행 시 이전 출력 메모리 이후부터 새로 출력을 해야한다. 이때 얼마나 출력했는지를 저장하기 위해 사용하는 int형 변수이다.

opHead : opcode를 저장하기 위해 만든 20개 크기의 Hash table 자료구조이다. opCode라는 user defined data structure를 사용한다.

historyHead : 사용자의 사용 기록을 조회하는 History 기능을 구현하기 위해 만든 전역 변수이다. historyLink라는 user defined data structure를 사용한다.

5.코드 설명

5.1 20151550.h

#include<stdio.h>

#include<stdlib.h>

#include<dirent.h>

#include<string.h>

#include<sys/stat.h>

typedef struct hist{

    char command[500];

    struct hist \*next;

}historyLink;

//Structure to save history.

typedef struct opc{

    char op[10];

    int format[2];

    int opCode;

    struct opc \*next;

}opCode;

const int commandLen = 50;

//Maximum Length of command.

const int help = 10;

const int dir = 11;

const int quit = 12;

const int history = 13;

const int dump = 14;

const int edit = 15;

const int fill = 16;

const int reset = 17;

const int opcode = 18;

const int opcodelist = 19;

//Above constants are needed to distinguish functions, thereby making codes more readable.

const int fail = -1;

const int success = 1;

//When functions succeed to make right result, returns success.

//In contrast, return fail when fail to make right result because of wrong command.

const int hasComma = -2;

const int noComma = -3;

// hasComma means a string includes comma at last.

// noComma means a string does not include comma.

const int memSize = 65536;

// memSize is total number of 16 bytes array.

// By defining memSize=65536, I can allocate 1MB memory.

extern char \*\*sicMemory;

extern int dumpAddress;

extern opCode\* opHead[20];

extern historyLink\* historyHead;

// Defining History link, opcode, memory address

int executeHelp();

int executeDir();

int executeHistory();

int executeDump();

int executeEdit();

int executeFill();

int executeReset();

int executeOpcode();

int executeOpcodelist();

int validDumpArg();

int validEditArg();

int validOpcodeArg();

int validFillArg();

int hashFunc();

int checkComma();

int stringToHex();

int checkFirstToken();

int checkSecondToken();

int executeCommand();

void addHistory();

void dumpNoArg();

void dumpWithArgs();

int getOrder();

void freeHistory();

void sicMemoryAllocation();

void sicMemoryFree();

void opcodeInitialize();

void opcodeFree();

void buildOpTable();

//pre-defining functions. Will explain on c file.

Header 파일은 정의부로 나뉘기에 앞서 언급한 전역 변수 및 함수, 구조체로 되어 있어 설명은 생략한다.

5.2 20151550.c

#include "20151550.h"

char \*\*sicMemory;

// Memory for sic machine.

int dumpAddress = 0;

// The variable is used to save command 'dump'.

// Dump without additional arguments need record of showed memory last time.

int checkComma(char\* str)

{

    if(str[strlen(str)-1] == ',')return success;

    else return fail;

}

스트링의 마지막에 ,가 있으면 1, 없으면 -1을 리턴한다.

,가 있는지 검사하기 위한 함수

//opCode\* opHead[20];

//historyLink\* historyLink;

/\*

checkComma is a function to find comma at last element of string.

This function is only called in stringToHex function when needed.

if the string don't have comman, return fail.

\*/

int stringToHex(char\* str, int needComma)

{

    int count;

    int stringLength = strlen(str);

    if(needComma == hasComma)

    {

    stringLength = strlen(str)-1;

    if(checkComma(str) == fail)return fail;

    }

    for(count = 0; count < stringLength; count++)

    {

    if(str[count] >= '0' && str[count] <= '9')continue;

    if(str[count] >= 'a' && str[count] <= 'f')continue;

    if(str[count] >= 'A' && str[count] <= 'F')continue;

    return fail;

    }

    int value = strtol(str,NULL,16);

    return value;

}

스트링으로 받은 명령어의 추가적인 argument들을 16진수 값으로 바꿔주는 함수

needComma는 ,가 필요한 경우 -2 필요하지 않은 경우 -3을 보내서 구분한다.

for문을 이용해 조건을 만족할 경우 16진수로 변환하는 함수를 사용하고 그 값을 리턴해준다.

/\*

   stringToHex is a function that convert string to hexadecimal number.

