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

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 A)
01 READ 10 ; (Ввод B)
02 LOAD 09 ; (Загрузка A в аккумулятор)
03 SUB 10 ; (Отнять B)
04 JNEG 07 ; (Переход на 07, если отрицательное)
05 WRITE 09 ; (Вывод A)
06 HALT 00 ; (Останов)
07 WRITE 10 ; (Вывод B)
08 HALT 00 ; (Останов)
09 = +0000 ; (Переменная A)
10 = +9999 ; (Переменная B)
```

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 С. 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.

.

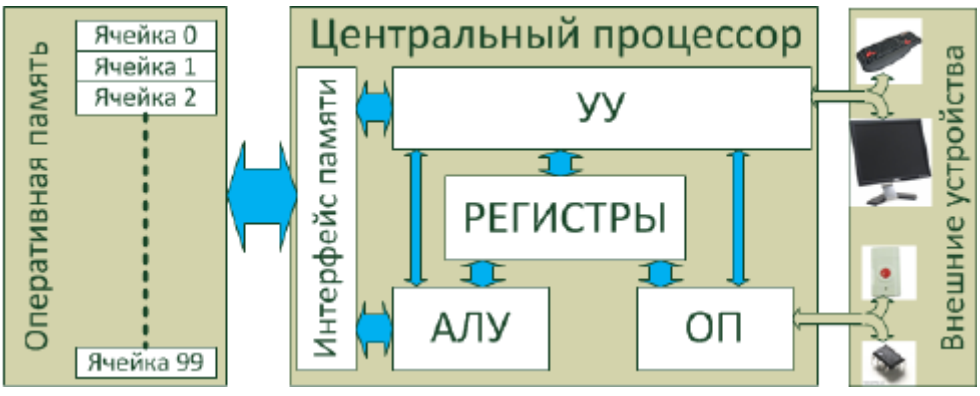


Рисунок 1 – Архитектура вычислительной машины Simple Computer

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».

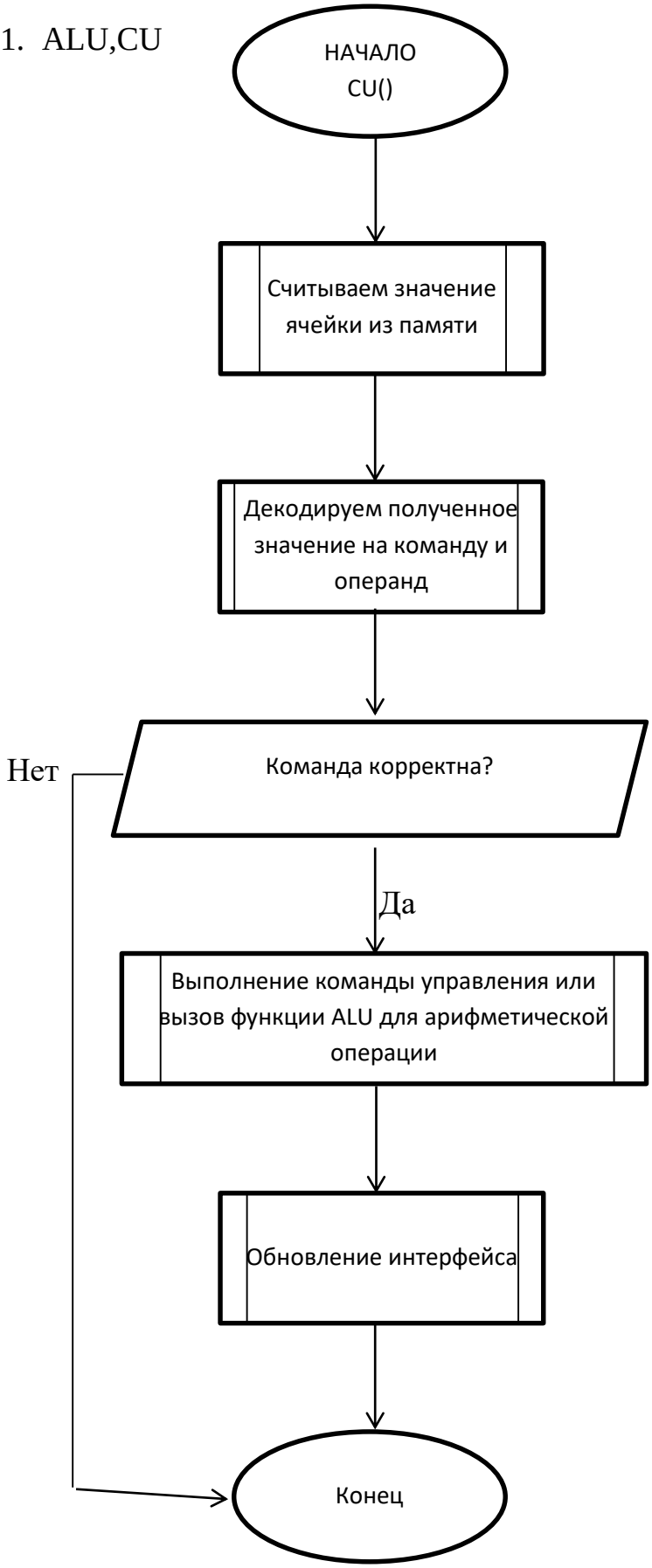
Управляющее устройство (англ. control unit, *CU*) координирует работу центрального процессора. По сути, именно это устройство отвечает за выполнение программы, записанной в оперативной памяти. В его функции входит: чтение текущей команды из памяти, еѐ декодирование, передача номера команды и операнда 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**

1. ALU,CU


НАЧАЛО
CU()
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
Обновление интерфейса
Конец

2.Simple Assembler

3.Simple Basic

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) { // неканоничный
                        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) {// канонический
                        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.

