**Федеральное государственное бюджетное образовательное учреждение «Сибирский государственный университет телекоммуникаций и информатики»**

Кафедра ВС

**КУРСОВАЯ РАБОТА**

По дисциплине «Архитектура ЭВМ»

Вариант 13

Выполнил:

студент гр. ИВ-621

Фамилия И.О.

Проверил:

Майданов Ю. С.

Новосибирск 2018

**Оглавление**

Постановка задачи..................................................................................................3

Блок схемы алгоритмов..........................................................................................7

Программная реализация……..............................................................................11

Результат работы программы...............................................................................16

Заключение…………………................................................................................17

Литература………………….................................................................................18

Листинг ..................................................................................................................19

**Постановка задачи**

В рамках курсовой работы необходимо доработать модель *Simple Computer* так, чтобы она обрабатывала команды, записанные в оперативной памяти. Система команд представлена в таблице 1. Из пользовательских функций необходимо реализовать только одну согласно варианту задания (номеру вашей учетной записи). Для разработки программ требуется создать трансляторы с языков *Simple Assembler* и *Simple Basic*.

**Обработка команд центральным процессором**

Для выполнения программ моделью *Simple Computer* необходимо реализовать две функции:

**int *ALU*** (*int command, int operand*) – реализует алгоритм работы арифметико-логического устройства. Если при выполнении функции возникла ошибка, которая не позволяет дальше выполнять программу, то функция возвращает -1, иначе 0;

**int *CU*** (void) – обеспечивает работу устройства управления. Обработку команд осуществляет устройство управления. Функция *CU* вызывается либо обработчиком сигнала от системного таймера, если не установлен флаг «игнорирование тактовых импульсов», либо при нажатии на клавишу *t*. Алгоритм работы функции следующий:

1. из оперативной памяти считывается ячейка, адрес которой храниться в регистре *instructionCounter*;

2. полученное значение декодируется как команда;

3. если декодирование невозможно, то устанавливаются флаги «указана неверная команда» и «игнорирование тактовых импульсов» (системный таймер можно отключить) и работа функции прекращается.

4. Если получена арифметическая или логическая операция, то вызывается функция *ALU*, иначе команда выполняется самим устройством управления.

5. Определяется, какая команда должна быть выполнена следующей и адрес еѐ ячейки памяти заносится в регистр *instructionCounter*.

6. Работа функции завершается.

**Транслятор с языка Simple Assembler**

Разработка программ для *Simple Computer* может осуществляться с использованием низкоуровневого языка *Simple Assembler*. Для того чтобы программа могла быть обработана *Simple Computer*необходимо реализовать транслятор, переводящий текст *Simple Assembler* в бинарный формат, которым может быть считан консолью управления.

Пример программы на **Simple Assembler:**

00 READ 09 ; (Ввод А)

01 READ 10 ; (Ввод В)

02 LOAD 09 ; (Загрузка А в аккумулятор)

03 SUB 10 ; (Отнять В)

04 JNEG 07 ; (Переход на 07, если отрицательное)

05 WRITE 09 ; (Вывод А)

06 HALT 00 ; (Останов)

07 WRITE 10 ; (Вывод В)

08 HALT 00 ; (Останов)

09 = +0000 ; (Переменная А)

10 = +9999 ; (Переменная В)

Программа транслируется по строкам, задающим значение одной ячейки памяти. Каждая строка состоит как минимум из трех полей: адрес ячейки памяти, команда (символьное обозначение), операнд. Четвертым полем может быть указан комментарий, который обязательно должен начинаться с символа точка с запятой. Название команд представлено в таблице 1. Дополнительно используется команда =, которая явно задает значение ячейки памяти в формате вывода его на экран консоли (+XXXX).

Команда запуска транслятора должна иметь вид: *sat* файл.*sa* файл.*o*, где файл.*sa*– имя файла, в котором содержится программа на *Simple Assembler*, файл.*o* – результат трансляции.

**Транслятор с языка Simple Basic**

Для упрощения программирования пользователю модели *Simple Computer* должен быть предоставлен транслятор с высокоуровневого языка *Simple Basic*. Файл, содержащий программу на *Simple Basic*, преобразуется в файл с кодом *Simple Assembler*. Затем *Simple Assembler*-файл транслируется в бинарный формат. В языке *Simple Basic* используются следующие операторы: *rem, input, output, goto, if, let, end*.

Пример программы на **Simple Basic:**

10 REM Это комментарий

20 INPUT A

30 INPUT B

40 LET C = A – B

50 IF C < 0 GOTO 20

60 PRINT C

70 END

Каждая строка программы состоит из номера строки, оператора *Simple Basic* и параметров. Номера строк должны следовать в возрастающем порядке. Все команды за исключением команды конца программы могут встречаться в программе многократно. *Simple Basic* должен оперировать с целыми выражениями, включающими операции +, -, \*, и /. Приоритет операций аналогичен C. Для того чтобы изменить порядок вычисления, можно использовать скобки.

Транслятор должен распознавания только букв верхнего регистра, то есть все символы в программе на *Simple Basic* должны быть набраны в верхнем регистре (символ нижнего регистра приведет к ошибке). Имя переменной может состоять только из одной буквы. *Simple Basic* оперирует только с целыми значениями переменных, в нем отсутствует объявление переменных, а упоминание переменной автоматически вызывает еѐ объявление и присваивает ей нулевое значение. Синтаксис языка не позволяет выполнять операций со строками.

**Архитектура *Simple Computer*** - включает следующие функциональные блоки:

· оперативную память;

· внешние устройства;

· центральный процессор.

·

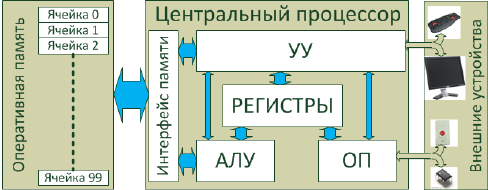


Рисунок 1 – Архитектура вычислительной машины Simple Computer

**Оперативная память**

Оперативная память – это часть *Simple Computer*, где хранятся программа и данные. Память состоит из ячеек (массив), каждая из которых хранит 15 двоичных разрядов. Ячейка – минимальная единица, к которой можно обращаться при доступе к памяти. Все ячейки последовательно пронумерованы целыми числами. Номер ячейки является еѐ адресом и задается 7-миразрядным числом.

**Внешние устройства**

Внешние устройства включают: клавиатуру и монитор, используемые для взаимодействия с пользователем, системный таймер, задающий такты работы *Simple Computer* и кнопку «*Reset*», позволяющую сбросить *Simple Computer* в исходное состояние.

**Центральный процессор**

Выполнение программ осуществляется центральным процессором *Simple Computer*. Процессор состоит из следующих функциональных блоков:

· регистры (аккумулятор, счетчик команд, регистр флагов);

· арифметико-логическое устройство (АЛУ);

· управляющее устройство (УУ);

· обработчик прерываний от внешних устройств (ОП);

· интерфейс доступа к оперативной памяти.

Регистры являются внутренней памятью процессора. Центральный процессор *Simple Computer* имеет: аккумулятор, используемый для временного хранения данных и результатов операций, счетчик команд, указывающий на адрес ячейки памяти, в которой хранится текущая выполняемая команда и регистр флагов, сигнализирующий об определѐнных событиях. Аккумулятор имеет разрядность 15 бит, счетчика команд – 7 бит. Регистр флагов содержит 5 разрядов: переполнение при выполнении операции, ошибка деления на 0, ошибка выхода за границы памяти, игнорирование тактовых импульсов, указана неверная команда.

Арифметико-логическое устройство (англ. arithmetic and logic unit, *ALU*) — блок процессора, который служит для выполнения логических и арифметических преобразований над данными. В качестве данных могут использоваться значения, находящиеся в аккумуляторе, заданные в операнде команды или хранящиеся в оперативной памяти. Результат выполнения операции сохраняется в аккумуляторе или может помещаться в оперативную память. В ходе выполнения операций АЛУ устанавливает значения флагов «деление на 0» и «переполнение».

Управляющее устройство (англ. control unit, *CU*) координирует работу центрального процессора. По сути, именно это устройство отвечает за выполнение программы, записанной в оперативной памяти. В его функции входит: чтение текущей команды из памяти, еѐ декодирование, передача номера команды и операнда в АЛУ, определение следующей выполняемой команды и реализации взаимодействий с клавиатурой и монитором. Выбор очередной команды из оперативной памяти производится по сигналу от системного таймера. Если установлен флаг «игнорирование тактовых импульсов», то эти сигналы устройством управления игнорируются. В ходе выполнения операций устройство управления устанавливает значения флагов «указана неверная команда» и «игнорирование тактовых импульсов».

Обработчик прерываний реагирует на сигналы от системного таймера и кнопки «Reset». При поступлении сигнала от кнопки «Reset» состояние процессора сбрасывается в начальное (значения всех регистров обнуляется и устанавливается флаг «игнорирование сигналов от таймера»). При поступлении сигнала от системного таймера, работать начинает устройство управления.

**Блок схемы используемых алгоритмов**

1. **ALU, CU**
2. **Simple Assembler**
3. **Simple Basic**
4. ALU,CU

Обновление интерфейса

Выбор и выполнение

полученной команды, если таковая есть

Считываем значение ячейки из памяти

Отправляем полученное команду в ALU()

Декодируем полученное значение

**2.Simple Assembler**

Файл с кодом на **Assembler,**

**вых. файл**

Пока не конец файла считываем по строчке

Сохраняем полученные команды в выходной файл

Если строка соответствует команде, то кодируем её

**3.Simple Basic**

Файл с кодом на **Basic,**

**вых. файл**

Пока не конец файла считываем по строчке

В translate\_basic()

Преабразовываем команду из basic в assembler

Вызов функции process\_undef\_goto,

правка индексов

Добавление заранее заданных данных

**Программная реализация**

Ниже указанны функции используемые для разработки простейшей вычислительной машины Simple Computer и даны краткие характеристики.