   However, if string does not suffice hexadecimal, return fail.

   But in case of comma, the function get the need of comma and if it is needed, check it through checkComma function.

   In normal case, it returns hexadecimal value.

\*/

int checkFirstToken(char \*firstTok)

{

    //if(firstTok == NULL)return fail;

    if (strcmp("h",firstTok)== 0)return help;

    if (strcmp("help",firstTok)==0)return help;

    if (strcmp("d",firstTok)== 0)return dir;

    if (strcmp("dir",firstTok)==0)return dir;

    if (strcmp("q",firstTok)== 0)return quit;

    if (strcmp("quit",firstTok)==0)return quit;

    if (strcmp("hi",firstTok)== 0)return history;

    if (strcmp("history",firstTok)==0)return history;

    if (strcmp("du",firstTok)== 0)return dump;

    if (strcmp("dump",firstTok)==0)return dump;

    if (strcmp("e",firstTok)== 0)return edit;

    if (strcmp("edit",firstTok)==0)return edit;

    if (strcmp("f",firstTok)== 0)return fill;

    if (strcmp("fill",firstTok)==0)return fill;

    if (strcmp("reset",firstTok)== 0)return reset;

    if (strcmp("opcode",firstTok)==0)return opcode;

    if (strcmp("opcodelist",firstTok)== 0)return opcodelist;

    return fail;

}

들어온 명령어를 확인하는 함수이다. 명세서의 명령어가 아닐 경우 -1을 리턴해 잘못된 명령어임을 밝힌다.

/\*

    checkFirstToken check validation of first token.

    First token in this function means a first argument of command string.

    Ex.) In "dump 4, 37" command, the first token is dump.

    This function check whether first token is right command.

    If it is, return a code number of command.

    If not, return fail.

\*/

int checkSecondToken(char \*secTok,int firstTok)

{

    if (firstTok == dump)return firstTok;

    else if(firstTok == help || firstTok == dir || firstTok == quit || firstTok == history || firstTok ==reset||firstTok == opcodelist)

    {

    if( secTok == NULL)return firstTok;

    else if(strlen(secTok) > 0)

    {

        printf("This Command Does Not Contain Arguments\n");

        return fail;

    }

    else return firstTok;

    }

    else

    {

    if(secTok == NULL || strlen(secTok) == 0)

    {

        printf("This Command Needs More Arguments\n");

        return fail;

    }

    else return firstTok;

    }

}

명령어의 두 번째 토큰은 일부 명령어는 사용하고 일부 명령어는 필요 없다는 점에서 특별하다. 따라서 각각의 경우에 대해 예외처리를 하는 함수이다.

/\*

    checkSecondToken check validation of second token.

    As explained above 'checkFirstToken' function, it check the second argument of string.

    Second string has speciality in that some command like help doesn't have second argument.

    Therefore, to commands don't have second argument, check if they really don't have second argument. if they have, return fail.

    To commands have second argument, if they actually don't have argument, return fail.

    In right case, return command code.

\*/

int executeCommand(int command, char\* Arg2,char\* Arg3,char\* Arg4,char\* Arg5,historyLink\* Hist,opCode\* Op[])

{

    if(command == help)return executeHelp();

    else if(command == dir)return executeDir();

    else if(command == history)return executeHistory(Hist);

    else if(command == dump){

    int dumpArgNum = validDumpArg(Arg2,Arg3,Arg4);

    if(dumpArgNum == fail)return fail;

    return executeDump(Arg2,Arg3,dumpArgNum);

    }

    else if(command == edit){

    int editArgNum = validEditArg(Arg2,Arg3,Arg4);

    if(editArgNum == fail)return fail;

    return executeEdit(Arg2,Arg3);

    }

    else if(command == fill){

    int fillArgNum = validFillArg(Arg2,Arg3,Arg4,Arg5);

    if(fillArgNum == fail)return fail;

    return executeFill(Arg2,Arg3,Arg4);

    }

    else if(command == reset)return executeReset();

    else if(command == opcode)

    {

    if(validOpcodeArg(Arg2,Arg3) == fail)return fail;

    return executeOpcode(Arg2,Op);

    }

    else if(command == opcodelist)

    {

    return executeOpcodelist(Op);

    }

    else return fail;

}

이전 함수에서 명령어 번호와 딸려오는 argument들을 받아서 각각의 명령을 수행하는 함수이다. 이때, argument에 대해서 validation이 필요한 경우 이를 수행하는 함수를 동작한 뒤 execution을 시행한다. 각각 명령어들이 수행 성공시 success 를, 실패시 fail을 리턴한다.

