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

Кафедра ВС

**КУРСОВАЯ РАБОТА**

По дисциплине «Архитектура ЭВМ»

Вариант 6

Выполнил:

студент гр. ИВ-621

Дьяченко Д.В.

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

**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.

·

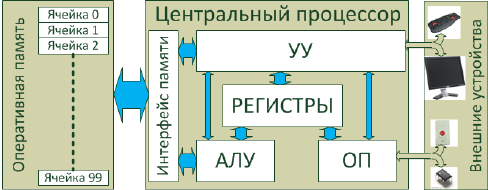


Рисунок 1 – Архитектура вычислительной машины Simple Computer

**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

Считываем значение ячейки из памяти

Декодируем полученное значение на команду и операнд

Команда корректна?

Нет

Да

Выполнение команды управления или вызов функции ALU для арифметической операции

**2.Simple Assembler**

Файл с кодом на **Assembler,**

**вых. файл**

Пока не конец файла считываем по строчке

Если строка корректна, кодируем её

Сохраняем полученные команды в выходной файл

**3.Simple Basic**

Файл с кодом на **Basic,**

**вых. файл**

Пока не конец файла считать строчку

Создать пул команд.

Декодировать каждую строчку и проверять ее на корректность. Если корректна добавить в пул, инчае ошибка.

Если команда не GOTO или IF с GOTO с операндом больше номера строки данной команды, то выполнить транслировать с ассемблер сразу, иначе пометить для обработки в конце работы цикла

**Программная реализация**

На сайте https://github.com/hegemonies/mySimpleComputer

**memory.h**

|  |
| --- |
| **#ifndef MEMORY\_H**  **#define MEMORY\_H**  **#include "terminal.h"**  **#define MEMORY\_SIZE 100**  **#define CELL\_SIZE sizeof(int)**  **//FLAGS**  **#define OD 0b00000001 // переполнение при выполнении операции**  **// overfulfillment during operation**  **#define DE 0b00000010 // ошибка деления на 0**  **// division error by 0**  **#define EG 0b00000100 // ошибка выхода за границы памяти**  **// error of going beyond borders**  **#define CI 0b00001000 // игнорирование тактовых импульсов**  **// clock ignoring**  **#define IC 0b00010000 // неверная команда**  **// invalid command**  **int ptr\_str[MEMORY\_SIZE];**  **unsigned short int flags;**  **int accum;**  **int sc\_memoryInit();**  **int sc\_memorySet(int address, int value);**  **int sc\_memoryGet(int address, int \*value);**  **int sc\_memorySave(char \*filename);**  **int sc\_memoryLoad(char\* filename);**  **void sm\_printMemory(int x, int y);**  **int sc\_regInit();**  **int sc\_regSet(int reg, int value);**  **int sc\_regGet(int register, int \*value);**  **int sc\_commandEncode(int command, int operand, int \*value);**  **int sc\_commandDecode(int value, int \*command, int \*operand);**  **#endif** |

**memory.c**

|  |
| --- |
| **#include "memory.h"**  **int sc\_memoryInit()**  **{**  **for (int i = 0; i < MEMORY\_SIZE; i++) {**  **ptr\_str[i] = 0;**  **}**  **accum = 0;**    **return 0;**  **}**  **int sc\_memorySet(int address, int value)**  **{**  **if (address < 0 || address > 99) {**  **sc\_regSet(EG, 1);**  **return 1;**  **}**  **ptr\_str[address] = value;**    **return 0;**  **}**  **int sc\_memoryGet(int address, int \*value)**  **{**  **if (address < 0 || address > 99) {**  **sc\_regSet(EG, 1);**  **return 1;**  **}**    **\*value = ptr\_str[address];**    **return 0;**  **}**  **int sc\_memorySave(char \*filename)**  **{**  **FILE \*ptrFile = fopen(filename, "wb");**  **fwrite(ptr\_str, CELL\_SIZE, MEMORY\_SIZE, ptrFile);**  **fclose(ptrFile);**  **return 0;**  **}**  **int sc\_memoryLoad(char\* filename)**  **{**  **FILE \*ptrFile = fopen(filename, "rb");**    **fread(ptr\_str, CELL\_SIZE, MEMORY\_SIZE, ptrFile);**    **fclose(ptrFile);**  **return 0;**  **}**  **void sm\_printMemory(int x, int y)**  **{**  **for (int i = 0; i < 10; i++) {**  **mt\_gotoXY(y + i, x);**  **for (int j = 0; j < 10; j++) {**  **if (ptr\_str[i \* 10 + j] < 65536) {**  **int tmp = ptr\_str[i \* 10 + j];**  **if (tmp >= 0) {**  **printf("+%04X ", tmp);**  **} else {**  **printf("-%04X ", tmp \* -1);**  **}**  **}**  **}**  **}**  **}**  **int sc\_regInit()**  **{**  **flags = 0;**  **if (sc\_regSet(CI, 1)) {**  **return 1;**  **}**  **return 0;**  **}**  **int sc\_regSet(int reg, int value)**  **{**  **if (reg == OD || reg == DE || reg == EG || reg == CI || reg == IC) {**  **if (value == 0) {**  **flags = flags & (~(reg));**  **} if (value == 1) {**  **flags = flags | reg;**  **} else {**  **return 1;**  **}**  **} else {**  **return 1;**  **}**  **return 0;**  **}**  **int sc\_regGet(int reg, int \*value)**  **{**  **if (!value) {**  **return 1;**  **}**  **if (reg == OD) {**  **\*value = flags & 0x1;**  **} else if (reg == DE) {**  **\*value = (flags >> 1) & 0x1;**  **} else if (reg == EG) {**  **\*value = (flags >> 2) & 0x1;**  **} else if (reg == CI) {**  **\*value = (flags >> 3) & 0x1;**  **} else if (reg == IC) {**  **\*value = (flags >> 4) & 0x1;**  **} else {**  **return 1;**  **}**  **return 0;**  **}**  **int sc\_commandEncode(int command, int operand, int \*value)**  **{**  **if ((command > 0 && command < 10) ||**  **(command > 11 && command < 20) ||**  **(command > 21 && command < 30) ||**  **(command > 33 && command < 40) ||**  **(command > 44 && command < 51) ||**  **command > 79) {**  **sc\_regSet(IC, 1);**  **return 1;**  **}**  **if (operand < 0 || operand > 127) {**  **return 1;**  **}**  **\*value = \*value | (command << 7);**  **\*value = \*value | operand;**  **return 0;**  **}**  **int sc\_commandDecode(int value, int \*command, int \*operand)**  **{**  **if ((value >> 14) != 0) {**  **mt\_gotoXY(28, 1);**  **printf("che 1\n");**  **return 1;**  **}**  **\*command = value >> 7;**  **if ((\*command > 0 && \*command < 10) ||**  **(\*command > 11 && \*command < 20) ||**  **(\*command > 21 && \*command < 30) ||**  **(\*command > 33 && \*command < 40) ||**  **(\*command > 44 && \*command < 51) ||**  **\*command > 79) {**  **sc\_regSet(IC, 1);**  **return 1;**  **}**  **\*operand = value & 0b1111111;**  **return 0;**  **}** |

**terminal.h**

|  |
| --- |
| **#ifndef TERMINAL\_H**  **#define TERMINAL\_H**  **#define \_GNU\_SOURCE**  **#define \_BSD\_SOURCE**  **#include <stdio.h>**  **#include <stdlib.h>**  **#include <unistd.h>**  **#include <fcntl.h>**  **#include <sys/ioctl.h>**  **enum colors {**  **black = 0,**  **red = 1,**  **green = 2,**  **yellow = 3,**  **blue = 4,**  **purple = 5,**  **cyan = 6,**  **white = 7**  **};**  **int mt\_clrscr();**  **int mt\_gotoXY(int y, int x);**  **int mt\_getscreensize(int \*rows, int \*cols);**  **int mt\_ssetfgcolor(enum colors color);**  **int mt\_ssetbgcolor(enum colors color);**  **int mt\_stopcolor();**  **#endif** |

**terminal.c**

|  |
| --- |
| **#include "terminal.h"**  **int mt\_clrscr()**  **{**  **printf("\E[H\E[J");**  **return 0;**  **}**  **int mt\_gotoXY(int y, int x)**  **{**  **printf("\E[%d;%dH", y, x);**  **return 0;**  **}**  **int mt\_getscreensize(int \*rows, int \*cols)**  **{**  **struct winsize ws;**  **if (ioctl(1, TIOCGWINSZ, &ws)) {**  **return 1;**  **} else {**  **\*rows = ws.ws\_row;**  **\*cols = ws.ws\_col;**  **}**  **return 0;**  **}**  **int mt\_ssetfgcolor(enum colors color)**  **{**  **printf("\E[3%dm", color);**  **return 0;**  **}**  **int mt\_ssetbgcolor(enum colors color)**  **{**  **printf("\E[4%dm", color);**  **return 0;**  **}**  **int mt\_stopcolor()**  **{**  **printf("\E[0m");**  **return 0;**  **}** |

**higchars.h**

|  |
| --- |
| **#ifndef BIGCHARS\_H**  **#define BIGCHARS\_H**  **#include "memory.h"**  **#define \_BSD\_SOURCE**  **//a f g i j k l m n o p q r s t u v w x y z { | } ~**  **//▒ ° ± ␋ ┘ ┐ ┌ └ ┼ ⎺ ⎻ ─ ⎼ ⎽ ├ ┤ ┴ ┬ │ ≤ ≥ π ≠ £ ·**  **#define bc\_cornerUpLeft 'l'**  **#define bc\_cornerDownLeft 'm'**  **#define bc\_cornerUpRight 'k'**  **#define bc\_cornerDownRight 'j'**  **#define bc\_horizontaleLine 'q'**  **#define bc\_verticalLine 'x'**  **#define bc\_shadedCell 'a'**  **#define bc\_Null(x) (x == 0) ? 1717992960 : 8283750**  **#define bc\_One(x) (x == 0) ? 471341056 : 3938328**  **#define bc\_Two(x) (x == 0) ? 538983424 : 3935292**  **#define bc\_Three(x) (x == 0) ? 2120252928 : 8282238**  **#define bc\_Four(x) (x == 0) ? 2120640000 : 6316158**  **#define bc\_Five(x) (x == 0) ? 2114092544 : 8273984**  **#define bc\_Six(x) (x == 0) ? 33701376 : 8274558**  **#define bc\_Seven(x) (x == 0) ? 811630080 : 396312**  **#define bc\_Eight(x) (x == 0) ? 2120646144 : 8283750**  **#define bc\_Nine(x) (x == 0) ? 2120646144 : 8282208**  **#define bc\_A(x) (x == 0) ? 1715214336 : 6710910**  **#define bc\_B(x) (x == 0) ? 1044528640 : 4080194**  **#define bc\_C(x) (x == 0) ? 37912064 : 8274434**  **#define bc\_D(x) (x == 0) ? 1111637504 : 4080194**  **#define bc\_E(x) (x == 0) ? 2114092544 : 8258050**  **#define bc\_F(x) (x == 0) ? 33701376 : 131646**  **#define bc\_Plus(x) (x == 0) ? 2115508224 : 1579134**  **#define bc\_Minus(x) (x == 0) ? 2113929216 : 126**  **int bc\_printA(char \*str);**  **int bc\_box(int x1, int y1, int x2, int y2);**  **int bc\_printbigchar(int a[2], int x, int y, enum colors, enum colors);**  **int bc\_setbigcharpos(int \*big, int x, int y, int value);**  **int bc\_getbigcharpos(int \*big, int x, int y, int \*value);**  **int bc\_bigcharwrite(int fd, int \*big, int count);**  **int bc\_bigcharread(int fd, int \*big, int need\_count, int \*count);**  **#endif** |

**bigchars.c**

|  |
| --- |
| **#include "bigchars.h"**  **int bc\_printA(char \*str)**  **{**  **if (!str) {**  **return 1;**  **}**  **printf("\E(0%s\E(B", str);**  **return 0;**  **}**  **int bc\_box(int x1, int y1, int x2, int y2)**  **{**  **if (x1 > x2 || y1 > y2) {**  **return 1;**  **}**  **if (x1 < 1 || y1 < 1) {**  **return 1;**  **}**  **mt\_gotoXY(y1, x1);**  **int x = x2 - x1 + 1;**  **int y = y2 - y1;**  **printf("\E(0");**  **for (int i = 0; i < y; i++) {**  **if (i == 0) {**  **printf("%c", bc\_cornerUpLeft);**  **for (int j = 1; j < x - 2; j++) {**  **printf("%c", bc\_horizontaleLine);**  **}**  **printf("%c", bc\_cornerUpRight);**  **printf("\n");**  **continue;**  **}**  **mt\_gotoXY(y1 + i, x1);**  **printf("%c", bc\_verticalLine);**  **for (int j = 1; j < x - 2; j++) {**  **printf("%c", ' ');**  **}**  **printf("%c", bc\_verticalLine);**  **printf("\n");**  **if (i == y - 1) {**  **mt\_gotoXY(y1 + i, x1);**  **printf("%c", bc\_cornerDownLeft);**  **for (int j = 1; j < x - 2; j++) {**  **printf("%c", bc\_horizontaleLine);**  **}**  **printf("%c", bc\_cornerDownRight);**  **printf("\n");**  **}**  **}**  **printf("\E(B");**  **return 0;**  **}**  **int bc\_printbigchar(int \*a, int x, int y, enum colors fg, enum colors bg)**  **{**  **if (!a ||**  **x < 1 || y < 1 ||**  **fg > 7 || fg < 0 ||**  **bg > 7 || bg < 0) {**  **return 1;**  **}**  **printf("\E(0");**  **mt\_ssetfgcolor(fg);**  **mt\_ssetbgcolor(bg);**  **for (int k = 0; k < 2; k++) {**  **for (int i = 0; i < 4; i++) {**  **mt\_gotoXY(y + i + k \* 4, x);**  **for (int j = 0; j < 8; j++) {**  **if ((a[k] >> (i \* 8 + j)) & 1) {**  **//printf("%c", bc\_shadedCell);**  **printf("f");**  **} else {**  **printf("%c", ' ');**  **}**  **}**  **}**  **}**    **printf("\E(B");**  **mt\_stopcolor();**  **return 0;**  **}**  **int bc\_setbigcharpos(int \*big, int x, int y, int value)**  **{**  **if (!big || x < 0 || x > 7 || y < 0 || y > 7 || value < 0 || value > 1) {**  **return 1;**  **}**  **y--;**  **x--;**  **if (y < 4) {**  **if (value) {**  **big[0] |= 1 << (y \* 8 + x);**  **} else {**  **big[0] = ~(1 << (y \* 8 + x));**  **}**  **} else {**  **if (value) {**  **big[1] |= 1 << (y \* 8 + x);**  **} else {**  **big[1] = ~(1 << (y \* 8 + x));**  **}**  **}**    **return 0;**  **}**  **int bc\_getbigcharpos(int \*big, int x, int y, int \*value)**  **{**  **if (!big || !value || x < 0 || x > 7 || y < 0 || y > 7) {**  **return 1;**  **}**  **x--;**  **y--;**  **if (y < 4) {**  **\*value = (big[0] >> (y \* 8 + x)) & 1;**  **} else {**  **\*value = (big[1] >> (y \* 8 + x)) & 1;**  **}**  **return 0;**  **}**  **int bc\_bigcharwrite(int fd, int \*big, int count)**  **{**  **if (!big || count < 0 || fd < 0) {**  **return 1;**  **}**  **for (; count > 0; count--) {**  **if (write(fd, &big[0], sizeof(int)) < 0) {**  **return 1;**  **}**  **if (write(fd, &big[1], sizeof(int)) < 0) {**  **return 1;**  **}**  **}**  **return 0;**  **}**  **int bc\_bigcharread(int fd, int \*big, int need\_count, int \*count)**  **{**  **if (!big || !count || fd < 0) {**  **return 1;**  **}**  **for (; need\_count > 0; need\_count--) {**  **if (read(fd, &big[0], sizeof(int)) < 0) {**  **\*count = 0;**  **return 1;**  **}**  **if (read(fd, &big[1], sizeof(int)) < 0) {**  **\*count = 0;**  **return 1;**  **}**  **}**  **return 0;**  **}** |