**interface.h**

int mi\_printinterface(int select\_x, int select\_y); *- отрисовка всего интерфейса*

int mi\_printmemory(int x, int y, int select\_x, int select\_y); *- отрисовка памяти*

int mi\_printflags(int x, int y); *- отрисовка флагов*

int mi\_printkeytooltip(int x, int y); *- отрисовка подсказки по клавишам*

int mi\_printselectedmemory(int x, int y, int address); *- отрисовка выбранной ячейки в виде bigchar*

int mi\_printinstrutioncounter(int x, int y, int insp\_reg); *- отрисовка счетчика инструкций*

int mi\_printaccumulator(int x, int y, int acc\_reg); *- отрисовка аккумулятора*

int mi\_printlog(int x, int y); *- отрисовка журнала*

**bigchars.h**

int bc\_printA(const char \*str); - *выводит символ*

int bc\_box(int x1, int y1, int w, int h);- *выводит рамку*

int bc\_printbigchar (int \*big, int x, int y, enum colors clr1, enum colors clr2); - *С помощью по битового выставления рисует ячейку памяти большими символами.*

int bc\_setbigcharpos(int \*big, int x, int y, int value); - *устанавливает значение знакоместа "большого символа" в строке x и столбце y в значение value*

int bc\_getbigcharpos(int \*big, int x, int y, int \*value); - *возвращает значение позиции в "большом символе" в строке x и столбце y;*

int bc\_bigcharwrite(int fd, int \*big, int count); *-записывает заданное число "больших символов" в файл. Формат записи определяется пользователем;*

int bc\_bigcharread(int fd, int \*big, int need\_count, int \*count); - *считывает из файла заданное количество "больших символов".*

int bc\_bigprintint(int x, int y, enum colors clr1, enum colors clr2, int value); - *выводит число в десятичном представлении*

**memory.h**

int sc\_memoryInit();– *инициализирует оперативную память Simple Computer*

int sc\_memorySet(int address, int value); - *задает значение указанной ячейки памяти как value.*

int sc\_memoryGet(int address, int \*value); – *возвращает значение указан- ной ячейки памяти в value*

int sc\_memorySave(char \*filename); - *сохраняет содержимое памяти в файл в бинарном виде (используя функцию write или fwrite);*

int sc\_memoryLoad(char \*filename); - *загружает из указанного файла содер- жимое оперативной памяти (используя функцию read или fread);*

int sc\_regInit(); - *– инициализирует регистр флагов нулевым значением;*

int sc\_regSet(int num\_register, int value); – *устанавливает значение указанно- го регистра флагов.*

int sc\_regGet(int num\_register, int \*value); - *возвращает значение указанного флага*.

int sc\_commandEncode(int command, int operand, int \*value); - *кодирует команду с указанным номером и операндом и помещает результат в value*.

int sc\_commandDecode(int value, int &command, int \*operand); - *декодирует значение как команду Simple Computer.*

**myreadkey.h**

int rk\_readkey(enum keys \*key); - *Определяет какая, клавиша была нажата*

int rk\_mytermsave(); - *Сохраняет настройки терминала*

int rk\_mytermrestore(); *- Возвращает терминалу стандартные настройки*

int rk\_mytermregime(int regime, int vtime, int vmin, int echo, int sigint); - *переключает терминала между режимами*

**myterm.h**

int mt\_clrscr(); *производит очистку и перемещение курсора в левый верхний угол экрана;*

int mt\_gotoXY(int x, int y); *перемещает курсор в указанную позицию*

int mt\_getscreensize(int \*rows, int \*cols); - *определяет размер экрана терминала (количество строк и столбцов);*

int mt\_setfgcolor(enum colors); *устанавливает цвет последующих выводимых символов.*

int mt\_setbgcolor(enum colors); *- устанавливает цвет фона последующих вы- водимых символов.*

int mt\_resetcolor(); *- востанавливает стандартные цвета*

int mt\_resetcursor(); *- перемещает курсор в влевый верхний угол экрана*

**cpu.h**

int ALU(int command, int operand); – *реализует алгоритм работы арифметико- логического устройства.*

int CU() – *Выполняет следуйщую инструкцию.*

**lexer.h(assembler)**

int lexer(char \*infile, char \*outfile); *– производит трансляцию simple assamblr в бинарный код*

**basiclexer.h**

int get\_var\_address(char \*cstr); *- получение адреса указаной переменной.*

int get\_tmp\_var\_address(char cstr); *- получение адреса временной переменной.*

int get\_num\_var\_address(int val); *- получение адреса где хранится заранее заданое число.*

char \*get\_token(char \*\*str\_p, char \*delimiters[], int size); *- разделяет строку на токены используя разделители в виде массива строк.*

int lexer(char \*infile, char \*outfile); *- производит трансляцию simple basic в simple assembler.*

int tonumber(char \*s, int \*number); *- приводит строку к численому представлению.*

int calcspaces(char \*code\_token, int num); *- подсчитывает количество пробелов до указаного номера символа.*

int translate\_basic(LineStruct \*line\_struct); *- функция трансляции строки simple basic в simple assembler.*

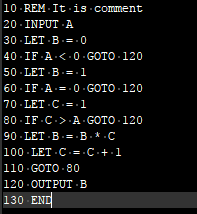
int validate\_operand(char \*cstr); *- проверка верности операнда.*

int validate\_command(char \*cstr); *- проверка верности команды.*

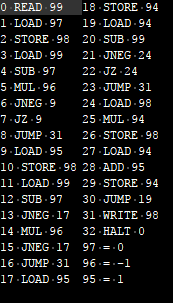
**Результаты проведенного исследования**

В качестве примера была взята программа подсчета факториала, написанная на Simple Basic, далее мы её транслировали на Simple Assembler, и в конце в бинарный формат, который может быть распознан консолью управления.

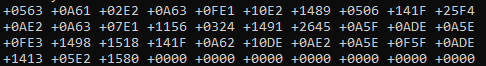
**Basic**

****

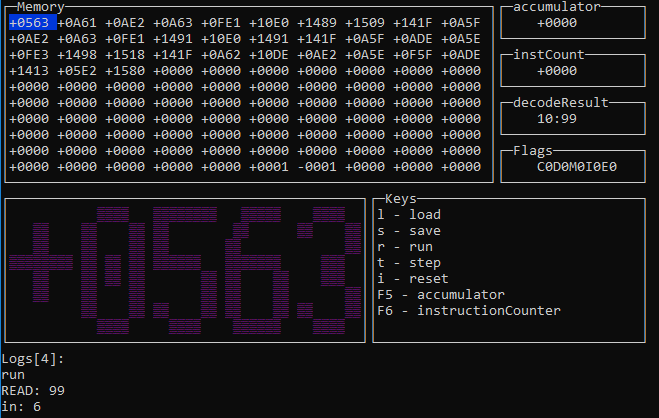
**Assembler**



**default.o**



Введем число 6



Факториал 6! = 6\*5\*4\*3\*2\*1 = 720.



**Заключение**

В рамках курсовой работы была реализована обработка команд центральным процессором с помощью функций: ALU() и CU(). Также был реализован транслятор, переводящий текст Simple Assembler в бинарный формат, который может быть считан консолью управления. И транслятор с высокоуровневого языка Simple Basic в файл, с кодом Simple Assembler. Затем Simple Assembler файл транслируется в бинарный формат.

**Литература**

1. Организация ЭВМ и систем. Практикум // С.Н. Мамойленко, Новосибирск: ГОУ ВПО «Сиб- ГУТИ», 2005 г., URL:

2. Цилькер Б.Я., Орлов С.А. Организация ЭВМ и систем: учебник для ВУЗов. – СПб.: Питер, 2004.