/\*

   executeCommand get code of command, and 4 additional argument on command string.

   If there's no argument, get NULL.

   Hist and Op are data for history and opcode.

   According to command code, it executes function.

   In some functions, validate arguments before actually execute command.

   If these process has error, return fail.

   Or else, return code number of command.

\*/

void addHistory(historyLink\* head,char \*record)

{

    historyLink\* temp = head;

    while(temp->next != NULL)

    {

        temp=temp->next;

    }

    historyLink \*newLink = (historyLink\*)malloc(sizeof(historyLink));

    temp->next = newLink;

    strcpy(temp->next->command,record);

    temp->next->next = NULL;

}

명령어가 완벽하게 수행됐을 때 이를 히스토리에 추가하는 함수이다.

히스토리는 링크드리스트의 형식을 취하기 때문에 링크드리스트의 마지막까지 가서 새로 메모리를 할당하고 이를 이어붙인다.

/\*

   addHistory function is called when a command successfully executed.

   Succeded execution means an "execute...." like function return success.

   By using linked list structure, it saves the command sequentially.

\*/

int executeHelp()

{

    printf("h[elp]\n");

    printf("d[ir]\n");

    printf("q[uit]\n");

    printf("hi[story]\n");

    printf("du[mp] [start,end]\n");

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

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

    printf("reset\n");

    printf("opcode mnemonic\n");

    printf("opcodelist\n");

    return success;

}

Help 명령어를 수행하는 함수.

/\*

   executeHelp is a function that executes help command.

   It prints all possible commands.

   After print, return success.

\*/

int executeDir()

{

    struct dirent \*\*dirList;

    struct stat exeCheck;

    int fileCount;

    int count;

    fileCount = scandir(".", &dirList, NULL, alphasort);

    for(count = 0; count < fileCount; count++)

    {

    printf("%s",dirList[count]->d\_name);

    if(stat(dirList[count]->d\_name,&exeCheck)==0 && exeCheck.st\_mode & S\_IFDIR)

    {

        printf("/");

    }

    else if(stat(dirList[count]->d\_name,&exeCheck)==0 && exeCheck.st\_mode & S\_IXUSR)

    {

        printf("\*");

    }

    printf("    ");

    }

    printf("\n");

    for(count = 0; count < fileCount; count++)

    {

    free(dirList[count]);

    }

    free(dirList);

    return success;

}

Dir을 수행하는 함수. Dir 명령어를 수행하기 위해 dirent.h 와 sys/stat.h 를 임포트하여 수행하였다. 현재 디렉토리 내의 모든 파일 및 디렉토리의 리스트를 받고, 각 파일의 상태를 표시하는 8비트를 받아 디렉토리인지 실행가능한지에 따라 추가적인 출력을 해주는 함수

/\*

   executeDir is a function that executes dir command.

   By using dirent, sys/stat library, get lists of current directory.

   Next, check bits of file status.

   If the file is directory, print it with '/'

   If the file is executable, print it with '\*'

   Else, print the file name.

   After print, return success.

\*/

int executeHistory(historyLink\* head)

{

    historyLink\* temp=head;

    int histNum = 1;

    if(head->next == NULL)

    {

    return success;

    }

    else

    {

    temp = temp->next;

    }

    while(temp->next != NULL)

    {

    printf("%-3d ",histNum);

    printf("%s\n",temp->command);

    temp = temp->next;

    histNum++;

    }

    printf("%-3d ",histNum);

    printf("%s\n",temp->command);

    return success;

}

히스토리를 수행하는 함수. 명령이 성공할 때만 addHistory가 수행되기 때문에 이 함수에서는 잘못된 명령어에 대해 걱정할 필요가 없다.

링크드리스트의 처음부터 끝까지 탐색하면서 출력한다.

/\*

   executeHistory is a function that executes history command.

   The function needs head as parameter, which is a linked list.

   Head contains a record of successfully executed command.

   Until the record exists, print it.

   After printing, return success.