**myReadKey.h**

|  |
| --- |
| **#ifndef MYREADKEY\_H**  **#define MYREADKEY\_H**  **#include "bigchars.h"**  **#include <termios.h>**  **#include <string.h>**  **// F5 F6 UP DOWN LEFT RIGHT**  **// \E[15~ \E[17~ \E[A \E[B \E[D \E[C**  **enum keys {**  **F5 = 10,**  **F6,**  **UP,**  **DOWN,**  **LEFT,**  **RIGHT,**  **OTHER,**  **MINUS,**  **PLUS**  **};**  **struct termios tty;**  **struct termios savetty;**  **int rk\_readKey(enum keys \*key);**  **int rk\_mytermsave();**  **int rk\_mytermrestore();**  **int rk\_mytermregime(int regime, int vtime, int vmin, int echo, int sigint);**  **#endif** |

**myReadKey.c**

|  |
| --- |
| **#include "myReadKey.h"**  **int rk\_readKey(enum keys \*key)**  **{**  **rk\_mytermsave();**  **char buf[6] = { 0 };**  **rk\_mytermregime(1, 0, 1, 1, 1);**  **read(STDIN\_FILENO, buf, 6);**  **if (strcmp(buf, "\E[C") == 0) {**  **\*key = RIGHT;**  **} else if (strcmp(buf, "\E[D") == 0) {**  **\*key = LEFT;**  **} else if (strcmp(buf, "\E[A") == 0) {**  **\*key = UP;**  **} else if (strcmp(buf, "\E[B") == 0) {**  **\*key = DOWN;**  **} else if (strcmp(buf, "\E[15~") == 0) {**  **\*key = F5;**  **} else if (strcmp(buf, "\E[17~") == 0) {**  **\*key = F6;**  **} else if (strcmp(buf, "-") == 0) {**  **\*key = MINUS;**  **} else if (strcmp(buf, "+") == 0) {**  **\*key = PLUS;**  **} else if (buf[0] >= 48 && buf[0] < 58) {**  **\*key = buf[0] - 48;**  **} else if (buf[0] > 64 && buf[0] < 91) {**  **\*key = buf[0];**  **} else if (buf[0] > 96 && buf[0] < 123) {**  **\*key = buf[0];**  **} else if (buf[0] == 17) {**  **\*key = buf[0] + 28;**  **} else if (buf[0] == 18) {**  **\*key = buf[0] + 25;**  **} else {**  **\*key = OTHER;**  **}**    **rk\_mytermrestore();**  **return 0;**  **}**  **int rk\_mytermsave()**  **{**  **tcgetattr(STDIN\_FILENO, &tty);**  **savetty = tty;**  **return 0;**  **}**  **int rk\_mytermrestore()**  **{**  **if (tcsetattr(STDIN\_FILENO, TCSANOW, &savetty)) {**  **return 1;**  **}**  **return 0;**  **}**  **int rk\_mytermregime(int regime, int vtime, int vmin, int echo, int sigint)**  **{**  **if (regime == 1) { // некононичный**  **tty.c\_lflag &= ~ICANON;**  **if (echo == 1) {**  **tty.c\_lflag &= ~ECHO;**  **} else if (echo == 0) {**  **tty.c\_lflag |= ECHO;**  **} else {**  **write(STDERR\_FILENO, "Uncorrect argument ECHO in rk\_mytermregime\n", 43);**  **return -1;**  **}**  **if (sigint == 1) {**  **tty.c\_lflag &= ~ISIG;**  **} else if (sigint == 0) {**  **tty.c\_lflag |= ISIG;**  **} else {**  **write(STDERR\_FILENO, "Uncorrect argument SIGINT in rk\_mytermregime\n", 43);**  **return -1;**  **}**  **tty.c\_cc[VMIN] = vmin;**  **tty.c\_cc[VTIME] = vtime;**  **} else if (regime == 0) {// кононичный**  **tty.c\_lflag |= ICANON;**  **} else {**  **write(STDERR\_FILENO, "Uncorrect argument REGIME in rk\_mytermregime\n", 43);**  **return -1;**  **}**  **tcsetattr (0, TCSANOW, &tty);**  **return 0;**  **}** |

**cpu.h**

|  |
| --- |
| **#ifndef CPU\_H**  **#define CPU\_H**  **#include "myReadKey.h"**  **#define READ 10**  **#define WRITE 11**  **#define LOAD 20**  **#define STORE 21**  **#define ADD 30**  **#define SUB 31**  **#define DIVIDE 32**  **#define MUL 33**  **#define JUMP 40**  **#define JNEG 41**  **#define JZ 42**  **#define JB 44**  **#define JC 56**  **#define SET 78**  **#define HALT 43**  **int memory\_tmp[100];**  **int ALU(int command, int operand);**  **int CU();**  **#endif** |

**cpu.c**

|  |
| --- |
| **#include "cpu.h"**  **#include "helper.h"**  **int ALU(int command, int operand)**  **{**  **if (operand > 99) {**  **return 1;**  **}**  **switch (command) {**  **case ADD:**  **if ((accum + ptr\_str[operand]) >= 65535) {**  **sc\_regSet(OD, 1);**  **break;**  **}**  **accum += ptr\_str[operand];**  **break;**  **case SUB:**  **if ((accum - ptr\_str[operand]) < -65534) {**  **sc\_regSet(OD, 1);**  **break;**  **}**  **accum -= ptr\_str[operand];**  **break;**  **case DIVIDE:**  **if (ptr\_str[operand] == 0 || accum == 0) {**  **sc\_regSet(DE, 1);**  **break;**  **}**  **accum /= ptr\_str[operand];**  **break;**  **case MUL:**  **if ((accum \* ptr\_str[operand]) >= 65535) {**  **sc\_regSet(OD, 1);**  **break;**  **}**  **accum \*= ptr\_str[operand];**  **break;**  **default:**  **return 1;**  **}**  **return 0;**  **}**  **int CU()**  **{**  **int command = 0;**  **int operand = 0;**  **if (sc\_commandDecode(ptr\_str[instCount], &command, &operand)) {**  **sc\_regSet(IC, 1);**  **return 1;**  **}**  **int value = 0;**  **if (command > 33 || command < 30) {**  **switch (command) {**  **case READ:**  **mt\_gotoXY(26 + numStrForLogs, 1);**  **printf("-> ");**  **int tmp = 0;**  **scanf("%d", &tmp);**  **printf("\n");**  **if (tmp > 65535) {**  **sc\_regSet(OD, 1);**  **break;**  **}**  **ptr\_str[operand] = tmp;**  **incrementNumStrForLogs();**  **break;**  **case WRITE:**  **mt\_gotoXY(26 + numStrForLogs, 1);**  **printf("%d\n", ptr\_str[operand]);**  **incrementNumStrForLogs();**  **break;**  **case LOAD:**  **accum = ptr\_str[operand];**  **break;**  **case STORE:**  **ptr\_str[operand] = accum;**  **break;**  **case JUMP:**  **if (operand > 99 || operand < 0) {**  **sc\_regSet(EG, 1);**  **break;**  **}**  **instCount = operand;**  **instCount--;**  **break;**  **case JNEG:**  **if (accum < 0) {**  **instCount = operand;**  **instCount--;**  **}**  **break;**  **case JZ:**  **if (accum == 0) {**  **instCount = operand;**  **instCount--;**  **}**  **break;**  **case JC:**  **sc\_regGet(OD, &value);**  **if (value == 1) {**  **instCount = operand;**  **instCount--;**  **}**  **break;**  **case JB:**  **if (accum > 0) {**  **instCount = operand;**  **instCount--;**  **}**  **break;**  **case SET:**  **accum = operand;**  **break;**  **case HALT:**  **return 2;**  **break;**  **}**  **} else {**  **if (ALU(command, operand)) {**  **return 1;**  **}**  **}**    **instCount++;**  **return 0;**  **}** |

**helper.h**

|  |
| --- |
| **#ifndef HELPER\_H**  **#define HELPER\_H**  **// #include "myReadKey.h"**  **#include "cpu.h"**  **#include <ctype.h>**  **#include <string.h>**  **char \*\*banner;**  **int getBannerFromFile(char \*namefile, int \*count\_lines);**  **void printBanner(int count\_lines);**  **int changeSizeTerm();**  **void load();**  **int printMemory();**  **int printAccumalte();**  **int printInstCounter();**  **int printOperation();**  **int printFlags();**  **int printBoxBC();**  **int printHelpBox();**  **int interface(int size, int ban, int mem, int acc, int insCoun, int oper, int fl, int bc, int h);**  **enum way {**  **way\_UP, way\_DOWN, way\_LEFT, way\_RIGHT, way\_DEFAULT**  **};**  **int cell;**  **int instCount;**  **int numStrForLogs;**  **int intToHex(int number, char \*str);**  **void initNumberCell();**  **void printCell();**  **int printBigCharInBox();**  **void selectCellMemory(enum way w);**  **void selectCellMemoryByNumber(int num);**  **void initInstCounter();**  **int load\_prog\_from\_file(char \*path);**  **int save\_prog\_in\_file(char \*path);**  **int runtime();**  **void initNumStrForLogs();**  **void incrementNumStrForLogs();**  **int runtime\_OneStep();**  **int m\_strcmp(char \*s1, char \*s2);**  **int get\_command\_asm(char \*command);**  **#endif** |