3. Wikipedia –[электронный ресурс]: https://ru.wikipedia.org

**Листинг**

**main.c**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <math.h>**

**#include <unistd.h>**

**#include <fcntl.h>**

**#include <sys/time.h>**

**#include <signal.h>**

**#include <termios.h>**

**#include <sys/ioctl.h>**

**#include <string.h>**

**#include "memory.h"**

**#include "debug.h"**

**#include "myterm.h"**

**#include "interface.h"**

**#include "bigchars.h"**

**#include "myreadkey.h"**

**#include "cpu.h"**

**#include "command.h"**

**int selected\_x = 0, selected\_y = 0;**

**void sigalrm\_proccess(int signo)**

**{**

**int val;**

**if (sc\_regGet(IF, &val)) {**

**return;**

**}**

**if (!val) {**

**int insp\_reg\_tmp = insp\_reg;**

**if (CU()) {**

**sc\_regSet(IF, 1);**

**} else if (insp\_reg\_tmp == insp\_reg) {**

**insp\_reg++;**

**}**

**mi\_printinterface(selected\_x, selected\_y);**

**}**

**struct itimerval nval;**

**nval.it\_interval.tv\_sec = 0;**

**nval.it\_interval.tv\_usec = 0;**

**nval.it\_value.tv\_sec = 0;**

**nval.it\_value.tv\_usec = 100000;**

**setitimer(ITIMER\_REAL, &nval, 0);**

**}**

**void sigusr1\_proccess(int signo)**

**{**

**sc\_memoryInit();**

**sc\_regInit();**

**if (sc\_regSet(IF, 1)) {**

**return;**

**}**

**}**

**void sigint\_proccess(int signo)**

**{**

**rk\_mytermrestor();**

**exit(0);**

**}**

**void sigwinch\_proccess(int signo)**

**{**

**mt\_clrsrc();**

**mi\_printinterface(selected\_x, selected\_y);**

**}**

**char \*need\_load = 0;**

**void print\_interface()**

**{**

**rk\_mytermrestor();**

**sc\_memoryInit();**

**sc\_regInit();**

**sc\_regSet(IF, 1);**

**signal(SIGALRM, sigalrm\_proccess);**

**signal(SIGUSR1, sigusr1\_proccess);**

**signal(SIGINT, sigint\_proccess);**

**signal(SIGWINCH, sigwinch\_proccess);**

**struct itimerval nval;**

**//printf("\033[8;80;81t"); // try set console size, actualy bad idea :(**

**nval.it\_interval.tv\_sec = 0;**

**nval.it\_interval.tv\_usec = 0;**

**nval.it\_value.tv\_sec = 1;**

**nval.it\_value.tv\_usec = 0;**

**setitimer(ITIMER\_REAL, &nval, 0); // init timer for incrementing instruction counter**

**if (need\_load) {**

**sc\_memoryLoad(need\_load);**

**}**

**int exit\_flag = 0;**

**int tmp\_val;**

**enum keys key;**

**mt\_clrsrc();**

**mi\_printinterface(selected\_x, selected\_y);**

**while (exit\_flag == 0) {**

**int ignore\_val;**

**if (sc\_regGet(IF, &ignore\_val)) {**

**return;**

**}**

**if (ignore\_val) {**

**rk\_readkey(&key);**

**if (key == KEY\_down && selected\_y < 9) {**

**selected\_y++;**

**} else if (key == KEY\_up && selected\_y > 0) {**

**selected\_y--;**

**} else if (key == KEY\_right && selected\_x < 9) {**

**selected\_x++;**

**} else if (key == KEY\_left && selected\_x > 0) {**

**selected\_x--;**

**} else if ((key >= '0' && key <= '9') || (key >= 'a' && key <= 'f')) {**

**if (sc\_memoryGet(selected\_y \* 10 + selected\_x, &tmp\_val)) {**

**printf("error memory address: %d cant read", selected\_y \* 10 + selected\_x);**

**return;**

**}**

**if (tmp\_val >= 0) {**

**tmp\_val = (((key >= '0' && key <= '9') ? (key - '0') : (key - 'a' + 10)) | (tmp\_val << 4));**

**} else {**

**tmp\_val = 0 - ((((key >= '0' && key <= '9') ? (key - '0') : (key - 'a' + 10)) | ((0 - tmp\_val) << 4)));**

**}**

**if (sc\_memorySet(selected\_y \* 10 + selected\_x, tmp\_val)) {**

**printf("error memory address: %d cant writed", selected\_y \* 10 + selected\_x);**

**return;**

**}**

**} else if (key == KEY\_backspace) {**

**if (sc\_memoryGet(selected\_y \* 10 + selected\_x, &tmp\_val)) {**

**printf("error memory address: %d cant read", selected\_y \* 10 + selected\_x);**

**return;**

**}**

**if (tmp\_val >= 0) {**

**tmp\_val = tmp\_val >> 4 & 0xFFFF;**

**} else {**

**tmp\_val = 0 - (((0 - tmp\_val) & 0xFFFF) >> 4);**

**}**

**if (sc\_memorySet(selected\_y \* 10 + selected\_x, tmp\_val)) {**

**printf("error memory address: %d cant writed", selected\_y \* 10 + selected\_x);**

**return;**

**}**

**} else if (key == KEY\_plus || key == KEY\_minus) {**

**if (sc\_memoryGet(selected\_y \* 10 + selected\_x, &tmp\_val)) {**

**printf("error memory address: %d cant read", selected\_y \* 10 + selected\_x);**

**return;**

**}**

**if (key == KEY\_plus) {**

**tmp\_val = tmp\_val & (~0x4000);**

**}**

**if (key == KEY\_minus) {**

**tmp\_val = tmp\_val | 0x4000;**

**}**

**if (sc\_memorySet(selected\_y \* 10 + selected\_x, tmp\_val)) {**

**printf("error memory address: %d cant writed", selected\_y \* 10 + selected\_x);**

**return;**

**}**

**} else if (key == KEY\_r) {**

**sc\_regSet(IF, 0);**

**strcat(log\_buff, "run\n");**

**} else if (key == KEY\_s) {**

**strcat(log\_buff, "save file name: ");**

**printf("save file name: ");**

**char buff[20];**

**scanf("%s", buff);**

**strcat(log\_buff, buff);**

**strcat(log\_buff, "\n");**

**sc\_memorySave(buff);**

**} else if (key == KEY\_l) {**

**strcat(log\_buff, "load file name: ");**

**printf("load file name: ");**

**char buff[20];**

**scanf("%s", buff);**

**strcat(log\_buff, buff);**

**strcat(log\_buff, "\n");**

**sc\_memoryLoad(buff);**

**} else if (key == KEY\_i) {**

**raise (SIGUSR1);**

**strcat(log\_buff, "reset\n");**

**} else if (key == KEY\_t) {**

**int insp\_reg\_tmp = insp\_reg;**

**if (CU()) {**

**sc\_regSet(IF, 1);**

**} else if (insp\_reg\_tmp == insp\_reg) {**

**insp\_reg++;**

**}**

**strcat(log\_buff, "step\n");**

**selected\_x = insp\_reg % 10;**

**selected\_y = insp\_reg / 10;**

**} else if (key == KEY\_f5) {**

**strcat(log\_buff, "accumulator value: ");**

**printf("accumulator value: ");**

**int buff;**

**scanf("%X", &buff);**

**char buff\_char[10];**

**sprintf(buff\_char, "%X\n", buff);**

**strcat(log\_buff, buff\_char);**

**acc\_reg = buff;**

**mi\_printinterface(selected\_x, selected\_y);**

**} else if (key == KEY\_f6) {**

**strcat(log\_buff, "instructionCounter value: ");**

**printf("instructionCounter value: ");**

**int buff;**

**scanf("%X", &buff);**

**char buff\_char[10];**

**sprintf(buff\_char, "%X\n", buff);**

**strcat(log\_buff, buff\_char);**

**insp\_reg = buff;**

**mi\_printinterface(selected\_x, selected\_y);**

**} else if (key == KEY\_other || key == KEY\_q) {**

**exit\_flag = 1;**

**}**

**mi\_printinterface(selected\_x, selected\_y);**

**}**

**}**

**rk\_mytermrestor();**

**}**

**int main(int argc, char\* argv[])**

**{**

**if (argc == 2) {**

**need\_load = argv[1];**

**}**

**print\_interface();**

**return 0;**

**}**

**cpu.c**

**#include <stdio.h>**

**#include <string.h>**

**#include "interface.h"**

**#include "memory.h"**

**#include "cpu.h"**

**#include "command.h"**

**#define PRINT\_COMMANDS\_INTO\_LOG**

**int ALU(int command, int operand)**

**{**

**int value;**

**char buff[20];**

**#ifdef PRINT\_COMMANDS\_INTO\_LOG**

**int value2;**

**#endif**

**switch (command) {**

**case COMMAND\_ADD:**

**#ifdef PRINT\_COMMANDS\_INTO\_LOG**

**if (sc\_memoryGet(operand, &value2)) {**

**return 1;**

**}**

**sprintf(buff, "ADD: %d[%X]\n", operand, value2);**

**strcat(log\_buff, buff);**

**printf("%s", buff);**

**#endif**

**if (sc\_memoryGet(operand, &value)) {**

**return 1;**

**}**

**acc\_reg = ((acc\_reg & 0x7FFF) + (value & 0x7FFF)) & 0x7FFF;**

**break;**

**case COMMAND\_SUB:**

**#ifdef PRINT\_COMMANDS\_INTO\_LOG**

**if (sc\_memoryGet(operand, &value2)) {**

**return 1;**

**}**

**sprintf(buff, "SUB: %d[%X]\n", operand, value2);**

**strcat(log\_buff, buff);**

**printf("%s", buff);**

**#endif**

**if (sc\_memoryGet(operand, &value)) {**

**return 1;**

**}**

**//acc\_reg -= value;**

**acc\_reg = ((acc\_reg & 0x7FFF) - (value & 0x7FFF)) & 0x7FFF;**

**break;**

**case COMMAND\_DIVIDE:**

**#ifdef PRINT\_COMMANDS\_INTO\_LOG**

**if (sc\_memoryGet(operand, &value2)) {**

**return 1;**

**}**

**sprintf(buff, "DIV: %d[%X]\n", operand, value2);**

**strcat(log\_buff, buff);**

**printf("%s", buff);**

**#endif**

**if (sc\_memoryGet(operand, &value)) {**

**return 1;**

**}**

**acc\_reg = acc\_reg / value; //TODO**

**break;**

**case COMMAND\_MUL:**

**#ifdef PRINT\_COMMANDS\_INTO\_LOG**

**if (sc\_memoryGet(operand, &value2)) {**

**return 1;**

**}**

**sprintf(buff, "MUL: %d[%X]\n", operand, value2);**

**strcat(log\_buff, buff);**

**printf("%s", buff);**

**#endif**

**if (sc\_memoryGet(operand, &value)) {**

**return 1;**

**}**

**if (!(acc\_reg & 0x7FFF) || !