\*/

int validDumpArg(char\* Arg2,char\* Arg3,char\* Arg4)

{

    if(Arg4 != NULL)

    {

    printf("Too many arguments\n");

    return fail;

    }

    if(Arg2 == NULL)return 0;

    int argNum2 = stringToHex(Arg2,noComma);

    if (argNum2 == fail && Arg3 == NULL){

    printf("Invalid Command Format.\n");

    return fail;

    }

    if(Arg3 == NULL)return 1;

    argNum2 = stringToHex(Arg2,hasComma);

    int argNum3 = stringToHex(Arg3,noComma);

    if(argNum3 == fail || argNum2 == fail){

    printf("Invalid Command Format.\n");

    return fail;

    }

    return 2;

}

Dump 명령어를 수행하기 전에 validation을 체크하는 함수. 이때 dump는 argument가 0개, 1개, 2개일 수 있기 때문에 특별함을 지닌다. 각 argument마다 stringToHex를 이용해 스트링이 argument로써 validation을 확인한다. 이때 argument가 ,가 필요할 수도 있는 경우가 있기 때문에 stringToHex에 noComma, hasComma의 값을 전해줘서 검사에 사용한다. 명령에 성공시 명령어에 추가적인 인자의 개수를 리턴한다

/\*

   validDumpArg is a function that checks validity of arguments for Dump command.

   Use stringToHex function to find error on string format and to get hexadecimal value of arguments.

   In normal case, it returns number of argument, since dump can have 0-2 additional aguments.

\*/

void dumpNoArg()

{

    int count;

    int byte;

    for(count = dumpAddress; count < dumpAddress+10; count++)

    {

    if(count >= memSize)

    {

        dumpAddress=0;

        return;

    }

    printf("%04x0 ",count);

    for(byte=0; byte < 16; byte++)

    {

        printf("%02x ",sicMemory[count][byte]);

    }

    printf(";");

    for(byte=0; byte < 16; byte++)

    {

        if(sicMemory[count][byte] == 0)printf(".");

        else printf("%c",sicMemory[count][byte]);

    }

    printf("\n");

    }

    dumpAddress = count;

}

명령어에 추가적인 인자가 없는 경우에 출력을 하는 방식. 이때 dumpAddress에 어느 주소까지 프린트했는지 저장한다.

/\*

   dumpNoArg is a print function when no additional argument exists.

   It uses dumpAddress to save last address of mememory executed before.

\*/

void dumpWithArgs(int h1, int h2)

{

    int count;

    int byte;

    int start = h1/16;

    int end;

    if(h2 == -1)

    {

    end=start+10;

    if(h1 % 16 != 0)end += 1;

    h2 = h1+159;

    }

    else

    {

    end = h2/16+1;

    //if(h2 % 16 != 0)end += 1;

    }

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

    {

    if(count >= memSize)

    {

        dumpAddress=0;

        return;

    }

    printf("%04x0 ",count);

    for(byte=0; byte < 16; byte++)

    {

        if(count\*16+byte < h1)printf("   ");

        else if(count == end-1){

        if(byte > h2%16)printf("   ");

        else printf("%02x ",sicMemory[count][byte]);

        }

        else printf("%02x ",sicMemory[count][byte]);

    }

    printf("; ");

    for(byte=0; byte < 16; byte++)

    {

        if(count\*16+byte < h1)printf(".");

        else if(count == end-1){

        if(byte > h2%16)printf(".");

        else if(sicMemory[count][byte] == 0)printf(".");

        else printf("%c",sicMemory[count][byte]);

        }

        else if(sicMemory[count][byte] == 0)printf(".");

        else printf("%c",sicMemory[count][byte]);

    }

    printf("\n");

    }

}

명령어에 추가적인 argument가 있었을 때 출력하는 함수

/\*

   dumpWithArgs is a print function when there are additional arguments in command string.

   When h2 is -1, it means only a argument were added.

   If h2 exists, it prints from h1 to h2.