**helper.c**

|  |
| --- |
| **#include "helper.h"**  **#include <stdint.h>**  **int getBannerFromFile(char \*namefile, int \*count\_lines)**  **{**  **FILE \*in = fopen(namefile, "r");**  **if (!in) {**  **return 1;**  **}**  **char \*buf = NULL;**  **size\_t len = 0;**  **\*count\_lines = 0;**  **while (getline(&buf, &len, in) != -1) {**  **(\*count\_lines)++;**  **}**  **fseek(in, 0, SEEK\_SET);**  **banner = calloc(\*count\_lines, sizeof(char\*));**  **for (int i = 0; getline(&buf, &len, in) != -1; i++) {**  **banner[i] = calloc(strlen(buf) + 2, sizeof(char));**  **strcpy(banner[i], buf);**  **}**  **fclose(in);**  **return 0;**  **}**  **void printBanner(int count\_lines)**  **{**  **enum colors fg = white;**  **enum colors bg = black;**  **printf("\E(0");**  **mt\_ssetfgcolor(fg);**  **mt\_ssetbgcolor(bg);**  **for (int i = 0; i < count\_lines; i++) {**  **printf("%s", banner[i]);**  **fflush(stdout);**  **}**  **printf("\E(B");**  **mt\_stopcolor();**  **}**  **int changeSizeTerm()**  **{**  **int size\_console\_x;**  **int size\_console\_y;**  **if (mt\_getscreensize(&size\_console\_y, &size\_console\_x) != 0) {**  **printf("Error\n");**  **return 1;**  **}**  **if (size\_console\_x < 85 || size\_console\_y < 46) {**  **printf("\033[8;45;84t");**  **}**  **mt\_clrscr();**  **mt\_gotoXY(1, 1);**  **return 0;**  **}**  **void load()**  **{**  **mt\_gotoXY(25, 1);**  **printf("loading");**  **for (int i = 0; i < 3; i++) {**  **sleep(1);**  **printf(".");**  **fflush(stdout);**  **}**  **printf("All good. Start\n");**  **sleep(1);**  **mt\_clrscr();**  **}**  **int printMemory()**  **{**  **if (bc\_box(1, 1, 63, 13) != 0) {**  **return 1;**  **}**  **mt\_gotoXY(1, 28);**  **printf("Memory");**  **sm\_printMemory(2, 2);**  **return 0;**  **}**  **int printAccumalte()**  **{**  **if (bc\_box(63, 1, 84, 4) != 0) {**  **return 1;**  **}**  **mt\_gotoXY(1, 68);**  **printf("Accumulator");**  **mt\_gotoXY(2, 70);**  **if (accum < 65536) {**  **int tmp = accum;**  **if (tmp >= 0) {**  **printf("+%04X", tmp);**  **} else {**  **printf("-%04X", tmp \* -1);**  **}**  **}**  **return 0;**  **}**  **int printInstCounter()**  **{**  **if (bc\_box(63, 4, 84, 7) != 0) {**  **return 1;**  **}**  **mt\_gotoXY(4, 64);**  **printf("InstructionCounter");**  **mt\_gotoXY(5, 70);**  **printf("+%04d", instCount);**  **return 0;**  **}**  **int printOperation()**  **{**  **if (bc\_box(63, 7, 84, 10) != 0) {**  **return 1;**  **}**  **mt\_gotoXY(7, 69);**  **printf("Operation");**  **int command = ptr\_str[cell] >> 7;**  **int operand = ptr\_str[cell] & 0b1111111;**  **mt\_gotoXY(8, 69);**  **printf("+ %02d : %02d\n", command, operand);**  **return 0;**  **}**  **int printFlags()**  **{**  **if (bc\_box(63, 10, 84, 13) != 0) {**  **return 1;**  **}**  **mt\_gotoXY(10, 71);**  **printf("Flags");**  **mt\_gotoXY(11, 64);**  **int \_OD;**  **sc\_regGet(OD, &\_OD);**  **int \_DE;**  **sc\_regGet(DE, &\_DE);**  **int \_EG;**  **sc\_regGet(EG, &\_EG);**  **int \_CI;**  **sc\_regGet(CI, &\_CI);**  **int \_IC;**  **sc\_regGet(IC, &\_IC);**  **printf("D-%d E-%d G-%d I-%d C-%d", \_OD, \_DE, \_EG, \_CI, \_IC);**  **return 0;**  **}**  **int printBoxBC()**  **{**  **if (bc\_box(1, 13, 63, 23) != 0) {**  **return 1;**  **}**  **int chr[2];**  **chr[0] = bc\_Plus(0);**  **chr[1] = bc\_Plus(1);**  **bc\_printbigchar(chr, 4, 14, 4, 7);**  **chr[0] = bc\_D(0);**  **chr[1] = bc\_D(1);**  **bc\_printbigchar(chr, 19, 14, 4, 7);**  **chr[0] = bc\_E(0);**  **chr[1] = bc\_E(1);**  **bc\_printbigchar(chr, 30, 14, 4, 7);**  **chr[0] = bc\_A(0);**  **chr[1] = bc\_A(1);**  **bc\_printbigchar(chr, 41, 14, 4, 7);**  **chr[0] = bc\_D(0);**  **chr[1] = bc\_D(1);**  **bc\_printbigchar(chr, 52, 14, 4, 7);**  **return 0;**  **}**  **int printHelpBox()**  **{**  **if (bc\_box(63, 13, 84, 23) != 0) {**  **return 1;**  **}**  **mt\_gotoXY(13, 64);**  **printf("Keys:");**  **mt\_gotoXY(14, 64);**  **printf("l - load");**  **mt\_gotoXY(15, 64);**  **printf("s - save");**  **mt\_gotoXY(16, 64);**  **printf("r - run");**  **mt\_gotoXY(17, 64);**  **printf("t - step");**  **mt\_gotoXY(18, 64);**  **printf("i - reset");**  **mt\_gotoXY(19, 64);**  **printf("F5 - accumulator");**  **mt\_gotoXY(20, 64);**  **printf("F6 - instrCounter");**  **return 0;**  **}**  **int interface(int size, int ban, int mem, int acc, int insCoun, int oper, int fl, int bc, int h)**  **{**  **rk\_mytermsave();**  **if (size) {**  **changeSizeTerm();**  **}**  **if (ban) {**  **int count\_lines = 0;**  **if (getBannerFromFile("banner.txt", &count\_lines)) {**  **return 1;**  **}**  **printBanner(count\_lines);**  **load();**  **}**  **if (mem) {**  **if (printMemory()) {**  **return 1;**  **}**  **}**    **if (acc) {**  **if (printAccumalte()) {**  **return 1;**  **}**  **}**    **if (insCoun) {**  **if (printInstCounter()) {**  **return 1;**  **}**  **}**    **if (oper) {**  **if (printOperation()) {**  **return 1;**  **}**  **}**    **if (fl) {**  **if (printFlags()) {**  **return 1;**  **}**  **}**    **if (bc) {**  **if (printBoxBC()) {**  **return 1;**  **}**  **}**    **if (h) {**  **if (printHelpBox()) {**  **return 1;**  **}**  **}**  **mt\_gotoXY(26, 1);**  **fflush(stdout);**  **return 0;**  **}**  **int intToHex(int number, char \*str)**  **{**  **if (!str || number >= 65535 || number < 0) {**  **return 1;**  **}**  **for (int i = 0; i < 5; i++) {**  **str[i] = 0;**  **}**  **int remainder;**  **int whole = number;**  **int i;**  **for (i = 0; whole >= 16; i++) {**  **remainder = whole % 16;**  **whole = whole / 16;**  **if (remainder == 10) {**  **str[i] = 'A';**  **} else if (remainder == 11) {**  **str[i] = 'B';**  **} else if (remainder == 12) {**  **str[i] = 'C';**  **} else if (remainder == 13) {**  **str[i] = 'D';**  **} else if (remainder == 14) {**  **str[i] = 'E';**  **} else if (remainder == 15) {**  **str[i] = 'F';**  **} else {**  **str[i] = remainder + 48;**  **}**  **}**  **if (whole != 0) {**  **if (whole == 10) {**  **str[i] = 'A';**  **} else if (whole == 11) {**  **str[i] = 'B';**  **} else if (whole == 12) {**  **str[i] = 'C';**  **} else if (whole == 13) {**  **str[i] = 'D';**  **} else if (whole == 14) {**  **str[i] = 'E';**  **} else if (whole == 15) {**  **str[i] = 'F';**  **} else {**  **str[i] = whole + 48;**  **}**  **}**    **return 0;**  **}**  **void initNumberCell()**  **{**  **cell = 0;**  **}**  **void printCell()**  **{**  **if (ptr\_str[cell] < 65536) {**  **int tmp = ptr\_str[cell];**  **if (tmp >= 0) {**  **printf("+%04X", tmp);**  **} else {**  **printf("-%04X", tmp \* -1);**  **}**  **}**  **fflush(stdout);**  **}**  **int printBigCharInBox()**  **{**  **int bigChars[5][2];**  **int tmp\_number = ptr\_str[cell];**  **if (tmp\_number >= 0) {**  **bigChars[0][0] = bc\_Plus(0);**  **bigChars[0][1] = bc\_Plus(1);**  **} else {**  **bigChars[0][0] = bc\_Minus(0);**  **bigChars[0][1] = bc\_Minus(1);**  **tmp\_number = -tmp\_number;**  **}**  **char buf[5];**  **if (intToHex(tmp\_number, buf)) {**  **return 1;**  **}**  **int j = 4;**  **for (int i = 0; i < 4; i++) {**  **if (buf[i] == '0') {**  **fflush(stdout);**  **bigChars[j][0] = bc\_Null(0);**  **bigChars[j][1] = bc\_Null(1);**  **} else if (buf[i] == '1') {**  **bigChars[j][0] = bc\_One(0);**  **bigChars[j][1] = bc\_One(1);**  **} else if (buf[i] == '2') {**  **bigChars[j][0] = bc\_Two(0);**  **bigChars[j][1] = bc\_Two(1);**  **} else if (buf[i] == '3') {**  **bigChars[j][0] = bc\_Three(0);**  **bigChars[j][1] = bc\_Three(1);**  **} else if (buf[i] == '4') {**  **bigChars[j][0] = bc\_Four(0);**  **bigChars[j][1] = bc\_Four(1);**  **} else if (buf[i] == '5') {**  **bigChars[j][0] = bc\_Five(0);**  **bigChars[j][1] = bc\_Five(1);**  **} else if (buf[i] == '6') {**  **bigChars[j][0] = bc\_Six(0);**  **bigChars[j][1] = bc\_Six(1);**  **} else if (buf[i] == '7') {**  **bigChars[j][0] = bc\_Seven(0);**  **bigChars[j][1] = bc\_Seven(1);**  **} else if (buf[i] == '8') {**  **bigChars[j][0] = bc\_Eight(0);**  **bigChars[j][1] = bc\_Eight(1);**  **} else if (buf[i] == '9') {**  **bigChars[j][0] = bc\_Nine(0);**  **bigChars[j][1] = bc\_Nine(1);**  **} else if (buf[i] == 'A') {**  **bigChars[j][0] = bc\_A(0);**  **bigChars[j][1] = bc\_A(1);**  **} else if (buf[i] == 'B') {**  **bigChars[j][0] = bc\_B(0);**  **bigChars[j][1] = bc\_B(1);**  **} else if (buf[i] == 'C') {**  **bigChars[j][0] = bc\_C(0);**  **bigChars[j][1] = bc\_C(1);**  **} else if (buf[i] == 'D') {**  **bigChars[j][0] = bc\_D(0);**  **bigChars[j][1] = bc\_D(1);**  **} else if (buf[i] == 'E') {**  **bigChars[j][0] = bc\_E(0);**  **bigChars[j][1] = bc\_E(1);**  **} else if (buf[i] == 'F') {**  **bigChars[j][0] = bc\_F(0);**  **bigChars[j][1] = bc\_F(1);**  **} else {**  **bigChars[j][0] = bc\_Null(0);**  **bigChars[j][1] = bc\_Null(1);**  **}**  **j--;**  **}**  **int x;**  **enum colors fg = red;**  **enum colors bg = white;**  **for (int i = 0; i < 5; i++) {**  **if (i == 0) {**  **x = 4;**  **} else {**  **x = 8;**  **}**  **bc\_printbigchar(bigChars[i], x + i \* 11, 14, fg, bg);**  **}**  **mt\_gotoXY(26, 1);**  **fflush(stdout);**  **return 0;**  **}**  **void selectCellMemory(enum way w)**  **{**  **enum colors color = red;**  **if (w == way\_RIGHT) {**  **mt\_gotoXY((cell / 10) + 2, (cell % 10) \* 6 + 2);**  **mt\_stopcolor();**  **printCell();**  **if (cell < 99) {**  **cell++;**  **}**  **color = red;**  **mt\_gotoXY((cell / 10) + 2, (cell % 10) \* 6 + 2);**  **mt\_ssetbgcolor(color);**    **printCell();**  **printBigCharInBox();**  **printOperation(); // TODO**  **}**  **if (w == way\_LEFT) {**  **mt\_gotoXY((cell / 10) + 2, (cell % 10) \* 6 + 2);**  **mt\_stopcolor();**    **printCell();**  **if (cell > 0) {**  **cell--;**  **}**  **color = red;**  **mt\_gotoXY((cell / 10) + 2, (cell % 10) \* 6 + 2);**  **mt\_ssetbgcolor(color);**    **printCell();**  **printBigCharInBox();**  **printOperation();**  **}**  **if (w == way\_UP) {**  **mt\_gotoXY((cell / 10) + 2, (cell % 10) \* 6 + 2);**  **mt\_stopcolor();**    **printCell();**  **if (cell > 9) {**  **cell -= 10;**  **}**  **color = red;**  **mt\_gotoXY((cell / 10) + 2, (cell % 10) \* 6 + 2);**  **mt\_ssetbgcolor(color);**    **printCell();**  **printBigCharInBox();**  **printOperation();**  **}**  **if (w == way\_DOWN) {**  **mt\_gotoXY((cell / 10) + 2, (cell % 10) \* 6 + 2);**  **mt\_stopcolor();**    **printCell();**  **if (cell < 90) {**  **cell += 10;**  **}**  **color = red;**  **mt\_gotoXY((cell / 10) + 2, (cell % 10) \* 6 + 2);**  **mt\_ssetbgcolor(color);**  **printCell();**  **printBigCharInBox();**  **printOperation();**  **}**  **if (w == way\_DEFAULT) {**  **printBigCharInBox();**  **color = red;**  **mt\_gotoXY((cell / 10) + 2, (cell % 10) \* 6 + 2);**  **mt\_ssetbgcolor(color);**  **printCell();**  **printOperation();**  **}**  **mt\_stopcolor();**  **}**  **void selectCellMemoryByNumber(int num)**  **{**  **if (num < 0 || num > 99) {**  **return;**  **}**  **cell = num;**  **mt\_gotoXY((cell / 10) + 2, (cell % 10) \* 6 + 2);**  **enum colors color = red;**  **mt\_ssetbgcolor(color);**  **printCell();**  **printBigCharInBox();**  **}**  **void initInstCounter()**  **{**  **instCount = 0;**  **}**  **int load\_prog\_from\_file(char \*path)**  **{**  **FILE \*in = fopen(path, "r");**  **if (!in) {**  **return 1;**  **}**  **for (int i = 0; i < 100; i++) {**  **fscanf(in, "%d", &ptr\_str[i]);**  **}**  **fclose(in);**  **return 0;**  **}**  **int save\_prog\_in\_file(char \*path)**  **{**  **FILE \*out = fopen(path, "w");**  **for (int i = 0; i < 100; i++) {**  **fprintf(out, "%d ", ptr\_str[i]);**  **}**  **fflush(out);**  **fclose(out);**  **return 0;**  **}**  **int runtime()**  **{**  **int statusIter = 0;**  **do {**  **statusIter = CU();**  **interface(0, 0, 1, 1, 1, 0, 1, 0, 0);**  **enum colors color = red;**  **mt\_ssetbgcolor(color);**  **mt\_gotoXY((instCount / 10) + 2, (instCount % 10) \* 6 + 2);**  **if (ptr\_str[instCount] < 65536) {**  **int tmp = ptr\_str[instCount];**  **if (tmp >= 0) {**  **printf("+%04X", tmp);**  **mt\_gotoXY(1000, 0);**  **} else {**  **printf("-%04X", tmp \* -1);**  **mt\_gotoXY(1000, 0);**  **}**  **}**  **fflush(stdout);**  **mt\_stopcolor();**  **sleep(1);**  **if (statusIter == 1) {**  **printf("Status Iteration = 1 ( Error )\n");**  **break;**  **}**  **} while (statusIter != 2);**  **instCount = 0;**  **mt\_gotoXY(26 + numStrForLogs, 1);**  **printf("End program\n");**  **incrementNumStrForLogs();**  **return 0;**  **}**  **void initNumStrForLogs()**  **{**  **numStrForLogs = 0;**  **}**  **void incrementNumStrForLogs()**  **{**  **numStrForLogs++;**  **if (numStrForLogs > 10) {**  **mt\_gotoXY(26, 1);**  **for (int i = 26; i < 36; i++) {**  **printf(" \n");**  **}**  **numStrForLogs = 0;**  **}**  **}**  **int runtime\_OneStep()**  **{**  **CU();**  **interface(0, 0, 1, 1, 1, 0, 1, 0, 0);**  **return 0;**  **}**  **int m\_strcmp(char \*s1, char \*s2)**  **{**  **int check = 0;**  **for (int i = 0; isalpha(s1[i]) && isalpha(s2[i]); i++) {**  **if (s1[i] == s2[i]) {**  **check++;**  **} else {**  **check = 0;**  **break;**  **}**  **}**  **return check;**  **}**  **int get\_command\_asm(char \*command)**  **{**  **if (m\_strcmp(command, "READ"))**  **return READ;**  **if (m\_strcmp(command, "WRITE"))**  **return WRITE;**  **if (m\_strcmp(command, "LOAD"))**  **return LOAD;**  **if (m\_strcmp(command, "STORE"))**  **return STORE;**  **if (m\_strcmp(command, "ADD"))**  **return ADD;**  **if (m\_strcmp(command, "SUB"))**  **return SUB;**  **if (m\_strcmp(command, "DIVIDE"))**  **return DIVIDE;**  **if (m\_strcmp(command, "MUL"))**  **return MUL;**  **if (m\_strcmp(command, "JUMP"))**  **return JUMP;**  **if (m\_strcmp(command, "JNEG"))**  **return JNEG;**  **if (m\_strcmp(command, "JZ"))**  **return JZ;**  **if (m\_strcmp(command, "JC"))**  **return JC;**  **if (m\_strcmp(command, "JB"))**  **return JB;**  **if (m\_strcmp(command, "SET"))**  **return SET;**  **if (m\_strcmp(command, "HALT"))**  **return HALT;**  **return 1;**  **}** |