(value & 0x7FFF)) {**

**acc\_reg = 0;**

**} else {**

**//acc\_reg \*= value;**

**if (acc\_reg & 0x4000) {**

**if (value & 0x4000) {**

**acc\_reg = (((0 - acc\_reg) & 0x3FFF) \* ((0 - value) & 0x3FFF)) & 0x3FFF;**

**} else {**

**acc\_reg = (((0 - acc\_reg) & 0x3FFF) \* (value & 0x3FFF)) & 0x3FFF;**

**acc\_reg ^= 0x4000;**

**}**

**} else {**

**if (value & 0x4000) {**

**acc\_reg = ((acc\_reg & 0x3FFF) \* ((0 - value) & 0x3FFF)) & 0x3FFF;**

**acc\_reg ^= 0x4000;**

**} else {**

**acc\_reg = ((acc\_reg & 0x3FFF) \* (value & 0x3FFF)) & 0x3FFF;**

**}**

**}**

**}**

**break;**

**// USERS COMMANDS**

**case COMMAND\_NOT:**

**#ifdef PRINT\_COMMANDS\_INTO\_LOG**

**if (sc\_memoryGet(operand, &value2)) {**

**return 1;**

**}**

**sprintf(buff, "NOT: %d[%X]\n", operand, value2);**

**strcat(log\_buff, buff);**

**printf("%s", buff);**

**#endif**

**if (sc\_memorySet(operand, ~acc\_reg)) {**

**return 1;**

**}**

**break;**

**case COMMAND\_AND:**

**#ifdef PRINT\_COMMANDS\_INTO\_LOG**

**if (sc\_memoryGet(operand, &value2)) {**

**return 1;**

**}**

**sprintf(buff, "AND: %d[%X]\n", operand, value2);**

**strcat(log\_buff, buff);**

**printf("%s", buff);**

**#endif**

**if (sc\_memoryGet(operand, &value)) {**

**return 1;**

**}**

**acc\_reg &= value;**

**break;**

**case COMMAND\_OR:**

**#ifdef PRINT\_COMMANDS\_INTO\_LOG**

**if (sc\_memoryGet(operand, &value2)) {**

**return 1;**

**}**

**sprintf(buff, "OR: %d[%X]\n", operand, value2);**

**strcat(log\_buff, buff);**

**printf("%s", buff);**

**#endif**

**if (sc\_memoryGet(operand, &value)) {**

**return 1;**

**}**

**acc\_reg |= value;**

**break;**

**case COMMAND\_XOR:**

**#ifdef PRINT\_COMMANDS\_INTO\_LOG**

**if (sc\_memoryGet(operand, &value2)) {**

**return 1;**

**}**

**sprintf(buff, "XOR: %d[%X]\n", operand, value2);**

**strcat(log\_buff, buff);**

**printf("%s", buff);**

**#endif**

**if (sc\_memoryGet(operand, &value)) {**

**return 1;**

**}**

**acc\_reg ^= value;**

**break;**

**case COMMAND\_RCR:**

**#ifdef PRINT\_COMMANDS\_INTO\_LOG**

**if (sc\_memoryGet(operand, &value2)) {**

**return 1;**

**}**

**sprintf(buff, "RCR: %d[%X]\n", operand, value2);**

**strcat(log\_buff, buff);**

**printf("%s", buff);**

**#endif**

**if (sc\_memoryGet(operand, &value)) {**

**return 1;**

**}**

**acc\_reg = ((value << 1) | ((value & 0x4000) && 1)) & 0x7FFF;**

**break;**

**}**

**return 0;**

**}**

**int CU()**

**{**

**int value;**

**if (sc\_memoryGet(insp\_reg, &value)) {**

**return 1;**

**}**

**int command;**

**int operand;**

**if (sc\_commandDecode(value, &command, &operand)) {**

**sc\_regSet(IF, 1);**

**return 1;**

**}**

**if ((command >= 30 && command <= 33) || (command >= 51 && command <= 54)) {**

**if (ALU(command, operand)) {**

**return 1;**

**}**

**} else {**

**int value;**

**char buff[20];**

**#ifdef PRINT\_COMMANDS\_INTO\_LOG**

**int value2;**

**#endif**

**switch (command) {**

**case COMMAND\_READ: //READ**

**#ifdef PRINT\_COMMANDS\_INTO\_LOG**

**sprintf(buff, "READ: %d\n", operand);**

**strcat(log\_buff, buff);**

**printf("%s", buff);**

**#endif**

**printf("in: ");**

**scanf("%d", &value);**

**sprintf(buff, "in: %d\n", value);**

**strcat(log\_buff, buff);**

**if (sc\_memorySet(operand, value)) {**

**return 1;**

**}**

**break;**

**case COMMAND\_WRITE: //WRITE**

**#ifdef PRINT\_COMMANDS\_INTO\_LOG**

**if (sc\_memoryGet(operand, &value2)) {**

**return 1;**

**}**

**sprintf(buff, "WRITE: %d[%d]\n", operand, value2);**

**strcat(log\_buff, buff);**

**printf("%s", buff);**

**#endif**

**if (sc\_memoryGet(operand, &value)) {**

**return 1;**

**}**

**sprintf(buff, "out: %d\n", value);**

**strcat(log\_buff, buff);**

**printf("%s", buff);**

**break;**

**case COMMAND\_LOAD: //LOAD**

**#ifdef PRINT\_COMMANDS\_INTO\_LOG**

**if (sc\_memoryGet(operand, &value2)) {**

**return 1;**

**}**

**sprintf(buff, "LOAD: %d[%X]\n", operand, value2);**

**strcat(log\_buff, buff);**

**printf("%s", buff);**

**#endif**

**if (sc\_memoryGet(operand, &value)) {**

**return 1;**

**}**

**acc\_reg = value;**

**break;**

**case COMMAND\_STORE: //STORE**

**#ifdef PRINT\_COMMANDS\_INTO\_LOG**

**if (sc\_memoryGet(operand, &value2)) {**

**return 1;**

**}**

**sprintf(buff, "STORE: %d[%X]\n", operand, value2);**

**strcat(log\_buff, buff);**

**printf("%s", buff);**

**#endif**

**if (sc\_memorySet(operand, acc\_reg)) {**

**return 1;**

**}**

**break;**

**case COMMAND\_SET:**

**#ifdef PRINT\_COMMANDS\_INTO\_LOG**

**sprintf(buff, "SET: %d[%X]\n", operand, acc\_reg);**

**strcat(log\_buff, buff);**

**printf("%s", buff);**

**#endif**

**acc\_reg = operand;**

**break;**

**case COMMAND\_JUMP:**

**#ifdef PRINT\_COMMANDS\_INTO\_LOG**

**sprintf(buff, "JUMP: %d[%X]\n", operand, insp\_reg);**

**strcat(log\_buff, buff);**

**printf("%s", buff);**

**#endif**

**insp\_reg = operand;**

**break;**

**case COMMAND\_JNEG:**

**#ifdef PRINT\_COMMANDS\_INTO\_LOG**

**sprintf(buff, "JNEG: %d[%X]\n", operand, insp\_reg);**

**strcat(log\_buff, buff);**

**printf("%s", buff);**

**#endif**

**if (acc\_reg & 0x4000) {**

**insp\_reg = operand;**

**}**

**break;**

**case COMMAND\_JZ:**

**#ifdef PRINT\_COMMANDS\_INTO\_LOG**

**sprintf(buff, "JZ: %d[%X]\n", operand, insp\_reg);**

**strcat(log\_buff, buff);**

**printf("%s", buff);**

**#endif**

**if (acc\_reg == 0) {**

**insp\_reg = operand;**

**}**

**break;**

**case COMMAND\_HALT: //HALT**

**#ifdef PRINT\_COMMANDS\_INTO\_LOG**

**sprintf(buff, "HALT: %d\n", operand);**

**strcat(log\_buff, buff);**

**printf("%s", buff);**

**#endif**

**sc\_regSet(IF, 1);**

**break;**

**case COMMAND\_NOP: //HALT**

**#ifdef PRINT\_COMMANDS\_INTO\_LOG**

**sprintf(buff, "NOP: %d\n", operand);**

**strcat(log\_buff, buff);**

**printf("%s", buff);**

**#endif**

**break;**

**default:**

**sc\_regSet(IF, 1);**

**#ifdef PRINT\_COMMANDS\_INTO\_LOG**

**sprintf(buff, "WRONG COMMAND:%d %d\n", command, operand);**

**strcat(log\_buff, buff);**

**printf("%s", buff);**

**#endif**

**break;**

**}**

**}**

**return 0;**

**}**

**lexer.c(assembler)**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <unistd.h>**

**#include <string.h>**

**#include <ctype.h>**

**#include <sys/types.h>**

**#include <sys/stat.h>**

**#include <fcntl.h>**

**#include "printinfo.h"**

**#include "lexer.h"**

**#include "command.h"**

**#include "myterm.h"**

**#include "memory.h"**

**int cstrtocint(char \*cstr, int \*cint)**

**{**

**if (!strcmp(cstr, "READ")) {**

**\*cint = COMMAND\_READ;**

**} else if (!strcmp(cstr, "WRITE")) {**

**\*cint = COMMAND\_WRITE;**

**} else if (!strcmp(cstr, "LOAD")) {**

**\*cint = COMMAND\_LOAD;**

**} else if (!strcmp(cstr, "STORE")) {**

**\*cint = COMMAND\_STORE;**

**} else if (!strcmp(cstr, "SET")) {**

**\*cint = COMMAND\_SET;**

**} else if (!strcmp(cstr, "ADD")) {**

**\*cint = COMMAND\_ADD;**

**} else if (!strcmp(cstr, "SUB")) {**

**\*cint = COMMAND\_SUB;**

**} else if (!strcmp(cstr, "DIVIDE")) {**

**\*cint = COMMAND\_DIVIDE;**

**} else if (!strcmp(cstr, "MUL")) {**

**\*cint = COMMAND\_MUL;**

**} else if (!strcmp(cstr, "JUMP")) {**

**\*cint = COMMAND\_JUMP;**

**} else if (!strcmp(cstr, "JNEG")) {**

**\*cint = COMMAND\_JNEG;**

**} else if (!strcmp(cstr, "JZ")) {**

**\*cint = COMMAND\_JZ;**

**} else if (!strcmp(cstr, "HALT")) {**

**\*cint = COMMAND\_HALT;**

**} else if (!strcmp(cstr, "NOT")) {**

**\*cint = COMMAND\_NOT;**

**} else if (!strcmp(cstr, "AND")) {**

**\*cint = COMMAND\_AND;**

**} else if (!strcmp(cstr, "OR")) {**

**\*cint = COMMAND\_OR;**

**} else if (!strcmp(cstr, "XOR")) {**

**\*cint = COMMAND\_XOR;**

**} else if (!strcmp(cstr, "NOP")) {**

**\*cint = COMMAND\_NOP;**

**} else if (!strcmp(cstr, "RCR")) {**

**\*cint = COMMAND\_RCR;**

**} else if (!strcmp(cstr, "=")) {**

**\*cint = 0;**

**} else {**

**return -1;**

**}**

**return 0;**

**}**

**int calcspaces(char \*code\_token, int num)**

**{**

**if (num < 0 || !code\_token) {**

**return -1;**

**}**

**int num\_spaces = 0;**

**while (num > 0) {**

**if (\*code\_token == '\t') {**

**num\_spaces += 8 - (num\_spaces % 8);**

**} else {**

**num\_spaces++;**

**}**

**code\_token++;**

**num--;**

**}**

**return num\_spaces;**

**}**

**int tonumber(char \*s, int \*number)**

**{**

**if (s == NULL || \*s == '\0' || isspace(\*s)) {**

**return 0;**

**}**

**char \*p;**

**\*number = (int)strtod(s, &p);**

**return \*p == '\0';**

**}**

**int lexer(char \*infile, char \*outfile)**

**{**

**int indesc = open(infile, O\_RDONLY);**

**if (indesc == -1) {**

**print\_error("error", "cant open infile");**

**}**

**off\_t size\_file = lseek(indesc, 0, SEEK\_END);**

**if (size\_file == -1L) {**

**print\_error("error", "cant read infile size");**

**return -1;**

**}**

**if (lseek(indesc, 0, SEEK\_SET) == -1L) {**

**print\_error("error", "cant read infile size");**

**return -1;**

**}**

**//printf("find %ld bytes\n", size\_file);**

**char \*code\_buff = calloc(size\_file, 1);**

**if (size\_file != read(indesc, code\_buff, size\_file)) {**

**print\_error("error", "cant read infile fully");**

**return -1;**

**}**

**if (close(indesc) == -1) {**

**print\_error("error", "cant close infile");**

**}**

**for (int i = 0; i < strlen(code\_buff); ++i) {**

**if (code\_buff[i] == '\r') {**

**code\_buff[i] = ' ';**

**}**

**}**

**int memory\_buff[100] = {};**

**char code\_line\_buff[100];**

**int num\_line = 0;**

**int error\_flag = 0;**

**char \*saveptr1;**

**char \*code\_line = strtok\_r(code\_buff, "\n", &saveptr1);**

**while (code\_line) {**

**strncpy(code\_line\_buff, code\_line, 99);**

**code\_line\_buff[99] = 0;**

**//printf("%s\n", code\_line);**

**int num\_memory\_cell = -1, command = 0, operand = 0, num\_token = 0;**

**char \*saveptr2;**

**char \*code\_token = strtok\_r(code\_line, " \t\n", &saveptr2);**

**while (code\_token) {**

**int num\_spaces = calcspaces(code\_line\_buff, code\_token - code\_line);**

**//printf("%d\n", num\_spaces);**

**if (num\_token == 0) {**

**int number;**

**if (!tonumber(code\_token, &number) || number < 0 || number > 99) {**

**char buff[200];**

**sprintf(buff, "cell address \"%s\" not exist\n%s\n\E[32m\E[1m^\E[0m", code\_token, code\_line\_buff);**

**print\_error("error", buff);**

**error\_flag = 1;**

**}**

**num\_memory\_cell = number;**

**} else if (num\_token == 1) {**

**if (cstrtocint(code\_token, &command) == -1) {**

**char buff[200];**

**sprintf(buff, "unknown command \"%s\" line %d\n%s\n\E[32m\E[1m%\*c\E[0m", code\_token, num\_line, code\_line\_buff, num\_spaces + 1, '^');**