\*/

int executeDump(char\* Arg2,char\* Arg3,int argNum)

{

    int hex1;

    int hex2;

    if(argNum == 0)

    {

    dumpNoArg();

    }

    else if(argNum == 1)

    {

    hex1 = stringToHex(Arg2,noComma);

    if(hex1 < 0 || hex1 >= memSize\*16)

    {

        printf("Memory Out Of Bound\n");

        return fail;

    }

    dumpWithArgs(hex1,-1);

    }

    else if(argNum == 2)

    {

    hex1 = stringToHex(Arg2,hasComma);

    hex2 = stringToHex(Arg3,noComma);

    if(hex1 < 0 || hex1 >= memSize\*16)

    {

        printf("Memory Out Of Bound\n");

        return fail;

    }

    if(hex2 < 0 || hex2 >= memSize\*16)

    {

        printf("Memory Out Of Bound\n");

        return fail;

    }

    if(hex1 > hex2)

    {

        printf("Start Argument Should Not Smaller Than End Argument\n");

        return fail;

    }

    dumpWithArgs(hex1,hex2);

    }

    return success;

}

Dump 명령어를 수행하는 함수이다. 이때 argument의 개수에 따라서 print수행하는 함수를 다르게 사용한다. 수행 이전에는 16진수로 바뀐 값들의 범위를 검사하고 범위 검사가 올바르지 않을 경우 fail을 리턴한다. 성공시에만 success를 리턴한다.

/\*

   executeDump is a function that executes dump command.

   Dump command need printed output. Therefore, it uses dumpWithArgs and dumpNoArg to print.

   Before execute these printing functions, it changes string formed arguments to hexadecimal using stringToHex.

   If there's error during the process, return fail.

\*/

int validEditArg(char\* Arg2,char\* Arg3,char\* Arg4)

{

    if(Arg4 != NULL){

    printf("Too many argument\n");

    return fail;

    }

    if(Arg2 == NULL){

    printf("Invalid Command Needs Address, Value.\n");

    return fail;

    }

    int argNum2 = stringToHex(Arg2,hasComma);

    if (argNum2 == fail)

    {

    printf("Invalid Command Format.\n");

    return fail;

    }

    if(Arg3 == NULL)

    {

    printf("Invalid Command Needs Address, Value.\n");

    return fail;

    }

    int argNum3 = stringToHex(Arg3,noComma);

    if(argNum3 == fail)

    {

    printf("Invalid Command Format.\n");

    return fail;

    }

    return success;

}

Edit 을 수행하기 전에 validation을 하는 함수이다. Edit은 추가적인 인자 2개가 들어가야한다.

/\*

   validEditArg is a function that validates arguments for edit command.

   Edit function needs 2 additional arguments. If the command string does not contain proper command, return fail.

\*/

int executeEdit(char\* Arg2, char\* Arg3)

{

    int address = stringToHex(Arg2,hasComma);

    int value = stringToHex(Arg3,noComma);

    if(address < 0 || address >= memSize\*16)

    {

    printf("Memory Out Of Bound.\n");

    return fail;

    }

    if(value < 0 || value >= 256)

    {

    printf("Value Out Of Bound.\n");

    return fail;

    }

    int line = address/16;

    int col = address%16;

    sicMemory[line][col] = value;

    return success;

}

Edit을 실제로 수행하는 함수이다. 이때 address가 1MB의 범위를 벗어나거나 value가 0~255의 범위를 벗어나는 경우 fail을 리턴한다. 그렇지 않다면 메모리의 해당 주소의 값을 value 값으로 변환시키고 success 를 리턴한다.

/\*

   executeEdit is a function that executes edit function.

   Edit changes value of allocated sic machine's memory.

   Since sic machine's memories are allocated through global memory, the function can directly access memory.

   If selected memory address is out of bound, return fail.

\*/

int validFillArg(char\* Arg2,char\* Arg3,char\* Arg4,char\* Arg5)

{

    if(Arg5 != NULL)

    {

    printf("Too many argument\n");

    return fail;

    }

    if(Arg2 == NULL)

    {

    printf("Fill Needs 3 Arguments\n");

    return fail;

    }

    int argNum2 = stringToHex(Arg2,hasComma);

    if (argNum2 == fail)

    {

    printf("Invalid Command Format\n");

    return fail;

    }

    if(Arg3 == NULL)

    {

    printf("Fill Needs 3 Arguments\n");

    return fail;

    }

    int argNum3 = stringToHex(Arg3,hasComma);

    if(argNum3 == fail)

    {

    printf("Invalid Command Format\n");

    return fail;

    }

    if(Arg4 == NULL)

    {

    printf("Fill Needs 3 Arguments\n");

    return fail;

    }

    int argNum4 = stringToHex(Arg4,noComma);

    if(argNum4 == fail)

    {

    printf("Invalid Command Format.\n");

    return fail;

    }

    return success;

}

Fill의 수행 전 validation을 수행하는 함수이다. Fill의 경우 추가적인 3개의 argument가 필요하다.