**asm.h**

|  |
| --- |
| **#ifndef ASM\_H**  **#define ASM\_H**  **#include "helper.h"**  **#define enemy\_ 10**  **void help();**  **int asm\_string\_parser(char \*str, int \*num\_str, int \*command, int \*num\_cell, int \*i);**  **int asm\_translate(char \*path\_from, char \*path\_where);**  **#endif** |

**asm.c**

|  |
| --- |
| **#include "asm.h"**  **void help()**  **{**  **printf("Help:\n");**  **printf("bin/sat [filename from [\*.asm]] [filename where [\*.bin]]\n");**  **}**  **int asm\_string\_parser(char \*str, int \*num\_str, int \*command, int \*num\_cell, int \*i)**  **{**  **char \*command\_ = calloc(0, sizeof(char) \* 8);**  **\*num\_str = 0;**  **\*num\_cell = 0;**  **for (\*i = 0; str[\*i] != ' '; (\*i) += 1) {**  **if (isdigit(str[\*i])) {**  **if (\*i == 0) {**  **\*num\_str += ((int)str[\*i] - 48) \* 10;**  **} else {**  **\*num\_str += ((int)str[\*i] - 48);**  **}**  **} else {**  **printf("Error incorrect format number");**  **return 1;**  **}**  **}**  **if (\*i > 2) {**  **printf("Too many line numbers");**  **return 1;**  **} else if (\*i < 2) {**  **printf("Too small line numbers = %d", \*i);**  **return 1;**  **}**  **int isEqually = 0;**  **if (str[3] != '=') {**  **int k;**  **for (k = 0, \*i = 3; isalpha(str[\*i]); k++, (\*i) += 1) {**  **command\_[k] = str[\*i];**  **}**  **if (\*i > 8) {**  **printf("Too many line numbers");**  **return 1;**  **} else if (\*i < 4) {**  **printf("Too small line numbers");**  **return 1;**  **}**    **if ((\*command = get\_command\_asm(command\_)) == 1) {**  **printf("Incorrect command");**  **return 1;**  **}**  **} else {**  **isEqually = 1;**  **}**  **int isMinus = 0;**  **for (; !isdigit(str[\*i]); (\*i)++) {**  **if (str[\*i] == '-') {**  **isMinus = 1;**  **}**  **if (str[\*i] == '+') {**  **isMinus = 2;**  **}**  **}**  **if (isEqually && !isMinus) {**  **printf("Need + or -");**  **return 1;**  **}**  **int j;**  **int tmp\_num\_cell[6];**  **for (int k = 0; k < 6; k++)**  **tmp\_num\_cell[k] = enemy\_;**  **for (j = 0; str[\*i] != '\0' && str[\*i] != ' ' && str[\*i] != '\n'; (\*i)++, j++) {**  **if (isdigit(str[\*i])) {**  **tmp\_num\_cell[j] = (int)str[\*i] - 48;**  **} else {**  **if (j == 2 && \*num\_cell == 0) {**  **break;**  **} else {**  **printf("Error incorrect format number");**  **return 1;**  **}**  **}**  **}**  **int count;**  **for (count = 0; tmp\_num\_cell[count] != enemy\_; count++) { }**  **int tnc[count];**  **for (int k = 0, n = count - 1; k < count; k++, n--) {**  **tnc[n] = tmp\_num\_cell[k];**  **}**  **for (int k = 0; k < count; k++) {**  **if (k == 0) {**  **\*num\_cell += tnc[k];**  **} else {**  **\*num\_cell += tnc[k] \* (10 \* k);**  **}**  **}**  **if (j > 2 && !isEqually) {**  **printf("Too many line numbers");**  **return 1;**  **} else if (j < 1 && !isEqually) {**  **printf("Too small line numbers");**  **return 1;**  **}**  **if (isMinus == 2) {**  **// printf("che\n");**  **if ((\*num\_cell) < 65535) {**  **// printf("che1\n");**  **memory\_tmp[\*num\_str] = \*num\_cell;**  **} else {**  **printf("Number is so big\n");**  **return 1;**  **}**  **\*command = 0;**  **} else if (isMinus == 1) {**  **if ((\*num\_cell) < 65535) {**  **memory\_tmp[\*num\_str] = \*num\_cell \* (-1);**  **} else {**  **printf("Number is so big\n");**  **return 1;**  **}**  **\*command = 0;**  **}**  **// free(command\_); // TODO: why dont work?**  **return 0;**  **}**  **int asm\_translate(char \*path\_from, char \*path\_where)**  **{**  **FILE \*in = fopen(path\_from, "r");**  **if (!in) {**  **printf("No such file.");**  **return 1;**  **}**  **char \*buf = NULL;**  **size\_t len = 0;**  **int count\_lines = 0;**  **int num\_line;**  **int command;**  **int num\_cell;**  **while (getline(&buf, &len, in) != -1) {**  **int i = 0;**  **if (asm\_string\_parser(buf, &num\_line, &command, &num\_cell, &i)) {**  **fclose(in);**  **printf(" in %d line\n", count\_lines + 1);**  **printf("%s\n", buf);**  **for (; i != 0; i--) {**  **printf(" ");**  **}**  **mt\_ssetbgcolor(red);**  **printf("^");**  **mt\_stopcolor();**  **printf(" Error is here\n");**  **return 1;**  **}**  **count\_lines++;**  **if (sc\_commandEncode(command, num\_cell, &memory\_tmp[num\_line])) {**  **printf("%d : %d : %d\n", num\_line, command, num\_cell);**  **fclose(in);**  **printf("Error encode command");**  **printf(" in %d line\n", count\_lines);**  **return 1;**  **}**  **// printf("%d : %d : %d\n", num\_line, command, num\_cell);**  **}**  **fclose(in);**  **FILE \*out = fopen(path\_where, "w");**  **for (int i = 0; i < 100; i++) {**  **fprintf(out, "%d ", memory\_tmp[i]);**  **}**  **fclose(out);**  **return 0;**  **}** |

**asm\_main.c**

|  |
| --- |
| **#include "asm.h"**  **int main(int argc, char \*\*args)**  **{**  **if (argc == 3) {**  **if (asm\_translate(args[1], args[2])) {**  **printf("Please rewrite your code.\n");**  **return 1;**  **}**  **} else {**  **help();**  **return 1;**  **}**  **printf("Finish\n");**  **return 0;**  **}** |

**basic.h**

|  |
| --- |
| **#ifndef BASIC\_H**  **#define BASIC\_H**  **#include "asm.h"**  **#include "helper.h"**  **#include <math.h>**  **typedef struct**  **{**  **int orig\_num\_line;**  **int num\_line;**  **int command;**  **int tmp\_dig;**  **char \*str;**  **} unit\_command;**  **typedef struct var**  **{**  **int num\_cell;**  **char name;**  **struct var \*next;**  **} var;**  **#define REM 1**  **#define INPUT 2**  **#define OUTPUT 3**  **#define END 4**  **#define GOTO 5**  **#define GOTO\_B 6**  **#define IF 7**  **#define IF\_B 8**  **#define LET 9**  **#define additional\_operations 10**  **int basic\_string\_parser\_first(char \*str, int \*i, unit\_command \*unit\_commands, int \*add\_oper, char \*name\_var);**  **int basic\_translator(char \*path\_from, char \*path\_where, int \*i);**  **int get\_command\_basic(char \*str);**    **int amount\_lines;**  **var \*head\_stack\_of\_vars;**  **int add\_var(char name, int num\_cell);**  **var \*get\_var(char name);**  **int cell\_number\_for\_variables;**  **int get\_cellNumberForNewVariables();**  **int basic\_translator\_goto(char \*str, int \*dig, int \*i);**  **int get\_num\_line\_to\_ass\_from\_pull(unit\_command \*pull\_commands, int num);**  **int isCommandInPull(unit\_command \*pull\_commands, int num);**  **#define EQL 20**  **#define LARGER 21**  **#define LESS 22**  **int basic\_translator\_if(char \*buf, char \*oper\_a, char \*oper\_b, int \*operation, int \*i, int \*num\_cell\_for\_jump);**  **int get\_num\_line\_for\_tmp\_var();**  **int isOperation(char symbol);**  **int basic\_translator\_let(char \*buf, unit\_command \*command, int \*i\_);**  **#define NMAX 100**  **typedef struct Stack**  **{**  **char str[NMAX];**  **int top;**  **int bot;**  **} Stack;**  **void init\_stack(Stack \*head);**  **char pop\_stack(Stack \*head);**  **void push\_stack(Stack \*head, char str);**  **char get\_head\_elem\_stack(Stack \*head);**  **char pop\_bot\_stack(Stack \*head);**  **#endif** |