**print\_error("error", buff);**

**error\_flag = 1;**

**}**

**} else if (num\_token == 2) {**

**int number;**

**if (!tonumber(code\_token, &number) || ((number < 0 || number > 127) && command != 0) || ((number < -0xFFFF || number > 0xFFFF) && command == 0)) {**

**char buff[200];**

**sprintf(buff, "invalid operand \"%s\" not exist\n%s\n\E[32m\E[1m%\*c\E[0m", code\_token, code\_line\_buff, num\_spaces + 1, '^');**

**print\_error("error", buff);**

**error\_flag = 1;**

**}**

**operand = number;**

**} else if (num\_token == 3) {**

**if (code\_token[0] != ';') {**

**char buff[200];**

**sprintf(buff, "unknown token \"%s\" line %d use \";\" for commentary\n%s\n\E[32m\E[1m%\*c\E[0m", code\_token, num\_line, code\_line\_buff, num\_spaces + 1, '^');**

**print\_error("error", buff);**

**error\_flag = 1;**

**}**

**break;**

**} else {**

**break;**

**}**

**code\_token = strtok\_r(0, " \t\n", &saveptr2);**

**num\_token++;**

**}**

**//printf("%d %d %d\n", num\_memory\_cell, command, operand);**

**if (command != 0) {**

**int coded\_command;**

**sc\_commandEncode(command, operand, &coded\_command);**

**memory\_buff[num\_memory\_cell] = coded\_command;**

**} else {**

**if (operand >= 0) {**

**memory\_buff[num\_memory\_cell] = operand & 0x3FFF;**

**} else {**

**memory\_buff[num\_memory\_cell] = ((0 - operand) & 0x3FFF) | 0x4000;**

**}**

**}**

**//printf("%X\n", coded\_command);**

**code\_line = strtok\_r(0, "\n", &saveptr1);**

**num\_line++;**

**//printf("\n");**

**}**

**if (error\_flag) {**

**return -1;**

**}**

**int outdesc = open(outfile, O\_WRONLY | O\_TRUNC | O\_CREAT, -1);**

**if (outdesc == -1) {**

**print\_error("error", "cant open outfile");**

**}**

**if (write(outdesc, memory\_buff, 400) < 0) {**

**print\_error("error", "cant write in outfine");**

**return -1;**

**}**

**if (close(outdesc) == -1) {**

**print\_error("error", "cant close outfile");**

**}**

**return 0;**

**}**

**basiclexer.c**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <unistd.h>**

**#include <string.h>**

**#include <ctype.h>**

**#include <sys/types.h>**

**#include <sys/stat.h>**

**#include <fcntl.h>**

**#include "basicprintinfo.h"**

**#include "basiclexer.h"**

**#include "command.h"**

**#include "myterm.h"**

**#include "memory.h"**

**#include "datalist.c"**

**int basic\_last\_num\_line = 0;**

**int asm\_last\_memory\_cell = 0;**

**int exp\_delim\_size = 9;**

**char \*expression\_delimiters[9] = {"\*", "/", "+", "-", "(", ")", "<", ">", "="};**

**int line\_delim\_size = 7;**

**char \*line\_delimiters[7] = {"GOTO", "END", "IF", "LET", "OUTPUT", "INPUT", "REM"};**

**DataList \*line\_struct\_list;**

**char\* strtok\_r(char \*str, const char \*delim, char \*\*nextp)**

**{**

**char \*ret;**

**if (str == NULL) {**

**str = \*nextp;**

**}**

**str += strspn(str, delim);**

**if (\*str == '\0') {**

**return NULL;**

**}**

**ret = str;**

**str += strcspn(str, delim);**

**if (\*str) {**

**\*str++ = '\0';**

**}**

**\*nextp = str;**

**return ret;**

**}**

**AsmCommandList \*AsmCommandPushEx(AsmCommandList \*list, int command, int operand, int num)**

**{**

**AsmCommandList \*tmp = list;**

**while(tmp->next != 0) {**

**tmp = tmp->next;**

**}**

**if (tmp->init) {**

**tmp->next = calloc(1, sizeof(AsmCommandList));**

**tmp = tmp->next;**

**}**

**tmp->init = 1;**

**tmp->current.command = command;**

**tmp->current.operand = operand;**

**tmp->current.num = num;**

**return tmp;**

**}**

**AsmCommandList \*AsmCommandPush(AsmCommandList \*list, int command, int operand)**

**{**

**return AsmCommandPushEx(list, command, operand, asm\_last\_memory\_cell++);**

**}**

**void AsmCommandCat(AsmCommandList \*list1, AsmCommandList \*list2)**

**{**

**AsmCommandList \*tmp = list1;**

**if (!list1 || !list2 || !list2->init) {**

**return;**

**}**

**while(tmp->next) {**

**tmp = tmp->next;**

**}**

**if (tmp->init) {**

**tmp->next = list2;**

**} else {**

**tmp->init = 1;**

**tmp->current = list2->current;**

**tmp->next = list2->next;**

**}**

**}**

**void AsmCommandPrintEx(AsmCommandList \*list, int out)**

**{**

**//AsmCommandPrintEx(list, 1);**

**AsmCommandList \*node = list;**

**char buff[256];**

**while (node) {**

**if (node->init) {**

**sprintf(buff, "%d ", node->current.num);**

**write(out, buff, strlen(buff));**

**switch (node->current.command) {**

**case COMMAND\_READ:**

**sprintf(buff, "READ");**

**write(out, buff, strlen(buff));**

**break;**

**case COMMAND\_WRITE:**

**sprintf(buff, "WRITE");**

**write(out, buff, strlen(buff));**

**break;**

**case COMMAND\_LOAD:**

**sprintf(buff, "LOAD");**

**write(out, buff, strlen(buff));**

**break;**

**case COMMAND\_STORE:**

**sprintf(buff, "STORE");**

**write(out, buff, strlen(buff));**

**break;**

**case COMMAND\_ADD:**

**sprintf(buff, "ADD");**

**write(out, buff, strlen(buff));**

**break;**

**case COMMAND\_SUB:**

**sprintf(buff, "SUB");**

**write(out, buff, strlen(buff));**

**break;**

**case COMMAND\_DIVIDE:**

**sprintf(buff, "DIVIDE");**

**write(out, buff, strlen(buff));**

**break;**

**case COMMAND\_MUL:**

**sprintf(buff, "MUL");**

**write(out, buff, strlen(buff));**

**break;**

**case COMMAND\_JUMP:**

**sprintf(buff, "JUMP");**

**write(out, buff, strlen(buff));**

**break;**

**case COMMAND\_JNEG:**

**sprintf(buff, "JNEG");**

**write(out, buff, strlen(buff));**

**break;**

**case COMMAND\_JZ:**

**sprintf(buff, "JZ");**

**write(out, buff, strlen(buff));**

**break;**

**case COMMAND\_HALT:**

**sprintf(buff, "HALT");**

**write(out, buff, strlen(buff));**

**break;**

**case COMMAND\_EQUAL:**

**sprintf(buff, "=");**

**write(out, buff, strlen(buff));**

**}**

**sprintf(buff, " %d\n", node->current.operand);**

**write(out, buff, strlen(buff));**

**}**

**node = node->next;**

**}**

**}**

**void AsmCommandPrint(AsmCommandList \*list)**

**{**

**AsmCommandPrintEx(list, 1);**

**}**

**int getPriority(char op)**

**{**

**switch (op)**

**{**

**case '(':**

**return -1; // не выталкивает сам и не дает вытолкнуть себя другим**

**case '\*': case '/':**

**return 1;**

**case '+': case '-':**

**return 2;**

**case '<': case '>': case '=':**

**return 3;**

**default:**

**return 0;**

**}**

**}**

**int canPop(char op1, DataList \*Functions)**

**{**

**if (datalist\_size(Functions) == 0)**

**return 0;**

**int p1 = getPriority(op1);**

**int p2 = getPriority(\*(char\*)datalist\_peek(Functions));**

**if (p1 == 0 || p2 == 0) {**

**printf("error %c\n", op1);**

**}**

**return p1 >= 0 && p2 >= 0 && p1 >= p2;**

**}**

**#define OPERATION\_LESS 1**

**#define OPERATION\_MORE 2**

**#define OPERATION\_EQUAL 3**

**int tmp\_var\_num = 0;**

**char last\_tmp\_var\_in\_acc = -1;**

**int last\_tmp\_var\_in\_acc\_num = -1;**

**int last\_operation = -1;**

**AsmCommandList \*popFunction(DataList \*Operands, DataList \*Functions)**

**{**

**AsmCommandList \*list = calloc(1, sizeof(AsmCommandList));**

**char \*A = datalist\_pop\_back(Operands);**

**char \*B = datalist\_pop\_back(Operands);**

**//char buff[256] = "";**

**//printf("%s %s\n", A, B);**

**if (last\_tmp\_var\_in\_acc != \*B) {**

**if (last\_tmp\_var\_in\_acc != -1) {**

**int var\_add = get\_tmp\_var\_address(last\_tmp\_var\_in\_acc);**

**AsmCommandPush(list, COMMAND\_STORE, var\_add);**

**}**

**if (isalpha(\*B)) {**

**int var\_cell;**

**if (isupper(\*B)) {**