/\*

   validFillArg is a function that checks validity of arguments for fill function.

   Fill function must have 3 additional arguments.

   Only when 3 arguments are exist and they are valid returns success.

\*/

int executeFill(char\* Arg2,char\* Arg3, char\* Arg4)

{

    int start = stringToHex(Arg2,hasComma);

    int end = stringToHex(Arg3,hasComma);

    int value = stringToHex(Arg4,noComma);

    if(start > end)

    {

    printf("Start Argument Should Not Bigger Than End Argument\n");

    return fail;

    }

    if(start < 0 || start >= memSize\*16)

    {

    printf("Access Memory Out Of Range\n");

    return fail;

    }

    if(end < 0 || end >= memSize\*16)

    {

    printf("Access Memory Out Of Range\n");

    return fail;

    }

    if(value < 0|| value >= 256)

    {

    printf("Value Out Of Bound\n");

    return fail;

    }

    int count;

    int line,col;

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

    {

    line = count/16;

    col = count%16;

    sicMemory[line][col] = value;

    }

    return success;

}

실제 fill을 수행하는 함수이다. 이때, 값을 완전히 넣기 전에 start, end의 메모리 주소가 1MB의 범위를 벗어나는 경우나 value의 값이 0~255의 범위를 벗어나는 경우 fail을 리턴한다.

/\*

   executeFill is a function that executes fill command.

   Fill command change memory value to Value argument from start to end address.

   If arguments are invalid, return fail and doesn't change memory and return fail.

   If valid, change the memory value of sicMemory and return success.

\*/

int executeReset()

{

    int count;

    int count2;

    for(count = 0; count < memSize; count++)

    {

    for(count2 = 0; count2 < 16; count2++)

       sicMemory[count][count2] = 0;

    }

    return success;

}

Reset 명령어를 수행하는 함수이다. 0~1MB의 메모리를 모두 0으로 초기화한다.

/\*

    executeReset is a function that initalize all memory value to zero.

\*/

int validOpcodeArg(char\* Arg2, char\* Arg3)

{

    if(Arg3 != NULL)

    {

    printf("Too many Argument\n");

    return fail;

    }

    if(Arg2 == NULL)

    {

    printf("Opcode Command Needs 1 Argument \n");

    return fail;

    }

    return success;

}

Opcode 명령어를 수행하기 전에 validation을 수행하는 함수이다. 원리는 다른 valid 함수들과 같다.

/\*

   validOpcodeArg is a function that checks validity of arguments for opcode command.

   However, except cases for missing argument or redundant arguments, hash table can solve other exceptions.

   Therefore, this function only checks above cases.

\*/

int executeOpcode(char\* Arg2,opCode\* Op[])

{

    int idx = hashFunc(Arg2);

    opCode\* temp = Op[idx];

    while(temp->next != NULL)

    {

    temp = temp->next;

    if(strcmp(Arg2,temp->op) == 0)

    {

        printf("opcode is %x\n",temp->opCode);

        return success;

    }

    }

    if(strcmp(Arg2,temp->op) == 0)

    {

    printf("opcode is %x\n",temp->opCode);

    return success;

    }

    printf("There's No Opcode Matches.\n");

    return fail;

}

Opcode명령어를 수행하는 함수이다. opCode라는 해시 테이블을 파라미터로 받는다. hashFunc라는 hash function을 이용해서 20 사이즈의 테이블에서 인덱스를 찾는다. 찾은 인덱스를 순회하며 같은 명령어를 가진 노드를 찾는다. 찾지 못할 경우 fail, 찾을 경우 해당 명령어의 opcode를 출력하고 success를 리턴한다.

/\*

   executeOpcode is a function that executes opcode command.

   it uses Op as parameter, which is 20 size hash table saving opcode commands.

   By using hashFunc, it can find matched index number of hash table.

   Explanation of hashFunc will be covered below.

   If there's no matched opcode, print no matched result and return fail.

   Otherwise, return success and print matched result.