**basic.c**

|  |
| --- |
| **#include "basic.h"**  **int basic\_string\_parser\_first(char \*str, int \*i, unit\_command \*unit\_commands, int \*add\_oper, char \*name\_var)**  **{**  **char \*command\_ = malloc(sizeof(char) \* 10);**  **int tmp\_num\_cell[6];**  **for (int k = 0; k < 6; k++)**  **tmp\_num\_cell[k] = enemy\_;**  **int j;**  **for (\*i = 0, j = 0; str[\*i] != '\0' && str[\*i] != ' ' && str[\*i] != '\n'; (\*i)++, j++) {**  **if (isdigit(str[\*i])) {**  **tmp\_num\_cell[j] = (int)str[\*i] - 48;**  **} else {**  **printf("Error incorrect format number");**  **return 1;**  **}**  **}**  **int count;**  **for (count = 0; tmp\_num\_cell[count] != enemy\_; count++) { }**  **int tnc[count];**  **for (int k = 0, n = count - 1; k < count; k++, n--) {**  **tnc[n] = tmp\_num\_cell[k];**  **}**  **for (int k = 0; k < count; k++) {**  **if (k == 0) {**  **unit\_commands->orig\_num\_line = tnc[k];**  **} else {**  **unit\_commands->orig\_num\_line += tnc[k] \* pow(10, k);**  **}**  **}**  **for (; !isalpha(str[\*i]); (\*i)++) { }**  **if (\*(i) < 3) {**  **printf("Too close");**  **return 1;**  **} else if (\*(i) > 10) {**  **printf("Too much distance");**  **return 1;**  **}**  **for (int j = 0; isalpha(str[\*i]); (\*i)++, j++) {**  **if (!isupper(str[\*i])) {**  **printf("Error. Command not must be in lowercase.");**  **return 1;**  **}**  **command\_[j] = str[\*i];**  **}**  **if ((unit\_commands->command = get\_command\_basic(command\_)) == -1) {**  **printf("Incorrect command");**  **return 1;**  **}**  **if (unit\_commands->command > 4) {**  **\*add\_oper = additional\_operations;**  **} else if (unit\_commands->command != REM && unit\_commands->command != END) {**  **for (; !isalpha(str[\*i]); (\*i)++) { }**  **\*name\_var = str[\*i];**  **(\*i)++;**  **var \*time\_var;**  **char short\_name\_var = \*name\_var;**  **if (!(time\_var = get\_var(short\_name\_var))) {**  **if (add\_var(short\_name\_var, get\_cellNumberForNewVariables())) {**  **printf("Sorry \n");**  **return 1;**  **}**  **}**  **}**  **free(command\_);**  **return 0;**  **}**  **int basic\_translator(char \*path\_from, char \*path\_where, int \*i\_)**  **{**  **FILE \*in = fopen(path\_from, "r");**  **if (!in) {**  **printf("No such file.\n");**  **return 1;**  **}**  **FILE \*out = fopen(path\_where, "w");**  **char \*buf = NULL;**  **size\_t len = 0;**  **amount\_lines = 0;**  **while (getline(&buf, &len, in) != -1) {**  **amount\_lines++;**  **}**  **unit\_command \*pull\_commands = malloc(sizeof(unit\_command) \* amount\_lines);**  **fseek(in, 0, SEEK\_SET);**  **int now\_lines = 0;**  **int add\_oper;**  **int real\_line = 0;**  **\*i\_ = 0;**  **int i = \*i\_;**  **while (getline(&buf, &len, in) != -1) {**  **add\_oper = 0;**  **pull\_commands[real\_line].num\_line = now\_lines;**  **char name\_var;**  **if (basic\_string\_parser\_first(buf, &i, &pull\_commands[real\_line], &add\_oper, &name\_var)) {**  **fclose(in);**  **printf(" in %d line\n", now\_lines);**  **printf("%s\n", buf);**  **for (; i != 0; i--) {**  **printf(" ");**  **}**  **mt\_ssetbgcolor(red);**  **printf("^");**  **mt\_stopcolor();**  **printf(" Error is here\n");**    **return 1;**  **}**  **int tmp\_command;**  **var \*tvar;**  **if (add\_oper) {**  **pull\_commands[real\_line].str = malloc(sizeof(char) \* 1000);**  **tmp\_command = pull\_commands[real\_line].command;**  **int dig;**  **char oper\_a;**  **char oper\_b;**  **int operation;**  **int num\_cell\_for\_jump;**  **int temp;**  **switch (tmp\_command) {**  **case GOTO:**  **basic\_translator\_goto(buf, &dig, &i);**  **pull\_commands[real\_line].tmp\_dig = dig;**  **if (isCommandInPull(pull\_commands, dig)) {**  **int num\_line\_to\_ass;**  **if ((num\_line\_to\_ass = get\_num\_line\_to\_ass\_from\_pull(pull\_commands, dig)) == -1) {**  **printf("ERror GOTO.\n");**  **return 1;**  **}**  **if (pull\_commands[real\_line].num\_line < 10) {**  **sprintf(pull\_commands[real\_line].str, "0%d JUMP %d", pull\_commands[real\_line].num\_line, num\_line\_to\_ass);**  **} else {**  **sprintf(pull\_commands[real\_line].str, "%d JUMP %d", pull\_commands[real\_line].num\_line, num\_line\_to\_ass);**  **}**  **} else {**  **pull\_commands[real\_line].command = GOTO\_B;**  **}**  **break;**  **case IF:**  **temp = pull\_commands[real\_line].num\_line;**  **basic\_translator\_if(buf, &oper\_a, &oper\_b, &operation, &i, &num\_cell\_for\_jump);**  **pull\_commands[real\_line].tmp\_dig = num\_cell\_for\_jump;**  **if (isCommandInPull(pull\_commands, num\_cell\_for\_jump)) {**  **int num\_line\_to\_ass;**  **if ((num\_line\_to\_ass = get\_num\_line\_to\_ass\_from\_pull(pull\_commands, num\_cell\_for\_jump)) == -1) {**  **printf("ERror IF.\n");**  **return 1;**  **}**  **if (add\_var(oper\_a, get\_cellNumberForNewVariables())) {**  **printf("Sorry \n");**  **return 1;**  **}**  **var \*vra = get\_var(oper\_a);**  **if (!vra) {**  **printf("AHTUNG.\n");**  **return 1;**  **}**  **if (pull\_commands[real\_line].num\_line < 10) {**  **sprintf(pull\_commands[real\_line].str, "0%d LOAD %d\n", pull\_commands[real\_line].num\_line, vra->num\_cell);**  **} else {**  **sprintf(pull\_commands[real\_line].str, "%d LOAD %d\n", pull\_commands[real\_line].num\_line, vra->num\_cell);**  **}**  **pull\_commands[real\_line].num\_line++;**  **if (operation == EQL) {**  **if (isalpha(oper\_b)) {**  **if (add\_var(oper\_b, get\_cellNumberForNewVariables())) {**  **printf("Sorry \n");**  **return 1;**  **}**  **var \*vrb = get\_var(oper\_b);**  **if (pull\_commands[real\_line].num\_line < 10) {**  **sprintf(pull\_commands[real\_line].str, "%s0%d SUB %d\n", pull\_commands[real\_line].str, pull\_commands[real\_line].num\_line, vrb->num\_cell);**  **} else {**  **sprintf(pull\_commands[real\_line].str, "%s%d SUB %d\n", pull\_commands[real\_line].str, pull\_commands[real\_line].num\_line, vrb->num\_cell);**  **}**  **pull\_commands[real\_line].num\_line++;**  **if (pull\_commands[real\_line].num\_line < 10) {**  **sprintf(pull\_commands[real\_line].str, "%s0%d JZ %d", pull\_commands[real\_line].str, pull\_commands[real\_line].num\_line, num\_line\_to\_ass);**  **} else {**  **sprintf(pull\_commands[real\_line].str, "%s%d JZ %d", pull\_commands[real\_line].str, pull\_commands[real\_line].num\_line, num\_line\_to\_ass);**  **}**  **now\_lines += 2;**  **} else if (isdigit(oper\_b)) {**  **pull\_commands[real\_line].num\_line--;**  **int tmp\_num\_cell\_for\_const = get\_num\_line\_for\_tmp\_var();**  **sprintf(pull\_commands[real\_line].str, "%s%d = +%c\n", pull\_commands[real\_line].str, tmp\_num\_cell\_for\_const, oper\_b);**  **pull\_commands[real\_line].num\_line++;**  **if (pull\_commands[real\_line].num\_line < 10) {**  **sprintf(pull\_commands[real\_line].str, "%s0%d SUB %d\n", pull\_commands[real\_line].str, pull\_commands[real\_line].num\_line, tmp\_num\_cell\_for\_const);**  **} else {**  **sprintf(pull\_commands[real\_line].str, "%s%d SUB %d\n", pull\_commands[real\_line].str, pull\_commands[real\_line].num\_line, tmp\_num\_cell\_for\_const);**  **}**  **pull\_commands[real\_line].num\_line++;**  **// sprintf(pull\_commands[real\_line].str, "%s%d = +0\n", pull\_commands[real\_line].str, tmp\_num\_cell\_for\_const);**  **if (pull\_commands[real\_line].num\_line < 10) {**  **sprintf(pull\_commands[real\_line].str, "%s0%d JZ %d", pull\_commands[real\_line].str, pull\_commands[real\_line].num\_line, num\_line\_to\_ass);**  **} else {**  **sprintf(pull\_commands[real\_line].str, "%s%d JZ %d", pull\_commands[real\_line].str, pull\_commands[real\_line].num\_line, num\_line\_to\_ass);**  **}**  **now\_lines += 2;**  **}**    **} else if (operation == LARGER) {**  **if (isalpha(oper\_b)) {**  **if (add\_var(oper\_b, get\_cellNumberForNewVariables())) {**  **printf("Sorry \n");**  **return 1;**  **}**  **var \*vrb = get\_var(oper\_b);**  **if (pull\_commands[real\_line].num\_line < 10) {**  **sprintf(pull\_commands[real\_line].str, "%s0%d SUB %d\n", pull\_commands[real\_line].str, pull\_commands[real\_line].num\_line, vrb->num\_cell);**  **} else {**  **sprintf(pull\_commands[real\_line].str, "%s%d SUB %d\n", pull\_commands[real\_line].str, pull\_commands[real\_line].num\_line, vrb->num\_cell);**  **}**  **pull\_commands[real\_line].num\_line++;**  **if (pull\_commands[real\_line].num\_line < 10) {**  **sprintf(pull\_commands[real\_line].str, "%s0%d JZ %d", pull\_commands[real\_line].str, pull\_commands[real\_line].num\_line, num\_line\_to\_ass);**  **} else {**  **sprintf(pull\_commands[real\_line].str, "%s%d JZ %d", pull\_commands[real\_line].str, pull\_commands[real\_line].num\_line, num\_line\_to\_ass);**  **}**  **now\_lines += 2;**  **} else if (isdigit(oper\_b)) {**  **pull\_commands[real\_line].num\_line--;**  **int tmp\_num\_cell\_for\_const = get\_num\_line\_for\_tmp\_var();**  **sprintf(pull\_commands[real\_line].str, "%s%d = +%c\n", pull\_commands[real\_line].str, tmp\_num\_cell\_for\_const, oper\_b);**  **pull\_commands[real\_line].num\_line++;**  **if (pull\_commands[real\_line].num\_line < 10) {**  **sprintf(pull\_commands[real\_line].str, "%s0%d SUB %d\n", pull\_commands[real\_line].str, pull\_commands[real\_line].num\_line, tmp\_num\_cell\_for\_const);**  **} else {**  **sprintf(pull\_commands[real\_line].str, "%s%d SUB %d\n", pull\_commands[real\_line].str, pull\_commands[real\_line].num\_line, tmp\_num\_cell\_for\_const);**  **}**  **pull\_commands[real\_line].num\_line++;**  **// sprintf(pull\_commands[real\_line].str, "%s%d = 0\n", pull\_commands[real\_line].str, tmp\_num\_cell\_for\_const);**  **if (pull\_commands[real\_line].num\_line < 10) {**  **sprintf(pull\_commands[real\_line].str, "%s0%d JB %d", pull\_commands[real\_line].str, pull\_commands[real\_line].num\_line, num\_line\_to\_ass);**  **} else {**  **sprintf(pull\_commands[real\_line].str, "%s%d JB %d", pull\_commands[real\_line].str, pull\_commands[real\_line].num\_line, num\_line\_to\_ass);**  **}**  **now\_lines += 2;**  **}**  **} else if (operation == LESS) {**  **if (isalpha(oper\_b)) {**  **if (add\_var(oper\_b, get\_cellNumberForNewVariables())) {**  **printf("Sorry \n");**  **return 1;**  **}**  **printf("test1 = %s\n", pull\_commands[real\_line].str);**  **var \*vrb = get\_var(oper\_b);**  **if (pull\_commands[real\_line].num\_line < 10) {**  **sprintf(pull\_commands[real\_line].str, "%s0%d SUB %d\n", pull\_commands[real\_line].str, pull\_commands[real\_line].num\_line, vrb->num\_cell);**  **} else {**  **sprintf(pull\_commands[real\_line].str, "%s%d SUB %d\n", pull\_commands[real\_line].str, pull\_commands[real\_line].num\_line, vrb->num\_cell);**  **}**  **pull\_commands[real\_line].num\_line++;**  **if (pull\_commands[real\_line].num\_line < 10) {**  **sprintf(pull\_commands[real\_line].str, "%s0%d JZ %d", pull\_commands[real\_line].str, pull\_commands[real\_line].num\_line, num\_line\_to\_ass);**  **} else {**  **sprintf(pull\_commands[real\_line].str, "%s%d JZ %d", pull\_commands[real\_line].str, pull\_commands[real\_line].num\_line, num\_line\_to\_ass);**  **}**  **now\_lines += 2;**  **} else if (isdigit(oper\_b)) {**  **pull\_commands[real\_line].num\_line--;**  **int tmp\_num\_cell\_for\_const = get\_num\_line\_for\_tmp\_var();**  **sprintf(pull\_commands[real\_line].str, "%s%d = +%c\n", pull\_commands[real\_line].str, tmp\_num\_cell\_for\_const, oper\_b);**  **pull\_commands[real\_line].num\_line++;**  **if (pull\_commands[real\_line].num\_line < 10) {**  **sprintf(pull\_commands[real\_line].str, "%s0%d SUB %d\n", pull\_commands[real\_line].str, pull\_commands[real\_line].num\_line, tmp\_num\_cell\_for\_const);**  **} else {**  **sprintf(pull\_commands[real\_line].str, "%s%d SUB %d\n", pull\_commands[real\_line].str, pull\_commands[real\_line].num\_line, tmp\_num\_cell\_for\_const);**  **}**  **pull\_commands[real\_line].num\_line++;**  **// sprintf(pull\_commands[real\_line].str, "%s%d = +0\n", pull\_commands[real\_line].str, tmp\_num\_cell\_for\_const);**  **if (pull\_commands[real\_line].num\_line < 10) {**  **sprintf(pull\_commands[real\_line].str, "%s0%d JNEG %d", pull\_commands[real\_line].str, pull\_commands[real\_line].num\_line, num\_line\_to\_ass);**  **} else {**  **sprintf(pull\_commands[real\_line].str, "%s%d JNEG %d", pull\_commands[real\_line].str, pull\_commands[real\_line].num\_line, num\_line\_to\_ass);**  **}**  **now\_lines += 2;**  **}**  **}**  **} else {**  **pull\_commands[real\_line].command = IF\_B;**  **// pull\_commands[real\_line].str = buf;**  **strcpy(pull\_commands[real\_line].str, buf);**  **now\_lines += 2;**  **}**  **pull\_commands[real\_line].num\_line = temp;**  **break;**  **case LET:**  **temp = pull\_commands[real\_line].num\_line;**  **basic\_translator\_let(buf, &pull\_commands[real\_line], &i);**  **now\_lines = pull\_commands[real\_line].num\_line;**  **now\_lines--;**  **pull\_commands[real\_line].num\_line = temp;**  **break;**  **}**    **} else {**  **tmp\_command = pull\_commands[real\_line].command;**  **switch (tmp\_command) {**  **case REM:**  **now\_lines--;**  **real\_line--;**  **break;**  **case INPUT:**  **tvar = get\_var(name\_var);**  **pull\_commands[real\_line].str = malloc(sizeof(char) \* 20);**  **if (pull\_commands[real\_line].num\_line < 10) {**  **sprintf(pull\_commands[real\_line].str, "0%d READ %d", pull\_commands[real\_line].num\_line, tvar->num\_cell);**  **} else {**  **sprintf(pull\_commands[real\_line].str, "%d READ %d", pull\_commands[real\_line].num\_line, tvar->num\_cell);**  **}**  **break;**  **case OUTPUT:**  **tvar = get\_var(name\_var);**  **pull\_commands[real\_line].str = malloc(sizeof(char) \* 20);**  **if (!tvar) {**  **printf("There is no such variable\n");**  **return 1;**  **break;**  **}**  **if (pull\_commands[real\_line].num\_line < 10) {**  **sprintf(pull\_commands[real\_line].str, "0%d WRITE %d", pull\_commands[real\_line].num\_line, tvar->num\_cell);**  **} else {**  **sprintf(pull\_commands[real\_line].str, "%d WRITE %d", pull\_commands[real\_line].num\_line, tvar->num\_cell);**  **}**  **break;**  **case END:**  **pull\_commands[real\_line].str = malloc(sizeof(char) \* 20);**  **if (pull\_commands[real\_line].num\_line < 10) {**  **sprintf(pull\_commands[real\_line].str, "0%d HALT 00", pull\_commands[real\_line].num\_line);**  **} else {**  **sprintf(pull\_commands[real\_line].str, "%d HALT 00", pull\_commands[real\_line].num\_line);**  **}**  **}**  **}**  **// if (pull\_commands[now\_lines + 1].command != REM) {**  **// printf("\norig = %d\nnum\_line = %d\ncommand = %d\ntmp\_dig = %d\n", pull\_commands[real\_line].orig\_num\_line, pull\_commands[real\_line].num\_line, pull\_commands[real\_line].command, pull\_commands[real\_line].tmp\_dig);**  **// if (pull\_commands[real\_line].str) {**  **// printf("str = %s\n\n", pull\_commands[real\_line].str);**  **// }**  **// }**  **now\_lines++;**  **real\_line++;**  **}**  **// int amount\_vars = get\_amount\_vars()**  **for (int j = 0; j < real\_line; j++) {**  **if (pull\_commands[j].orig\_num\_line % 10 != 0 || pull\_commands[j].orig\_num\_line < 10) {**  **printf("Error format line");**  **\*i\_ = j + 1;**  **return 1;**  **}**  **if (j > 0 && pull\_commands[j - 1].orig\_num\_line != (pull\_commands[j].orig\_num\_line - 10)) {**  **printf("Error number line");**  **\*i\_ = j + 1;**  **return 1;**  **}**  **if (pull\_commands[j].command == GOTO\_B) {**  **int n\_line\_ass = get\_num\_line\_to\_ass\_from\_pull(pull\_commands, pull\_commands[j].tmp\_dig);**  **if (n\_line\_ass == -1) {**  **printf("Error in GOTO\nThere is no such mark %d!\n", pull\_commands[j].tmp\_dig);**  **return 1;**  **}**  **if (pull\_commands[j].num\_line < 10) {**  **sprintf(pull\_commands[j].str, "0%d JUMP %d", pull\_commands[j].num\_line, n\_line\_ass);**  **} else {**  **sprintf(pull\_commands[j].str, "%d JUMP %d", pull\_commands[j].num\_line, n\_line\_ass);**  **}**  **} else if (pull\_commands[j].command == IF\_B) {**  **int temp = pull\_commands[j].num\_line;**  **int num\_cell\_for\_jump;**  **int operation;**  **char oper\_a = 0;**  **char oper\_b = 0;**  **int k;**  **basic\_translator\_if(pull\_commands[j].str, &oper\_a, &oper\_b, &operation, &k, &num\_cell\_for\_jump);**  **int num\_line\_to\_ass;**  **if ((num\_line\_to\_ass = get\_num\_line\_to\_ass\_from\_pull(pull\_commands, num\_cell\_for\_jump)) == -1) {**  **printf("ERror IF\_B.\n");**  **return 1;**  **}**  **if (add\_var(oper\_a, get\_cellNumberForNewVariables())) {**  **printf("Sorry \n");**  **return 1;**  **}**  **var \*vra = get\_var(oper\_a);**  **if (!vra) {**  **printf("AHTUNG IF\_B.\n");**  **return 1;**  **}**  **if (pull\_commands[j].num\_line < 10) {**  **sprintf(pull\_commands[j].str, "0%d LOAD %d\n", pull\_commands[j].num\_line, vra->num\_cell);**  **} else {**  **sprintf(pull\_commands[j].str, "%d LOAD %d\n", pull\_commands[j].num\_line, vra->num\_cell);**  **}**  **pull\_commands[j].num\_line++;**  **if (operation == EQL) {**  **if (isalpha(oper\_b)) {**  **if (add\_var(oper\_b, get\_cellNumberForNewVariables())) {**  **printf("Sorry \n");**  **return 1;**  **}**  **var \*vrb = get\_var(oper\_b);**  **if (pull\_commands[j].num\_line < 10) {**  **sprintf(pull\_commands[j].str, "%s0%d SUB %d\n", pull\_commands[j].str, pull\_commands[j].num\_line, vrb->num\_cell);**  **} else {**  **sprintf(pull\_commands[j].str, "%s%d SUB %d\n", pull\_commands[j].str, pull\_commands[j].num\_line, vrb->num\_cell);**  **}**  **pull\_commands[j].num\_line++;**  **if (pull\_commands[j].num\_line < 10) {**  **sprintf(pull\_commands[j].str, "%s0%d JZ %d", pull\_commands[j].str, pull\_commands[j].num\_line, num\_line\_to\_ass);**  **} else {**  **sprintf(pull\_commands[j].str, "%s%d JZ %d", pull\_commands[j].str, pull\_commands[j].num\_line, num\_line\_to\_ass);**  **}**  **now\_lines += 2;**  **} else if (isdigit(oper\_b)) {**  **pull\_commands[j].num\_line--;**  **int tmp\_num\_cell\_for\_const = get\_num\_line\_for\_tmp\_var();**  **sprintf(pull\_commands[j].str, "%s%d = +%c\n", pull\_commands[j].str, tmp\_num\_cell\_for\_const, oper\_b);**  **pull\_commands[j].num\_line++;**  **if (pull\_commands[j].num\_line < 10) {**  **sprintf(pull\_commands[j].str, "%s0%d SUB %d\n", pull\_commands[j].str, pull\_commands[j].num\_line, tmp\_num\_cell\_for\_const);**  **} else {**  **sprintf(pull\_commands[j].str, "%s%d SUB %d\n", pull\_commands[j].str, pull\_commands[j].num\_line, tmp\_num\_cell\_for\_const);**  **}**  **pull\_commands[j].num\_line++;**    **if (pull\_commands[j].num\_line < 10) {**  **sprintf(pull\_commands[j].str, "%s0%d JZ %d", pull\_commands[j].str, pull\_commands[j].num\_line, num\_line\_to\_ass);**  **} else {**  **sprintf(pull\_commands[j].str, "%s%d JZ %d", pull\_commands[j].str, pull\_commands[j].num\_line, num\_line\_to\_ass);**  **}**  **now\_lines += 2;**  **}**  **} else if (operation == LARGER) {**  **if (isalpha(oper\_b)) {**  **if (add\_var(oper\_b, get\_cellNumberForNewVariables())) {**  **printf("Sorry \n");**  **return 1;**  **}**  **var \*vrb = get\_var(oper\_b);**  **if (pull\_commands[j].num\_line < 10) {**  **sprintf(pull\_commands[j].str, "%s0%d SUB %d\n", pull\_commands[j].str, pull\_commands[j].num\_line, vrb->num\_cell);**  **} else {**  **sprintf(pull\_commands[j].str, "%s%d SUB %d\n", pull\_commands[j].str, pull\_commands[j].num\_line, vrb->num\_cell);**  **}**  **pull\_commands[j].num\_line++;**  **if (pull\_commands[j].num\_line < 10) {**  **sprintf(pull\_commands[j].str, "%s0%d JB %d", pull\_commands[j].str, pull\_commands[j].num\_line, num\_line\_to\_ass);**  **} else {**  **sprintf(pull\_commands[j].str, "%s%d JB %d", pull\_commands[j].str, pull\_commands[j].num\_line, num\_line\_to\_ass);**  **}**  **now\_lines += 2;**  **} else if (isdigit(oper\_b)) {**  **pull\_commands[j].num\_line--;**  **int tmp\_num\_cell\_for\_const = get\_num\_line\_for\_tmp\_var();**  **sprintf(pull\_commands[j].str, "%s%d = +%c\n", pull\_commands[j].str, tmp\_num\_cell\_for\_const, oper\_b);**  **pull\_commands[j].num\_line++;**  **if (pull\_commands[j].num\_line < 10) {**  **sprintf(pull\_commands[j].str, "%s0%d SUB %d\n", pull\_commands[j].str, pull\_commands[j].num\_line, tmp\_num\_cell\_for\_const);**  **} else {**  **sprintf(pull\_commands[j].str, "%s%d SUB %d\n", pull\_commands[j].str, pull\_commands[j].num\_line, tmp\_num\_cell\_for\_const);**  **}**  **pull\_commands[j].num\_line++;**  **if (pull\_commands[j].num\_line < 10) {**  **sprintf(pull\_commands[j].str, "%s0%d JB %d", pull\_commands[j].str, pull\_commands[j].num\_line, num\_line\_to\_ass);**  **} else {**  **sprintf(pull\_commands[j].str, "%s%d JB %d", pull\_commands[j].str, pull\_commands[j].num\_line, num\_line\_to\_ass);**  **}**  **now\_lines += 4;**  **}**  **} else if (operation == LESS) {**  **if (isalpha(oper\_b)) {**  **if (add\_var(oper\_b, get\_cellNumberForNewVariables())) {**  **printf("Sorry \n");**  **return 1;**  **}**  **var \*vrb = get\_var(oper\_b);**  **if (pull\_commands[j].num\_line < 10) {**  **sprintf(pull\_commands[j].str, "%s0%d SUB %d\n", pull\_commands[j].str, pull\_commands[j].num\_line, vrb->num\_cell);**  **} else {**  **sprintf(pull\_commands[j].str, "%s%d SUB %d\n", pull\_commands[j].str, pull\_commands[j].num\_line, vrb->num\_cell);**  **}**  **pull\_commands[j].num\_line++;**  **if (pull\_commands[j].num\_line < 10) {**  **sprintf(pull\_commands[j].str, "%s0%d JZ %d", pull\_commands[j].str, pull\_commands[j].num\_line, num\_line\_to\_ass);**  **} else {**  **sprintf(pull\_commands[j].str, "%s%d JZ %d", pull\_commands[j].str, pull\_commands[j].num\_line, num\_line\_to\_ass);**  **}**  **now\_lines += 4;**  **} else if (isdigit(oper\_b)) {**  **pull\_commands[j].num\_line--;**  **int tmp\_num\_cell\_for\_const = get\_num\_line\_for\_tmp\_var();**  **sprintf(pull\_commands[j].str, "%s%d = +%c\n", pull\_commands[j].str, tmp\_num\_cell\_for\_const, oper\_b);**  **pull\_commands[j].num\_line++;**  **if (pull\_commands[j].num\_line < 10) {**  **sprintf(pull\_commands[j].str, "%s0%d SUB %d\n", pull\_commands[j].str, pull\_commands[j].num\_line, tmp\_num\_cell\_for\_const);**  **} else {**  **sprintf(pull\_commands[j].str, "%s%d SUB %d\n", pull\_commands[j].str, pull\_commands[j].num\_line, tmp\_num\_cell\_for\_const);**  **}**  **pull\_commands[j].num\_line++;**  **if (pull\_commands[j].num\_line < 10) {**  **sprintf(pull\_commands[j].str, "%s0%d JNEG %d", pull\_commands[j].str, pull\_commands[j].num\_line, num\_line\_to\_ass);**  **} else {**  **sprintf(pull\_commands[j].str, "%s%d JNEG %d", pull\_commands[j].str, pull\_commands[j].num\_line, num\_line\_to\_ass);**  **}**  **now\_lines += 4;**  **}**  **}**  **pull\_commands[j].num\_line = temp;**  **now\_lines = temp;**  **}**  **}**  **for (int j = 0; j < real\_line; j++) {**  **fprintf(out, "%s", pull\_commands[j].str);**  **if (j != real\_line - 1) {**  **fprintf(out, "\n");**  **}**  **}**  **fclose(in);**  **fclose(out);**  **return 0;**  **}**  **int get\_command\_basic(char \*str)**  **{**  **if (m\_strcmp(str, "REM"))**  **return REM;**  **if (m\_strcmp(str, "INPUT"))**  **return INPUT;**  **if (m\_strcmp(str, "OUTPUT"))**  **return OUTPUT;**  **if (m\_strcmp(str, "GOTO"))**  **return GOTO;**  **if (m\_strcmp(str, "GOTO\_B"))**  **return GOTO\_B;**  **if (m\_strcmp(str, "IF"))**  **return IF;**  **if (m\_strcmp(str, "LET"))**  **return LET;**  **if (m\_strcmp(str, "END"))**  **return END;**  **return -1;**  **}**  **int add\_var(char name\_, int num\_cell\_)**  **{**  **if (!head\_stack\_of\_vars) {**  **head\_stack\_of\_vars = malloc(sizeof(var));**  **if (!head\_stack\_of\_vars) {**  **printf("Bad alloc head\_stack\_of\_vars\n");**  **return 1;**  **}**  **head\_stack\_of\_vars->name = name\_;**  **head\_stack\_of\_vars->num\_cell = num\_cell\_;**  **head\_stack\_of\_vars->next = NULL;**  **} else {**  **var \*tmp = head\_stack\_of\_vars;**  **var \*prev;**  **while (tmp != NULL) {**  **prev = tmp;**  **tmp = tmp->next;**  **}**  **tmp = malloc(sizeof(var));**  **tmp->name = name\_;**  **tmp->num\_cell = num\_cell\_;**  **prev->next = tmp;**  **}**  **return 0;**  **}**  **var \*get\_var(char name)**  **{**  **if (!head\_stack\_of\_vars) {**  **return NULL;**  **}**  **var \*tmp = head\_stack\_of\_vars;**  **while (tmp != NULL) {**  **if (tmp->name == name) {**  **return tmp;**  **}**  **tmp = tmp->next;**  **}**  **return NULL;**  **}**  **int get\_cellNumberForNewVariables()**  **{**  **if (cell\_number\_for\_variables < 51) {**  **printf("Too many variables\n");**  **exit(1);**  **}**  **return --cell\_number\_for\_variables;**  **}**  **int basic\_translator\_goto(char \*str, int \*dig, int \*i)**  **{**  **for (; !isdigit(str[\*i]); (\*i)++) { }**  **int tmp\_num\_cell[6];**  **for (int k = 0; k < 6; k++)**  **tmp\_num\_cell[k] = enemy\_;**  **int j;**  **for (j = 0; str[\*i] != '\0' && str[\*i] != ' ' && str[\*i] != '\n'; (\*i)++, j++) {**  **if (isdigit(str[\*i])) {**  **tmp\_num\_cell[j] = (int)str[\*i] - 48;**  **} else {**  **if (j == 2 && \*dig == 0) {**  **break;**  **} else {**  **printf("Error incorrect format number");**  **return 1;**  **}**  **}**  **}**  **int count;**  **for (count = 0; tmp\_num\_cell[count] != enemy\_; count++) { }**  **int tnc[count];**  **for (int k = 0, n = count - 1; k < count; k++, n--) {**  **tnc[n] = tmp\_num\_cell[k];**  **}**  **for (int k = 0; k < count; k++) {**  **if (k == 0) {**  **\*dig = tnc[k];**  **} else {**  **\*dig += tnc[k] \* pow(10, k);**  **}**  **}**  **return 0;**  **}**  **int get\_num\_line\_to\_ass\_from\_pull(unit\_command \*pull\_commands, int num)**  **{**  **for (int i = 0; i < amount\_lines; i++) {**  **if (pull\_commands[i].orig\_num\_line == num) {**  **return pull\_commands[i].num\_line;**  **}**  **}**  **return -1;**  **}**  **int isCommandInPull(unit\_command \*pull\_commands, int num)**  **{**  **for (int i = 0; i < amount\_lines; i++) {**  **if (pull\_commands[i].orig\_num\_line == num) {**  **return 1;**  **}**  **}**  **return 0;**  **}**  **int basic\_translator\_if(char \*buf, char \*oper\_a, char \*oper\_b, int \*operation, int \*i, int \*num\_cell\_for\_jump)**  **{**  **\*i = 0;**  **int j;**  **for (; !isalpha(buf[\*i]); (\*i)++) { }**  **for (j = 0; isalpha(buf[\*i]); (\*i)++, j++) {**  **}**  **for (; !isalpha(buf[\*i]); (\*i)++) { }**  **\*oper\_a = buf[\*i];**  **(\*i)++;**  **for (; buf[\*i] != '<' && buf[\*i] != '>' && buf[\*i] != '='; (\*i)++) { }**  **if ('<' == buf[\*i]) {**  **\*operation = LESS;**  **} else if ('>' == buf[\*i]) {**  **\*operation = LARGER;**  **} else if ('=' == buf[\*i]) {**  **\*operation = EQL;**  **} else {**  **\*operation = 0;**  **printf("Error operation.");**  **return 1;**  **}**  **for (; !isalpha(buf[\*i]); (\*i)++) {**  **if (isdigit(buf[\*i])) {**  **break;**  **}**  **}**  **\*oper\_b = buf[\*i];**  **(\*i)++;**    **char \*cmnd = malloc(sizeof(char) \* 5);**  **for (; !isalpha(buf[\*i]); (\*i)++) {}**  **for (j = 0; isalpha(buf[\*i]); (\*i)++, j++) {**  **cmnd[j] = buf[\*i];**  **}**  **if (get\_command\_basic(cmnd) != GOTO) {**  **printf("Error. Need GOTO\n");**  **return 1;**  **}**  **for (; !isdigit(buf[\*i]); (\*i)++) {**  **if (buf[\*i] == '\0' || buf[\*i] == '\n') {**  **printf("Error\n");**  **return 1;**  **}**  **}**  **if (basic\_translator\_goto(buf, num\_cell\_for\_jump, i)) {**  **printf("Error\n");**  **return 1;**  **}**  **return 0;**  **}**  **int get\_num\_line\_for\_tmp\_var()**  **{**  **var \*tmp = head\_stack\_of\_vars;**  **while (tmp->next != NULL) {**  **tmp = tmp->next;**  **}**  **return tmp->num\_cell - 1;**  **}**  **void init\_stack(Stack \*head)**  **{**  **head->top = 0;**  **head->bot = 0;**  **}**  **void push\_stack(Stack \*head, char s)**  **{**  **if (head->top < NMAX) {**  **head->top++;**  **head->str[head->top] = s;**  **} else {**  **printf("Stack is full\n");**  **return;**  **}**  **}**  **char pop\_stack(Stack \*head)**  **{**  **char tmp = 0;**  **if (head->top > 0) {**  **tmp = head->str[head->top];**  **head->str[head->top] = 0;**  **head->top--;**  **} else if (head->top == 0) {**  **tmp = head->str[head->top];**  **head->str[head->top] = 0;**  **}**  **return tmp;**  **}**  **char pop\_bot\_stack(Stack \*head)**  **{**  **if (head->bot > head->top) {**  **return 0;**  **}**  **return head->str[head->bot++];**  **}**  **char get\_head\_elem\_stack(Stack \*head)**  **{**  **return head->str[head->top];**  **}**  **int isOperation(char symbol)**  **{**  **if (symbol == '+' || symbol == '-' || symbol == '\*' || symbol == '/' || symbol == '(' || symbol == ')') {**  **return 1;**  **}**  **return 0;**  **}**  **int basic\_translator\_let(char \*buf, unit\_command \*command, int \*i\_)**  **{**  **int i = \*i\_;**  **for (; !isalpha(buf[i]); i++) { }**  **char var\_where\_store;**  **var\_where\_store = buf[i];**  **i++;**  **var \*var\_store;**  **if (!(var\_store = get\_var(var\_where\_store))) {**  **if (add\_var(var\_where\_store, get\_cellNumberForNewVariables())) {**  **printf("Sorry \n");**  **return 1;**  **}**  **var\_store = get\_var(var\_where\_store);**  **}**  **char \*inf = malloc(sizeof(char) \* (strlen(buf) - i + 4));**  **if (!inf) {**  **printf("Bad alloc\n");**  **return 1;**  **}**  **for (int j = 0; buf[i] != '\0' && buf[i] != '\n'; i++) {**  **if (isalpha(buf[i]) || isdigit(buf[i]) || isOperation(buf[i])) {**  **inf[j] = buf[i];**  **j++;**  **}**  **}**  **inf[strlen(inf)] = '$';**  **Stack \*post = malloc(sizeof(Stack));**  **init\_stack(post);**  **Stack \*in = malloc(sizeof(Stack));**  **init\_stack(in);**  **int status = 2;**  **int j = 0;**  **while (status == 2) {**  **if (isalpha(inf[j]) || isdigit(inf[j])) {**  **push\_stack(post, inf[j]);**  **j++;**  **}**  **char first = get\_head\_elem\_stack(in);**  **if (inf[j] == '+' || inf[j] == '-') {**  **if (first == 0 || first == '(') {**  **push\_stack(in, inf[j]);**  **j++;**  **} else if (first == '+' || first == '-' || first == '\*' || first == '/') {**  **push\_stack(post, pop\_stack(in));**  **}**  **} else if (inf[j] == '\*' || inf[j] == '/') {**  **if (first == 0 || first == '(' || first == '+' || first == '-') {**  **push\_stack(in, inf[j]);**  **j++;**  **} else if (first == '\*' || first == '/') {**  **push\_stack(post, pop\_stack(in));**  **}**  **} else if (inf[j] == '(') {**  **push\_stack(in, inf[j]);**  **j++;**  **} else if (inf[j] == ')') {**  **if (first == 0) {**  **status = 0;**  **} else if (first == '+' || first == '-' || first == '\*' || first == '/') {**  **push\_stack(post, pop\_stack(in));**  **} else if (first == '(') {**  **pop\_stack(in);**  **j++;**  **}**  **} else if (inf[j] == '$') {**  **if (first == 0) {**  **status = 1;**  **} else if (first == '+' || first == '-' || first == '\*' || first == '/') {**  **push\_stack(post, pop\_stack(in));**  **} else if (first == '(') {**  **status = 0;**  **}**  **}**  **}**  **\*i\_ = i;**  **var \*tmp\_var;**  **char name\_tmp\_var;**  **if (post->top > 3) {**  **name\_tmp\_var = 126;**  **if (!(tmp\_var = get\_var(name\_tmp\_var))) {**  **if (add\_var(name\_tmp\_var, get\_cellNumberForNewVariables())) {**  **printf("Sorry \n");**  **return 1;**  **}**  **tmp\_var = get\_var(name\_tmp\_var);**  **}**  **}**  **if (post->top == 1) {**  **char first = pop\_bot\_stack(post);**  **first = pop\_bot\_stack(post);**  **if (isalpha(first)) {**  **var \*var\_A;**  **if (!(var\_A = get\_var(first))) {**  **if (add\_var(first, get\_cellNumberForNewVariables())) {**  **printf("Sorry \n");**  **return 1;**  **}**  **var\_A = get\_var(first);**  **}**  **if (command->num\_line < 10) {**  **sprintf(command->str, "%s0%d LOAD %d\n", command->str, command->num\_line, var\_A->num\_cell);**  **} else {**  **sprintf(command->str, "%s%d LOAD %d\n", command->str, command->num\_line, var\_A->num\_cell);**  **}**  **} else if (isdigit(first)) {**  **if (command->num\_line < 10) {**  **sprintf(command->str, "%s0%d SET %c\n", command->str, command->num\_line, first);**  **} else {**  **sprintf(command->str, "%s%d SET %c\n", command->str, command->num\_line, first);**  **}**  **}**  **command->num\_line++;**  **if (command->num\_line < 10) {**  **sprintf(command->str, "%s0%d STORE %d", command->str, command->num\_line, var\_store->num\_cell);**  **} else {**  **sprintf(command->str, "%s%d STORE %d", command->str, command->num\_line, var\_store->num\_cell);**  **}**  **command->num\_line++;**  **}**  **while (post->bot <= post->top) {**  **char first = pop\_bot\_stack(post);**  **if (isalpha(first) || isdigit(first)) {**  **push\_stack(in, first);**  **} else if (isOperation(first)) {**  **char oper\_a = pop\_stack(in);**  **char oper\_b = pop\_stack(in);**  **if (isalpha(oper\_b) || oper\_b == name\_tmp\_var) {**  **var \*var\_a;**  **if (!(var\_a = get\_var(oper\_b))) {**  **if (add\_var(oper\_b, get\_cellNumberForNewVariables())) {**  **printf("Sorry \n");**  **return 1;**  **}**  **var\_a = get\_var(oper\_b);**  **}**  **if (command->num\_line < 10) {**  **sprintf(command->str, "%s0%d LOAD %d\n", command->str, command->num\_line, var\_a->num\_cell);**  **} else {**  **sprintf(command->str, "%s%d LOAD %d\n", command->str, command->num\_line, var\_a->num\_cell);**  **}**  **} else if (isdigit(oper\_b)) { // TODO**  **if (command->num\_line < 10) {**  **sprintf(command->str, "%s0%d SET %c\n", command->str, command->num\_line, oper\_b);**  **} else {**  **sprintf(command->str, "%s%d SET %c\n", command->str, command->num\_line, oper\_b);**  **}**  **}**  **command->num\_line++;**    **if (isalpha(oper\_a) || oper\_a == name\_tmp\_var) {**  **var \*var\_b;**  **if (!(var\_b = get\_var(oper\_a))) {**  **if (add\_var(oper\_a, get\_cellNumberForNewVariables())) {**  **printf("Sorry \n");**  **return 1;**  **}**  **var\_b = get\_var(oper\_a);**  **}**  **if (command->num\_line < 10) {**  **switch (first) {**  **case '+':**  **sprintf(command->str, "%s0%d ADD %d\n", command->str, command->num\_line, var\_b->num\_cell);**  **break;**  **case '-':**  **sprintf(command->str, "%s0%d SUB %d\n", command->str, command->num\_line, var\_b->num\_cell);**  **break;**  **case '\*':**  **sprintf(command->str, "%s0%d MUL %d\n", command->str, command->num\_line, var\_b->num\_cell);**  **break;**  **case '/':**  **sprintf(command->str, "%s0%d DIVIDE %d\n", command->str, command->num\_line, var\_b->num\_cell);**  **break;**  **}**  **} else {**  **switch (first) {**  **case '+':**  **sprintf(command->str, "%s%d ADD %d\n", command->str, command->num\_line, var\_b->num\_cell);**  **break;**  **case '-':**  **sprintf(command->str, "%s%d SUB %d\n", command->str, command->num\_line, var\_b->num\_cell);**  **break;**  **case '\*':**  **sprintf(command->str, "%s%d MUL %d\n", command->str, command->num\_line, var\_b->num\_cell);**  **break;**  **case '/':**  **sprintf(command->str, "%s%d DIVIDE %d\n", command->str, command->num\_line, var\_b->num\_cell);**  **break;**  **}**  **}**  **} else if (isdigit(oper\_a)) { // TODO**  **int temp\_num\_cell = get\_num\_line\_for\_tmp\_var();**  **if (temp\_num\_cell < 10) {**  **sprintf(command->str, "%s0%d = +%c\n", command->str, temp\_num\_cell, oper\_a);**  **} else {**  **sprintf(command->str, "%s%d = +%c\n", command->str, temp\_num\_cell, oper\_a);**  **}**  **if (command->num\_line < 10) {**  **switch (first) {**  **case '+':**  **sprintf(command->str, "%s0%d ADD %d\n", command->str, command->num\_line, temp\_num\_cell);**  **break;**  **case '-':**  **sprintf(command->str, "%s0%d SUB %d\n", command->str, command->num\_line, temp\_num\_cell);**  **break;**  **case '\*':**  **sprintf(command->str, "%s0%d MUL %d\n", command->str, command->num\_line, temp\_num\_cell);**  **break;**  **case '/':**  **sprintf(command->str, "%s0%d DIVIDE %d\n", command->str, command->num\_line, temp\_num\_cell);**  **break;**  **}**  **} else {**  **switch (first) {**  **case '+':**  **sprintf(command->str, "%s%d ADD %d\n", command->str, command->num\_line, temp\_num\_cell);**  **break;**  **case '-':**  **sprintf(command->str, "%s%d SUB %d\n", command->str, command->num\_line, temp\_num\_cell);**  **break;**  **case '\*':**  **sprintf(command->str, "%s%d MUL %d\n", command->str, command->num\_line, temp\_num\_cell);**  **break;**  **case '/':**  **sprintf(command->str, "%s%d DIVIDE %d\n", command->str, command->num\_line, temp\_num\_cell);**  **break;**  **}**  **}**  **}**  **command->num\_line++;**  **if (post->str[post->bot]) {**  **if (command->num\_line < 10) {**  **sprintf(command->str, "%s0%d STORE %d\n", command->str, command->num\_line, tmp\_var->num\_cell);**  **} else {**  **sprintf(command->str, "%s%d STORE %d\n", command->str, command->num\_line, tmp\_var->num\_cell);**  **}**  **command->num\_line++;**  **push\_stack(in, name\_tmp\_var);**  **} else {**  **if (command->num\_line < 10) {**  **sprintf(command->str, "%s0%d STORE %d", command->str, command->num\_line, var\_store->num\_cell);**  **} else {**  **sprintf(command->str, "%s%d STORE %d", command->str, command->num\_line, var\_store->num\_cell);**  **}**  **command->num\_line++;**  **}**  **}**  **}**  **return 0;**  **}** |