**var\_cell = get\_var\_address(B);**

**} else {**

**var\_cell = get\_tmp\_var\_address(\*B);**

**//printf("%d %s\n", var\_cell, B);**

**}**

**AsmCommandPush(list, COMMAND\_LOAD, var\_cell);**

**} else {**

**int var\_cell = get\_num\_var\_address(atoi(B));**

**AsmCommandPush(list, COMMAND\_LOAD, var\_cell);**

**}**

**}**

**int var\_cell = -1;**

**if (isalpha(\*A)) {**

**if (isupper(\*A)) {**

**var\_cell = get\_var\_address(A);**

**} else {**

**var\_cell = get\_tmp\_var\_address(\*A);**

**//printf("%d %s\n", var\_cell, A);**

**}**

**} else {**

**var\_cell = get\_num\_var\_address(atoi(A));**

**printf("%s %d\n", A, var\_cell);**

**}**

**char operation = \*(char\*)datalist\_pop\_back(Functions);**

**switch (operation)**

**{**

**case '+':**

**if (var\_cell != -1) {**

**AsmCommandPush(list, COMMAND\_ADD, var\_cell);**

**}**

**break;**

**case '-': //int\_vector\_push\_back(Operands, A - B);**

**if (var\_cell != -1) {**

**AsmCommandPush(list, COMMAND\_SUB, var\_cell);**

**}**

**break;**

**case '\*': //int\_vector\_push\_back(Operands, A \* B);**

**if (var\_cell != -1) {**

**AsmCommandPush(list, COMMAND\_MUL, var\_cell);**

**}**

**break;**

**case '/': //int\_vector\_push\_back(Operands, A / B);**

**if (var\_cell != -1) {**

**AsmCommandPush(list, COMMAND\_DIVIDE, var\_cell);**

**}**

**break;**

**case '<':**

**if (var\_cell != -1) {**

**AsmCommandPush(list, COMMAND\_SUB, var\_cell);**

**AsmCommandPush(list, COMMAND\_MUL, get\_num\_var\_address(-1));**

**last\_operation = OPERATION\_LESS;**

**}**

**break;**

**case '>':**

**if (var\_cell != -1) {**

**AsmCommandPush(list, COMMAND\_SUB, var\_cell);**

**last\_operation = OPERATION\_MORE;**

**}**

**break;**

**case '=':**

**if (var\_cell != -1) {**

**AsmCommandPush(list, COMMAND\_SUB, var\_cell);**

**last\_operation = OPERATION\_EQUAL;**

**}**

**break;**

**default:**

**printf("che eto za simvol????? :%c\n", operation);**

**}**

**char \*arr = calloc(2, sizeof(char));**

**arr[0] = 'a' + tmp\_var\_num++;**

**last\_tmp\_var\_in\_acc = arr[0];**

**datalist\_push\_back(Operands, arr);**

**return list;**

**}**

**AsmCommandList \*translate\_expression(char \*cstr)**

**{**

**//char translated\_code[256] = "";**

**tmp\_var\_num = 0;**

**last\_tmp\_var\_in\_acc = -1;**

**last\_tmp\_var\_in\_acc\_num = -1;**

**last\_operation = -1;**

**char \*expression = calloc(strlen(cstr) + 6, sizeof(char));**

**sprintf(expression, "( %s ) ", cstr);**

**char \*expression\_str = calloc(strlen(expression) + 1, sizeof(char));**

**char \*expression\_addres = expression\_str;**

**strcpy(expression\_str, expression);**

**DataList \*Operands = datalist\_new(10);**

**DataList \*Functions = datalist\_new(10);**

**AsmCommandList \*list = calloc(1, sizeof(AsmCommandList));**

**//char \*prev\_token = "";**

**char \*saveptr;**

**char \*code\_token = strtok\_r(expression\_str, " ", &saveptr);**

**while (code\_token) {**

**//printf("che: %s\n", code\_token);**

**if (isdigit(\*code\_token)) {**

**datalist\_push\_back(Operands, code\_token);**

**} else if (isalpha(\*code\_token)) {**

**if (!isupper(\*code\_token)) {**

**return 0;**

**}**

**datalist\_push\_back(Operands, code\_token);**

**} else {**

**int flag = 0;**

**for (int j = 0; j < exp\_delim\_size; j++) {**

**if (!strcmp(code\_token, expression\_delimiters[j])) {**

**flag = 1;**

**}**

**}**

**if (flag) {**

**if (!strcmp(code\_token, ")")) {**

**while (datalist\_size(Functions) > 0 && strcmp(datalist\_peek(Functions), "(")) {**

**//AsmCommandList asm\_code = popFunction(Operands, Functions);**

**AsmCommandCat(list, popFunction(Operands, Functions));**

**//if (asm\_code) {**

**// //strcat(translated\_code[256], asm\_code);**

**//}**

**}**

**datalist\_pop\_back(Functions);**

**} else {**

**while (canPop(\*code\_token, Functions)) { // Если можно вытолкнуть**

**AsmCommandCat(list, popFunction(Operands, Functions)); // то выталкиваем**

**}**

**datalist\_push\_back(Functions, code\_token); // Кидаем новую операцию в стек**

**}**

**} else {**

**return 0;**

**}**

**}**

**//prev\_token = code\_token;**

**//printf("%s\n", code\_token);**

**code\_token = strtok\_r(NULL, " ", &saveptr);**

**}**

**char \*tmp = datalist\_pop\_back(Operands);**

**//printf("che: %s\n", tmp);**

**if (last\_tmp\_var\_in\_acc != \*tmp) {**

**int var\_cell = get\_var\_address(tmp);**

**if (isalpha(\*tmp)) {**

**if (isupper(\*tmp)) {**

**var\_cell = get\_var\_address(tmp);**

**} else {**

**var\_cell = get\_tmp\_var\_address(\*tmp);**

**//printf("%d %s\n", var\_cell, B);**

**}**

**} else {**

**var\_cell = get\_num\_var\_address(atoi(tmp));**

**}**

**AsmCommandPush(list, COMMAND\_LOAD, var\_cell);**

**}**

**/\*for (int i = 0; i < strlen(token); i++) {**

**if (isblank(token[i])) {**

**continue;**

**} else if (isalpha(token[i])) {**

**if (!isupper(token[i])) {**

**return 0;**

**}**

**} else if (isdigit(token[i])) {**

**//int\_vector\_push\_back(IntVector \*v, int item)**

**} else {**

**}**

**}\*/**

**free(expression\_addres);**

**return list;**

**}**

**// Get tokens separated by delimiters strings**

**char \*get\_token(char \*\*str\_p, char \*delimiters[], int size)**

**{**

**char \*str = \*str\_p;**

**char buff[256] = {0};**

**int buff\_index = 0;**

**int start\_token = 0;**

**for (int i = 0; i <= strlen(str) + 1; ++i) {**

**if (str[i] == 0 && i == 0) {**

**return 0;**

**}**

**if (str[i] == ' ' || str[i] == 0) {**

**int flag = 0;**

**for (int j = 0; j < size; j++) {**

**if (!strcmp(buff, delimiters[j])) {**

**flag = 1;**

**}**

**}**

**if (flag) {**

**if (start\_token != 0) {**

**str[start\_token] = 0;**

**\*str\_p = str + start\_token + 1;**

**return str;**

**}**

**str[i] = 0;**

**\*str\_p = str + i + 1;**

**return str;**

**} else {**

**if (str[i] == 0) {**

**\*str\_p = str + i;**

**return str;**

**}**

**}**

**start\_token = i;**

**buff[0] = 0;**

**buff\_index = 0;**

**continue;**

**}**

**buff[buff\_index] = str[i];**

**buff[buff\_index + 1] = 0;**

**++buff\_index;**

**//printf("%s\n", buff);**

**}**

**return 0;**

**}**

**int translate\_input(LineStruct \*line\_struct)**

**{**

**int error\_flag = 0;**

**int memory\_address = get\_var\_address(line\_struct->basic\_operand);**

**if (memory\_address < 0) {**

**char buff[200];**

**int num\_spaces = calcspaces(line\_struct->basic\_str, line\_struct->basic\_operand - line\_struct->basic\_str);**

**sprintf(buff, "var name \"%s\" unacceptable\n%s\n\E[32m\E[1m%\*c\E[0m", line\_struct->basic\_operand, line\_struct->basic\_str, num\_spaces + 1, '^');**

**print\_error("error", buff);**

**error\_flag = 1;**

**}**

**printf("address:%d\n", memory\_address);**

**line\_struct->translated\_line\_num = asm\_last\_memory\_cell;**

**AsmCommandPush(line\_struct->command\_list, COMMAND\_READ, memory\_address);**

**return error\_flag;**

**}**

**int translate\_output(LineStruct \*line\_struct)**

**{**

**int error\_flag = 0;**

**int memory\_address = get\_var\_address(line\_struct->basic\_operand);**

**if (memory\_address < 0) {**

**char buff[200];**

**int num\_spaces = calcspaces(line\_struct->basic\_str, line\_struct->basic\_operand - line\_struct->basic\_str);**

**sprintf(buff, "var name \"%s\" unacceptable\n%s\n\E[32m\E[1m%\*c\E[0m", line\_struct->basic\_operand, line\_struct->basic\_str, num\_spaces + 1, '^');**