\*/

int executeOpcodelist(opCode\* Op[])

{

    int count;

    opCode\* temp;

    for(count = 0; count < 20; count++)

    {

    temp = Op[count];

    if(temp->next == NULL)continue;

    printf("%d : ",count);

    while(temp->next != NULL)

    {

        temp = temp->next;

        printf("[%s,%x] ->",temp->op,temp->opCode);

    }

    printf("[%s,%x]\n",temp->op,temp->opCode);

    }

    return success;

}

Opcodelist의 명령어를 수행하는 함수이다. 모든 해시 테이블의 인덱스에 대해서 NULL까지 연결 노드를 탐색하며 출력한다. 해시 테이블을 만드는 것은 다른 함수에서 미리 시행하기에 이 함수에서는 해시 테이블의 유무, 오류를 판단할 이유가 없다.

/\*

   executeOpcodeList is a function that executes opcodelist command.

   opcodelist commands do only print the list of opcode.

   When this program start, the opcode automatically saved to linked list formed hash table Op.

   Therefore, this function do only, for 20 size hash table, print the opcode.

\*/

int getOrder(historyLink\* histHead, opCode\* opHead[]){

    char command[500];

    char commandCopy[500];

    char \*splitedCommand;

    char \*secArg;

    char \*thirdArg;

    char \*fourthArg;

    char \*fifthArg;

    int commandCode;

    int ArgCode;

    fgets(command,sizeof(command)-1,stdin);

    command[strlen(command)-1] = 0;

    strcpy(commandCopy,command);

    if(strlen(command) >= commandLen)

    {

    printf("Command Is Too Long\n");

    return fail;

    }

    splitedCommand = strtok(command," ");

    if(splitedCommand == NULL)return fail;

    commandCode = checkFirstToken(splitedCommand);

    if(commandCode == fail)

    {

    printf("Cannot Understand The Command. Please Check Help Command.\n");

    return fail;

    }

    secArg = strtok(NULL," ");

    ArgCode = checkSecondToken(secArg,commandCode);

    if(ArgCode == fail)

    {

    return fail;

    }

    thirdArg = strtok(NULL," ");

    fourthArg = strtok(NULL, " ");

    fifthArg = strtok(NULL, " ");

    if(executeCommand(commandCode,secArg,thirdArg,fourthArg,fifthArg,histHead,opHead) == success)

    {

    addHistory(histHead,commandCopy);

    }

    return commandCode;

}

getOrder 함수는 명령 프롬프트의 주 기능을 수행하는 함수이다. 명령어로 받은 string을 토큰으로 나누고, 이를 일부 검사한 뒤 executeCommand로 넘겨주는 역할을 한다. executeCommand는 적합한 명령어에 대해 검사 후 명령 수행의 역할을 하기 때문에, getOrder은 명령어의 적합성을 판단하고, 토큰으로 나눠서 executeCommand에 보내는 역할을 맡아서 수행한다. 또한 executeCommand가 success를 리턴할 경우 성공한 명령이라 간주해 addHistory를 수행하기도 해야한다. 또한 성공한 command의 command code를 넘겨주어 만약 quit이 들어올 경우 프로그램이 종료되는 것도 돕는다.

/\*

   getOrder is main function of command line.

   It is executed in while loop acting as command line.

   This function gets a string from user and execute command.

   Before executing command, it first cut string with space.

   And then, it checks validity of the command through above functions.

   For each command, it return commandCode to main function.

   If main function get commandCode as quit, it finishes the program.

\*/

void freeHistory(historyLink \*head)

{

    historyLink \*temp=head;

    while(temp->next != NULL)

    {

    temp=temp->next;

    free(head);

    head = temp;

    }

    free(temp);

}

History를 저장하는 링크드 리스트의 메모리를 해제하는 함수

/\*

   freeHistory function frees memory of linked list with historyLink structure.

   It is executed after quit command are entered, thereby while loop is terminated.

\*/

void sicMemoryAllocation(){

    sicMemory = (char \*\*)malloc(sizeof(char\*)\*memSize);

    int count;

    for(count = 0; count < memSize; count++)

    {

    sicMemory[count] = (char \*)malloc(sizeof(char)\*16);

    }

}

1MB를 할당하는 sicMemory의 메모리를 할당하는 함수

/\*

   At the very start of program, sicMemoryAllocation is called to allocate 1MB memory for sic/xe machine.

\*/

void sicMemoryFree()

{

    int count;

    for(count = 0; count < memSize; count++)

    {

    free(sicMemory[count]);

    }

    free(sicMemory);

}

1MB를 할당하는 sicMemory의 메모리를 해제하는 함수

/\*

   sicMemoryFree function frees memory of sic/xe machine's allocated memory.