**basic\_main.c**

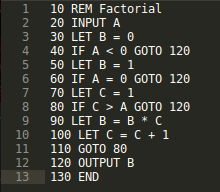
|  |
| --- |
| **#include "basic.h"**  **int main(int argc, char \*\*args)**  **{**  **if (argc == 3) {**  **cell\_number\_for\_variables = 100;**  **int i = 0;**  **if (basic\_translator(args[1], args[2], &i)) {**  **printf(" in %d line\n", i);**  **printf("Please rewrite your code.\n");**  **return 1;**  **}**  **} else {**  **help();**  **return 1;**  **}**  **printf("Finish\n");**  **return 0;**  **}** |

**main.c**

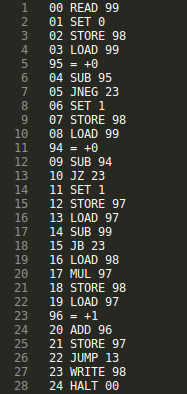
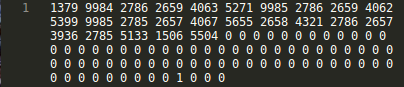
|  |
| --- |
| **#include "helper.h"**  **#include <signal.h>**  **void sighandler(int sig)**  **{**  **printf("Получен сигнал - %d\n", sig);**  **}**  **int presentProgram()**  **{**  **if (cell != 55) {**  **mt\_gotoXY((cell / 10) + 2, (cell % 10) \* 6 + 2);**  **mt\_stopcolor();**  **printCell();**  **}**  **int prepareNumCell[20] = {95, 84, 86, 73, 77, 62, 68, 51, 59, 41, 49, 31, 39, 22, 28, 33, 37, 44, 46, 55};**  **// selectCellMemoryByNumber(0);**  **for (int i = 0; i < 20; i++) {**  **interface(0, 0, 0, 0, 1, 0, 1, 0, 0);**  **selectCellMemoryByNumber(prepareNumCell[i]);**  **ptr\_str[cell] += i \* prepareNumCell[i] + 1;**  **selectCellMemoryByNumber(prepareNumCell[i]);**  **instCount++;**  **if (i > 0 && i < 19) {**  **usleep(300000);**  **i++;**  **interface(0, 0, 0, 0, 1, 0, 1, 0, 0);**  **selectCellMemoryByNumber(prepareNumCell[i]);**  **ptr\_str[cell] += i \* prepareNumCell[i];**  **selectCellMemoryByNumber(prepareNumCell[i]);**  **instCount++;**  **}**  **usleep(300000);**  **}**  **instCount = 0;**  **return 0;**  **}**  **void test()**  **{**  **sc\_commandEncode(10, 78, &ptr\_str[0]); // READ 78**  **sc\_commandEncode(10, 88, &ptr\_str[1]); // READ 88**  **sc\_commandEncode(20, 79, &ptr\_str[2]); // LOAD 79**  **sc\_commandEncode(30, 89, &ptr\_str[3]); // ADD 89**  **sc\_commandEncode(21, 99, &ptr\_str[4]); // STORE 99**  **sc\_commandEncode(20, 78, &ptr\_str[5]); // LOAD 78**  **sc\_commandEncode(30, 88, &ptr\_str[6]); // ADD 88**  **sc\_commandEncode(21, 98, &ptr\_str[7]); // STORE 98**  **sc\_commandEncode(43, 0, &ptr\_str[8]); // HALT**  **}**  **int main()**  **{**  **sc\_memoryInit();**  **sc\_regInit();**  **initNumberCell();**  **initInstCounter();**  **initNumStrForLogs();**  **test();**  **interface(1, 0, 1, 1, 1, 1, 1, 1, 1);**  **enum keys key;**  **enum way w;**  **w = way\_DEFAULT;**  **selectCellMemory(w);**  **while (1) {**  **interface(0, 0, 0, 1, 1, 1, 1, 0, 0);**  **rk\_readKey(&key);**  **if (key == 'q') {**  **break;**  **}**  **int CI\_value;**  **sc\_regGet(CI, &CI\_value);**  **if (CI\_value == 0) {**  **continue;**  **}**    **if (key == UP) {**  **w = way\_UP;**  **selectCellMemory(w);**  **continue;**  **}**  **if (key == LEFT) {**  **w = way\_LEFT;**  **selectCellMemory(w);**  **continue;**  **}**  **if (key == DOWN) {**  **w = way\_DOWN;**  **selectCellMemory(w);**  **continue;**  **}**  **if (key == RIGHT) {**  **w = way\_RIGHT;**  **selectCellMemory(w);**  **continue;**  **}**    **if (key >= 0 && key < 10) {**  **if (ptr\_str[cell] > -65534 && ptr\_str[cell] < 65535) {**  **ptr\_str[cell] += key;**  **}**  **w = way\_DEFAULT;**  **selectCellMemory(w);**  **continue;**  **}**  **if (key == 'a') {**  **if (ptr\_str[cell] > -65534 && ptr\_str[cell] < 65535) {**  **ptr\_str[cell] += 10;**  **}**  **w = way\_DEFAULT;**  **selectCellMemory(w);**  **continue;**  **}**  **if (key == 'b') {**  **if (ptr\_str[cell] > -65534 && ptr\_str[cell] < 65535) {**  **ptr\_str[cell] += 11;**  **}**  **w = way\_DEFAULT;**  **selectCellMemory(w);**  **continue;**  **}**  **if (key == 'c') {**  **if (ptr\_str[cell] > -65534 && ptr\_str[cell] < 65535) {**  **ptr\_str[cell] += 12;**  **}**  **w = way\_DEFAULT;**  **selectCellMemory(w);**  **continue;**  **}**  **if (key == 'd') {**  **if (ptr\_str[cell] > -65534 && ptr\_str[cell] < 65535) {**  **ptr\_str[cell] += 13;**  **}**  **w = way\_DEFAULT;**  **selectCellMemory(w);**  **continue;**  **}**  **if (key == 'e') {**  **if (ptr\_str[cell] > -65534 && ptr\_str[cell] < 65535) {**  **ptr\_str[cell] += 14;**  **}**  **w = way\_DEFAULT;**  **selectCellMemory(w);**  **continue;**  **}**  **if (key == 'f') {**  **if (ptr\_str[cell] > -65534 && ptr\_str[cell] < 65535) {**  **ptr\_str[cell] += 15;**  **}**  **w = way\_DEFAULT;**  **selectCellMemory(w);**  **continue;**  **}**  **if (key == 'p') { //пасхалка**  **sc\_regSet(CI, 0);**  **presentProgram();**  **sc\_regSet(CI, 1);**  **continue;**  **}**  **if (key == MINUS) {**  **if (ptr\_str[cell] > 0) {**  **ptr\_str[cell] \*= -1;**  **w = way\_DEFAULT;**  **selectCellMemory(w);**  **mt\_gotoXY(29, 1);**  **}**  **continue;**  **}**  **if (key == PLUS) {**  **if (ptr\_str[cell] < 0) {**  **ptr\_str[cell] \*= -1;**  **w = way\_DEFAULT;**  **selectCellMemory(w);**  **mt\_gotoXY(29, 1);**  **}**  **continue;**  **}**  **if (key == F5) {**  **accum = ptr\_str[cell];**  **interface(0, 0, 0, 1, 0, 0, 0, 0, 0);**  **continue;**  **}**  **if (key == 'i') {**  **sc\_regInit();**  **sc\_memoryInit();**  **initInstCounter();**  **interface(0, 0, 1, 1, 1, 1, 1, 0, 0);**  **cell = 0;**  **selectCellMemoryByNumber(cell);**  **continue;**  **}**  **if (key == 'l') {**  **mt\_gotoXY(26 + numStrForLogs, 1);**  **incrementNumStrForLogs();**  **printf("Enter path to file: ");**  **// numStrForLogs++;**  **char \*path = calloc(0, sizeof(char) \* 30);**  **scanf("%s", path);**  **load\_prog\_from\_file(path);**  **interface(0, 0, 1, 1, 1, 0, 1, 0, 0);**  **continue;**  **}**  **if (key == 's') {**  **mt\_gotoXY(26 + numStrForLogs, 1);**  **incrementNumStrForLogs();**  **printf("Enter path to file: ");**  **// numStrForLogs++;**  **char \*path = calloc(0, sizeof(char) \* 30);**  **scanf("%s", path);**  **save\_prog\_in\_file(path);**  **interface(0, 0, 1, 1, 1, 0, 1, 0, 0);**  **continue;**  **}**  **if (key == 'r') {**  **sc\_regSet(CI, 0);**  **runtime();**  **sc\_regSet(CI, 1);**  **continue;**  **}**  **if (key == 't') {**  **sc\_regSet(CI, 0);**  **runtime\_OneStep();**  **sc\_regSet(CI, 1);**  **continue;**  **}**  **}**  **mt\_gotoXY(26, 1);**  **return 0;**  **}** |

**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.

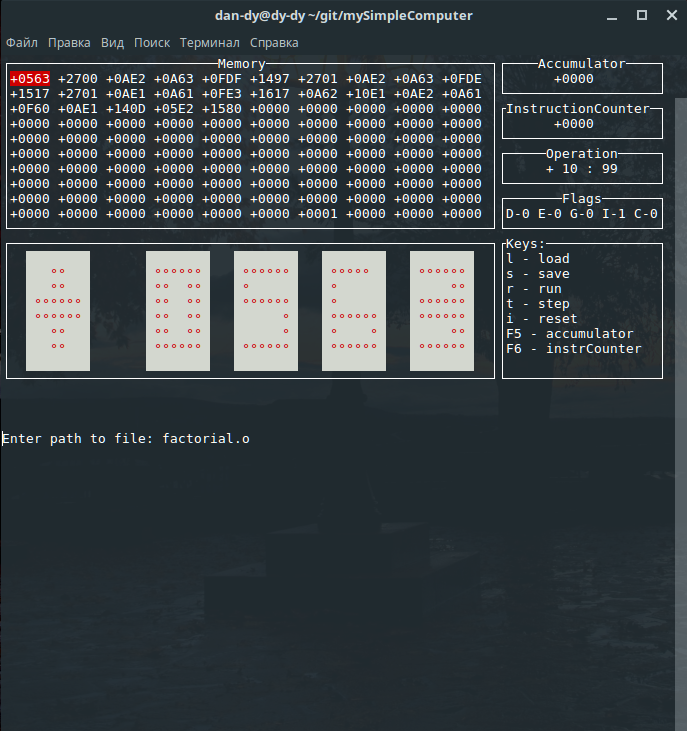
**facrotail.bsc**

**factorial.asm**

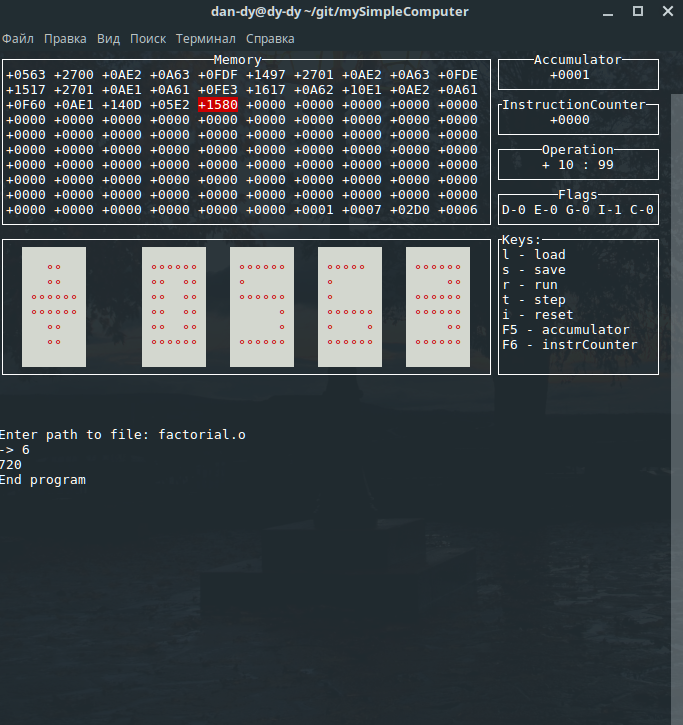


**factorial.o**

Загруженная программа в SimpleComputer



Запуск программы, ввод числа 6 и вывод результата.



**Заключение**

В рамках курсовой работы была реализована обработка команд центральным процессором с помощью функций: ALU() и CU(). Также был реализован транслятор, переводящий код SimpleAssembler в бинарный формат, который может быть считан с помощью SimpleCompiter. И транслятор с высокоуровневого языка SimpleBasic в код SimpleAssembler.

**Литература**

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

2. Архитектура компьютера. 4-е изд. // Э. Танненбаум. – СПб.: Питер, 2003.

3. Организация ЭВМ. 5-е изд. / К. Хамахер, З. Вранешич, С. Заки. – СПб.: Питер; Киев: Изда- тельская группа BHV, 2003.

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

5. Wikipedia –[электронный ресурс]: <https://ru.wikipedia.org>

Дата\_\_\_ \_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_

Подпись\_\_\_\_\_\_\_\_\_/\_\_\_\_\_\_\_\_\_\_\_