**print\_error("error", buff);**

**error\_flag = 1;**

**}**

**printf("address:%d\n", memory\_address);**

**line\_struct->translated\_line\_num = asm\_last\_memory\_cell;**

**AsmCommandPush(line\_struct->command\_list, COMMAND\_WRITE, memory\_address);**

**return error\_flag;**

**}**

**int translate\_let(LineStruct \*line\_struct)**

**{**

**int error\_flag = 0;**

**char \*token1;**

**char \*token2;**

**char \*token3;**

**char \*basic\_str = calloc(strlen(line\_struct->basic\_operand) + 1, sizeof(char));**

**strcpy(basic\_str, line\_struct->basic\_operand);**

**char \*basic\_str\_parse = basic\_str;**

**char \*str\_delim[1] = {"="};**

**int num\_token = 0;**

**char \*code\_token = get\_token(&basic\_str\_parse, str\_delim, 1);**

**while (code\_token) {**

**if (num\_token == 0) {**

**token1 = code\_token;**

**} else if (num\_token == 1) {**

**token2 = code\_token;**

**} else if (num\_token == 2) {**

**token3 = code\_token;**

**} else {**

**error\_flag = 1;**

**break;**

**}**

**code\_token = get\_token(&basic\_str\_parse, str\_delim, 1);**

**num\_token++;**

**}**

**printf("%s, %s, %s\n", token1, token2, token3);**

**int memory\_address = get\_var\_address(token1);**

**line\_struct->translated\_line\_num = asm\_last\_memory\_cell;**

**AsmCommandList \*list = translate\_expression(token3);**

**AsmCommandCat(line\_struct->command\_list, list);**

**AsmCommandPush(line\_struct->command\_list, COMMAND\_STORE, memory\_address);**

**AsmCommandPrint(line\_struct->command\_list);**

**//int memory\_address = get\_var\_address(line\_struct->basic\_operand);**

**/\* if (memory\_address < 0) {**

**char buff[200];**

**int num\_spaces = calcspaces(line\_struct->basic\_str\_full, line\_struct->basic\_operand - line\_struct->basic\_str);**

**sprintf(buff, "var name \"%s\" unacceptable\n%s\n\E[32m\E[1m%\*c\E[0m", line\_struct->basic\_operand, line\_struct->basic\_str\_full, num\_spaces + 1, '^');**

**print\_error("error", buff);**

**error\_flag = 1;**

**}**

**printf("address:%d\n", memory\_address);**

**char buff[3];**

**sprintf(buff, "%d", memory\_address);**

**line\_struct->translated\_str = calloc(strlen("WRITE XX") + 1, sizeof(char));**

**strcpy(line\_struct->translated\_str, "WRITE ");**

**strcat(line\_struct->translated\_str, buff);**

**line\_struct->translated\_line\_num = asm\_last\_memory\_cell;**

**asm\_last\_memory\_cell = asm\_last\_memory\_cell + 1;**

**printf("%s\n", line\_struct->translated\_str); \*/**

**free(basic\_str);**

**return error\_flag;**

**}**

**int translate\_end(LineStruct \*line\_struct)**

**{**

**int error\_flag = 0;**

**line\_struct->translated\_line\_num = asm\_last\_memory\_cell;**

**AsmCommandPush(line\_struct->command\_list, COMMAND\_HALT, 0);**

**return error\_flag;**

**}**

**DataList \*goto\_list = NULL;**

**void add\_undef\_goto(LineStruct \*line\_struct)**

**{**

**if (!goto\_list) {**

**goto\_list = datalist\_new();**

**}**

**datalist\_push\_back(goto\_list, line\_struct);**

**}**

**void process\_undef\_goto()**

**{**

**while (1) {**

**if (datalist\_size(goto\_list) > 0) {**

**LineStruct \*next = datalist\_pop\_back(goto\_list);**

**DataList \*node = line\_struct\_list;**

**if (!node) {**

**break;**

**}**

**LineStruct \*node\_val = node->value;**

**while (node\_val) {**

**if (node\_val->basic\_line\_num == atoi(next->basic\_operand)) {**

**//next->command\_list->current.num = next->translated\_line\_num;**

**next->command\_list->current.operand = node\_val->translated\_line\_num;**

**//printf("!%s %d!\n", node\_val->basic\_str, node\_val->translated\_line\_num);**

**break;**

**}**

**node = node->next;**

**node\_val = node->value;**

**}**

**} else {**

**break;**

**}**

**}**

**}**

**int translate\_goto(LineStruct \*line\_struct)**

**{**

**add\_undef\_goto(line\_struct);**

**printf("addgoto\n");**

**AsmCommandPush(line\_struct->command\_list, COMMAND\_JUMP, 0);**

**return 0;**

**}**

**int translate\_if(LineStruct \*line\_struct)**

**{**

**int error\_flag = 0;**

**line\_struct->translated\_line\_num = asm\_last\_memory\_cell;**

**AsmCommandList \*list = translate\_expression(line\_struct->basic\_operand);**

**printf("%s\n", line\_struct->basic\_operand);**

**AsmCommandCat(line\_struct->command\_list, list);**

**// IF 2 < 5 GOTO 20**

**// - 2 + 5 = 3**

**// 5 - 2 = 3**

**// 3 JNEG 5**

**// 4 GOTO 20**

**// 5 //any**

**AsmCommandList \*jmp = 0;**

**AsmCommandList \*jmp2 = 0;**

**if (last\_operation == OPERATION\_LESS || last\_operation == OPERATION\_MORE) {**

**jmp = AsmCommandPush(line\_struct->command\_list, COMMAND\_JNEG, 0);**

**jmp2 = AsmCommandPush(line\_struct->command\_list, COMMAND\_JZ, 0);**

**} else if (last\_operation == OPERATION\_EQUAL) {**

**jmp = AsmCommandPush(line\_struct->command\_list, COMMAND\_JNEG, 0);**

**AsmCommandPush(line\_struct->command\_list, COMMAND\_MUL, get\_num\_var\_address(-1));**

**jmp2 = AsmCommandPush(line\_struct->command\_list, COMMAND\_JNEG, 0);**

**}**

**LineStruct \*tmp\_line = calloc(1, sizeof(LineStruct));**

**tmp\_line->basic\_str = line\_struct->basic\_str;**

**tmp\_line->basic\_command = line\_struct->basic\_command2;**

**tmp\_line->basic\_operand = line\_struct->basic\_operand2;**

**tmp\_line->basic\_line\_num = line\_struct->basic\_line\_num;**

**tmp\_line->command\_list = calloc(1, sizeof(AsmCommandList));**

**if (!strcmp(tmp\_line->basic\_command, "REM")) {**

**//line\_struct->translated\_str = calloc(1, sizeof(char));**

**} else if (!strcmp(tmp\_line->basic\_command, "INPUT")) {**

**if (translate\_input(tmp\_line)) {**

**error\_flag = 1;**

**}**

**} else if (!strcmp(tmp\_line->basic\_command, "OUTPUT")) {**

**if (translate\_output(tmp\_line)) {**

**error\_flag = 1;**

**}**

**} else if (!strcmp(tmp\_line->basic\_command, "GOTO")) {**

**if (translate\_goto(tmp\_line)) {**

**error\_flag = 1;**

**}**

**} else if (!strcmp(tmp\_line->basic\_command, "IF")) {**

**if (translate\_if(tmp\_line)) {**

**error\_flag = 1;**

**}**

**} else if (!strcmp(tmp\_line->basic\_command, "LET")) {**

**if (translate\_let(tmp\_line)) {**

**error\_flag = 1;**

**}**

**} else if (!strcmp(tmp\_line->basic\_command, "END")) {**

**if (translate\_end(tmp\_line)) {**

**error\_flag = 1;**

**}**

**}**

**//printf("che\n");**

**if (jmp) {**

**AsmCommandCat(line\_struct->command\_list, tmp\_line->command\_list);**

**jmp->current.operand = asm\_last\_memory\_cell;**

**printf("%d\n", asm\_last\_memory\_cell);**

**}**

**if (jmp2) {**

**//AsmCommandCat(line\_struct->command\_list, tmp\_line->command\_list);**

**jmp2->current.operand = asm\_last\_memory\_cell;**

**printf("%d\n", asm\_last\_memory\_cell);**

**}**

**AsmCommandPrint(line\_struct->command\_list);**

**return error\_flag;**

**}**

**int translate\_basic(LineStruct \*line\_struct)**

**{**

**if (!line\_struct->basic\_str) {**

**print\_error("internal error", "couldnt find basic\_str");**

**return -1;**

**}**

**int error\_flag = 0;**

**int num\_token = 0;**

**char \*basic\_str = calloc(strlen(line\_struct->basic\_str) + 1, sizeof(char));**

**strcpy(basic\_str, line\_struct->basic\_str);**

**char \*basic\_str\_parse = basic\_str;**

**char \*code\_token = get\_token(&basic\_str\_parse, line\_delimiters, line\_delim\_size);**

**while (code\_token) {**

**int num\_spaces = calcspaces(line\_struct->basic\_str, code\_token - basic\_str);**

**if (num\_token == 0) {**

**int number;**

**if (!tonumber(code\_token, &number) || (number - basic\_last\_num\_line <= 0)) {**

**char buff[200];**

**sprintf(buff, "line number \"%s\" is wrong\n%s\n\E[32m\E[1m^\E[0m", code\_token, line\_struct->basic\_str);**

**print\_error("error", buff);**

**error\_flag = 1;**

**}**

**basic\_last\_num\_line = number;**

**line\_struct->basic\_line\_num = number;**

**} else if (num\_token == 1) {**

**if (validate\_command(code\_token) == -1) {**

**char buff[200];**

**sprintf(buff, "unknown command \"%s\" line %d\n%s\n\E[32m\E[1m%\*c\E[0m", code\_token, line\_struct->basic\_line\_num, line\_struct->basic\_str, num\_spaces + 1, '^');**

**print\_error("error", buff);**

**error\_flag = 1;**

**}**

**line\_struct->basic\_command = code\_token;**

**} else if (num\_token == 2) {**

**if (validate\_operand(code\_token) == -1) {**

**char buff[200];**

**sprintf(buff, "invalid operand \"%s\" not exist\n%s\n\E[32m\E[1m%\*c\E[0m", code\_token, line\_struct->basic\_str, num\_spaces + 1, '^');**

**print\_error("error", buff);**

**error\_flag = 1;**

**}**

**line\_struct->basic\_operand = code\_token;**

**} else if (num\_token == 3) {**

**if (validate\_command(code\_token) == -1) {**

**char buff[200];**

**sprintf(buff, "unknown command \"%s\" line %d\n%s\n\E[32m\E[1m%\*c\E[0m", code\_token, line\_struct->basic\_line\_num, line\_struct->basic\_str, num\_spaces + 1, '^');**

**print\_error("error", buff);**

**error\_flag = 1;**

**}**

**line\_struct->basic\_command2 = code\_token;**

**} else if (num\_token == 4) {**

**if (validate\_operand(code\_token) == -1) {**

**char buff[200];**

**sprintf(buff, "invalid operand \"%s\" not exist\n%s\n\E[32m\E[1m%\*c\E[0m", code\_token, line\_struct->basic\_str, num\_spaces + 1, '^');**