   It is executed after quit command are entered, thereby while loop is terminated.

\*/

void opcodeInitialize(opCode\* head[])

{

    int count;

    for(count = 0; count < 20; count++)

    {

    head[count] = (opCode\*)malloc(sizeof(opCode));

    head[count]->next = NULL;

    head[count]->opCode = -1;

    head[count]->format[0] = 0;

    head[count]->format[1] = 0;

    }

}

해시 테이블의 헤드들을 초기화 하는 함수이다.

/\*

   opcodeInitialize is a function that initialize heads of hash table.

   It is executed at the very start of program.

\*/

void opcodeFree(opCode\* head[])

{

    opCode\* temp;

    opCode\* rear;

    int count;

    for(count = 0; count < 20; count++)

    {

    temp = head[count];

    rear = temp;

    while(temp->next != NULL)

    {

        temp = temp->next;

        free(rear);

        rear = temp;

    }

    }

}

opcode들을 위한 해시 테이블의 메모리를 해제하는 함수

/\*

   opcodeFree function frees memory of hash table for opcode.

   It is executed only after quit command are entered.

\*/

int hashFunc(char \*command)

{

    int count;

    int idx=0;

    for(count = 0; count < strlen(command);count++)

    {

    idx +=(int) command[count];

    }

    return idx%20;

}

hashFunc함수는 해시 테이블의 인덱스를 지정하는 hash function이다. 원리는

1. 스트링의 모든 캐릭터를 더한다.
2. 더한 값을 20으로 나눈 나머지를 인덱스로 사용한다.

/\*

    hashFunc is an algorithm to save opcode through an efficient way in aspect of memory, time.

    When an opcode is entered, add all ascii value of each character.

    Then, return remainder of the value with 20.

    By using this function, it can find the index of hash table only with the string of opcode.

\*/

void buildOpTable(opCode\* head[])

{

    FILE\* opTxt;

    char command[10];

    char format[10];

    int code;

    int hashIdx;

    opCode\* temp;

    opCode\* allocator;

    opTxt = fopen("opcode.txt","r");

    if(opTxt == NULL)printf("opcode.txt does not exist. You cannot execute opcode\_concerned command.");

    while(fscanf(opTxt,"%x  %s  %s\n",&code,command,format) != EOF)

    {

    hashIdx = hashFunc(command);

    temp = (opCode\*)malloc(sizeof(opCode));

    allocator = head[hashIdx];

    while(allocator->next != NULL)allocator = allocator->next;

    allocator->next = temp;

    allocator = allocator->next;

    allocator->next = NULL;

    allocator->opCode = code;

    strcpy(allocator->op,command);

    if(strlen(format) == 1)allocator->format[0] = atoi(format);

    else{

        allocator->format[0] = format[0]-'0';

        allocator->format[1] = format[1]-'0';

    }

    }

}

Opcode에 사용하는 데이터를 텍스트에서 로드해 해시 테이블에 저장하는 함수이다.

Opcode.txt의 모든 데이터를 받아서 각 줄마다 해시 테이블에 저장한다. 이때 저장하는 인덱스는 hashFunc라는 함수를 사용한다.

/\*

    buildOpTable function get head pointers of hash tables.

    It opens opcode.txt and read data of opcode.

    Using hashFunc, it saves data to 20 sized hash table.

    it is called at the very start of program.

\*/

int main(){

    int commandStatus;

    historyLink\* historyHead;

    opCode\* opHead[20];

    opcodeInitialize(opHead);

    buildOpTable(opHead);

    historyHead = malloc(sizeof(historyLink));

    sicMemoryAllocation();

    while(1){

    printf("sicsim> ");

    commandStatus = getOrder(historyHead,opHead);

    if (commandStatus == quit)break;

    }

    opcodeFree(opHead);

    sicMemoryFree();

    freeHistory(historyHead);

    return 0;

}

메인 함수에서는 많은 기능을 수행하지 않는다.

전역 변수들의 초기화, 해시 펑션 빌드, prompt를 위한 무한 루프 실행, 메모리 해제 를 수행한다.

나머지 기능은 모두 getOrder에서 수행한다. 이때, getOrder가 quit 명령어였음을 알리면 프로그램은 종료된다.

/\*

   main function do only initialization, start of while loop, and deallocation of memory

\*/