**print\_error("error", buff);**

**error\_flag = 1;**

**}**

**line\_struct->basic\_operand2 = code\_token;**

**} else {**

**error\_flag = 1;**

**break;**

**}**

**code\_token = get\_token(&basic\_str\_parse, line\_delimiters, line\_delim\_size);**

**num\_token++;**

**}**

**if (num\_token == 0) {**

**return -1;**

**}**

**if (error\_flag) {**

**return -1;**

**}**

**printf("1)%d 2)%s 3)%s", line\_struct->basic\_line\_num, line\_struct->basic\_command, line\_struct->basic\_operand);**

**if (line\_struct->basic\_command2) {**

**printf(" 4)%s 5)%s", line\_struct->basic\_command2, line\_struct->basic\_operand2);**

**}**

**printf("\n");**

**error\_flag = 0;**

**if (!strcmp(line\_struct->basic\_command, "REM")) {**

**//line\_struct->translated\_str = calloc(1, sizeof(char));**

**} else if (!strcmp(line\_struct->basic\_command, "INPUT")) {**

**if (translate\_input(line\_struct)) {**

**error\_flag = 1;**

**}**

**} else if (!strcmp(line\_struct->basic\_command, "OUTPUT")) {**

**if (translate\_output(line\_struct)) {**

**error\_flag = 1;**

**}**

**} else if (!strcmp(line\_struct->basic\_command, "GOTO")) {**

**if (translate\_goto(line\_struct)) {**

**error\_flag = 1;**

**}**

**} else if (!strcmp(line\_struct->basic\_command, "IF")) {**

**if (translate\_if(line\_struct)) {**

**error\_flag = 1;**

**}**

**} else if (!strcmp(line\_struct->basic\_command, "LET")) {**

**if (translate\_let(line\_struct)) {**

**error\_flag = 1;**

**}**

**} else if (!strcmp(line\_struct->basic\_command, "END")) {**

**if (translate\_end(line\_struct)) {**

**error\_flag = 1;**

**}**

**}**

**return 0;**

**}**

**int tonumber(char \*s, int \*number)**

**{**

**if (s == NULL || \*s == '\0' || isspace(\*s)) {**

**return 0;**

**}**

**char \*p;**

**\*number = (int)strtod(s, &p);**

**return \*p == '\0';**

**}**

**//count symbols in string**

**int calcspaces(char \*code\_token, int num)**

**{**

**if (num < 0 || !code\_token) {**

**return -1;**

**}**

**int num\_spaces = 0;**

**while (num > 0) {**

**if (\*code\_token == '\t') {**

**num\_spaces += 8 - (num\_spaces % 8);**

**} else {**

**num\_spaces++;**

**}**

**code\_token++;**

**num--;**

**}**

**return num\_spaces;**

**}**

**int validate\_operand(char \*cstr)**

**{**

**return 0;**

**}**

**//REM, INPUT, OUTPUT, GOTO, IF, LET, END.**

**int validate\_command(char \*cstr)**

**{**

**if (!strcmp(cstr, "REM")) {**

**} else if (!strcmp(cstr, "INPUT")) {**

**} else if (!strcmp(cstr, "OUTPUT")) {**

**} else if (!strcmp(cstr, "GOTO")) {**

**} else if (!strcmp(cstr, "IF")) {**

**} else if (!strcmp(cstr, "LET")) {**

**} else if (!strcmp(cstr, "END")) {**

**} else {**

**return -1;**

**}**

**return 0;**

**}**

**typedef struct VarStructChar {**

**int asm\_address;**

**char name;**

**struct VarStructChar \*next;**

**} VarStructChar;**

**typedef struct VarStructNumber {**

**int init;**

**int asm\_address;**

**int value;**

**struct VarStructNumber \*next;**

**} VarStructNumber;**

**VarStructChar \*var\_struct\_head = 0;**

**VarStructChar \*tmp\_var\_struct\_head = 0;**

**VarStructNumber \*num\_var\_struct\_head = 0;**

**int free\_asm\_address = 99;**

**int get\_var\_address(char \*cstr)**

**{**

**if (!cstr) {**

**return -1;**

**}**

**if (!(\*cstr >= 'A' && \*cstr <= 'Z')) {**

**return -1;**

**}**

**if (!var\_struct\_head) {**

**var\_struct\_head = calloc(1, sizeof(VarStructChar));**

**var\_struct\_head->name = \*cstr;**

**var\_struct\_head->asm\_address = free\_asm\_address--;**

**return var\_struct\_head->asm\_address;**

**}**

**VarStructChar \*last\_free = 0;**

**VarStructChar \*free = var\_struct\_head;**

**while (free) {**

**if (free->name == \*cstr) {**

**return free->asm\_address;**

**}**

**last\_free = free;**

**free = free->next;**

**}**

**last\_free->next = calloc(1, sizeof(VarStructChar));**

**last\_free->next->name = \*cstr;**

**last\_free->next->asm\_address = free\_asm\_address--;**

**return last\_free->next->asm\_address;**

**}**

**int get\_num\_var\_address(int val)**

**{**

**if (!num\_var\_struct\_head) {**

**num\_var\_struct\_head = calloc(1, sizeof(VarStructNumber));**

**num\_var\_struct\_head->value = val;**

**num\_var\_struct\_head->init = 1;**

**num\_var\_struct\_head->asm\_address = free\_asm\_address--;**

**return num\_var\_struct\_head->asm\_address;**

**}**

**VarStructNumber \*pre\_last = 0;**

**VarStructNumber \*free = num\_var\_struct\_head;**

**while (free) {**

**if (free->init && free->value == val) {**

**return free->asm\_address;**

**}**

**pre\_last = free;**

**free = free->next;**

**}**

**pre\_last->next = calloc(1, sizeof(VarStructNumber));**

**pre\_last->next->init = 1;**

**pre\_last->next->value = val;**

**pre\_last->next->asm\_address = free\_asm\_address--;**

**return pre\_last->next->asm\_address;**

**}**

**int get\_tmp\_var\_address(char cstr)**

**{**

**if (!cstr) {**

**return -1;**

**}**

**if (!(cstr >= 'a' && cstr <= 'z')) {**

**return -1;**

**}**

**if (!tmp\_var\_struct\_head) {**

**tmp\_var\_struct\_head = calloc(1, sizeof(VarStructChar));**

**tmp\_var\_struct\_head->name = cstr;**

**tmp\_var\_struct\_head->asm\_address = free\_asm\_address--;**

**return tmp\_var\_struct\_head->asm\_address;**

**}**

**VarStructChar \*last\_free = 0;**

**VarStructChar \*free = tmp\_var\_struct\_head;**

**while (free) {**

**if (free->name == cstr) {**

**return free->asm\_address;**

**}**

**last\_free = free;**

**free = free->next;**

**}**

**last\_free->next = calloc(1, sizeof(VarStructChar));**

**last\_free->next->name = cstr;**

**last\_free->next->asm\_address = free\_asm\_address--;**

**return last\_free->next->asm\_address;**

**}**

**int lexer(char \*infile, char \*outfile)**

**{**

**int indesc = open(infile, O\_RDONLY);**

**if (indesc == -1) {**

**print\_error("error", "cant open infile");**

**}**

**off\_t size\_file = lseek(indesc, 0, SEEK\_END);**

**if (size\_file == -1L) {**

**print\_error("error", "cant read infile size");**

**return -1;**

**}**

**if (lseek(indesc, 0, SEEK\_SET) == -1L) {**

**print\_error("error", "cant read infile size");**

**return -1;**

**}**

**//printf("find %ld bytes\n", size\_file);**

**char \*code\_buff = calloc(size\_file, 1);**

**if (size\_file != read(indesc, code\_buff, size\_file)) {**

**print\_error("error", "cant read infile fully");**

**return -1;**

**}**

**if (close(indesc) == -1) {**

**print\_error("error", "cant close infile");**

**}**

**for (int i = 0; i < strlen(code\_buff); ++i) {**

**if (code\_buff[i] == '\r') {**

**code\_buff[i] = ' ';**

**}**

**}**

**int num\_line = 0;**

**int error\_flag = 0;**

**char \*saveptr1;**

**char \*code\_line = strtok\_r(code\_buff, "\n", &saveptr1);**

**line\_struct\_list = datalist\_new();**

**while (code\_line) {**

**LineStruct \*line\_struct = calloc(1, sizeof(LineStruct));**

**line\_struct->basic\_str = calloc(strlen(code\_line) + 1, sizeof(char));**

**strcpy(line\_struct->basic\_str, code\_line);**

**line\_struct->basic\_line\_num = num\_line;**

**line\_struct->command\_list = calloc(1, sizeof(AsmCommandList));**

**datalist\_push\_back(line\_struct\_list, line\_struct);**

**// analis and fill struct**

**if (translate\_basic(line\_struct)) {**

**error\_flag = 1;**

**}**

**code\_line = strtok\_r(0, "\n", &saveptr1);**

**num\_line++;**

**}**

**if (error\_flag) {**

**printf("error\n");**

**return -1;**

**}**

**process\_undef\_goto();**

**AsmCommandList \*all\_command = calloc(1, sizeof(AsmCommandList));**

**DataList \*node = line\_struct\_list;**

**LineStruct \*node\_val = node->value;**

**while (node\_val) {**

**AsmCommandCat(all\_command, node\_val->command\_list);**

**node = node->next;**

**if (!node) {**

**break;**

**}**

**node\_val = node->value;**

**}**

**VarStructNumber \*num\_node = num\_var\_struct\_head;**

**while (num\_node) {**

**if (num\_node->init) {**

**AsmCommandPushEx(all\_command, COMMAND\_EQUAL, num\_node->value, num\_node->asm\_address);**

**}**

**num\_node = num\_node->next;**

**}**

**int outdesc = open(outfile, O\_WRONLY | O\_TRUNC | O\_CREAT, -1);**

**if (outdesc == -1) {**

**print\_error("error", "cant open outfile");**

**}**

**AsmCommandPrintEx(all\_command, outdesc);**

**datalist\_free(line\_struct\_list);**

**//pairs line\_num to fill memory**

**/\* if (write(outdesc, memory\_buff, 400) < 0) {**

**print\_error("error", "cant write in outfine");**

**return -1;**

**} \*/**

**if (close(outdesc) == -1) {**

**print\_error("error", "cant close outfile");**

**}**

**return 0;**

**}**