```
10 REM Factorial
20 INPUT A
30 LET B = 0
40 IF A < 0 GOTO 120
50 LET B = 1
60 IF A = 0 GOTO 120
70 LET C = 1
80 IF C > A GOTO 120
90 LET B = B * C
100 LET C = C + 1
110 GOTO 80
120 OUTPUT B
130 END
```

facrotail.bsc

factorial.asm

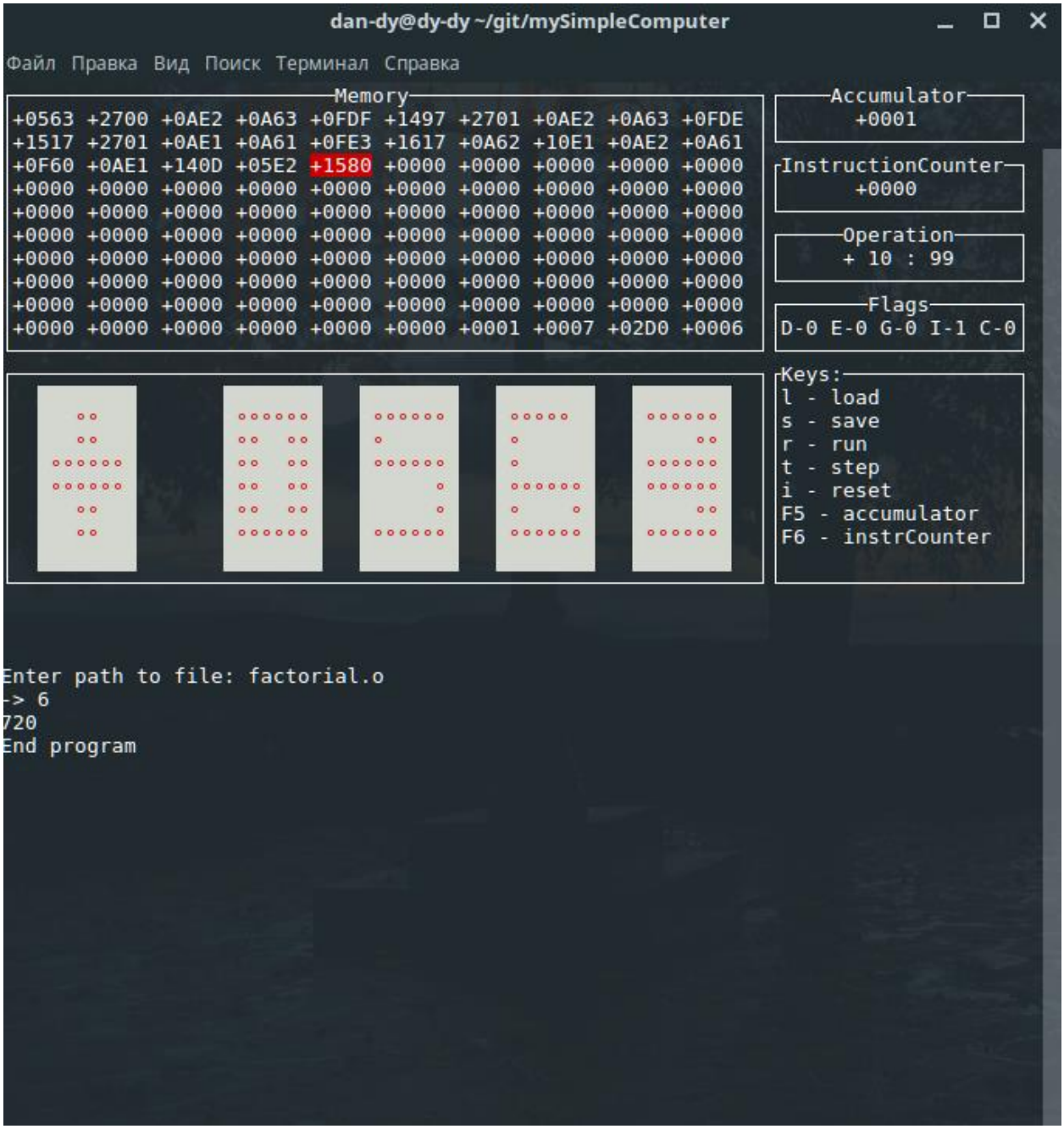
```
00 READ 99
01 SET 0
02 STORE 98
03 LOAD 99
95 = +0
04 SUB 95
05 JNEG 23
06 SET 1
07 STORE 98
08 LOAD 99
94 = +0
09 SUB 94
10 JZ 23
11 SET 1
12 STORE 97
13 LOAD 97
14 SUB 99
15 JB 23
16 LOAD 98
17 MUL 97
18 STORE 98
19 LOAD 97
96 = +1
20 ADD 96
21 STORE 97
22 JUMP 13
23 WRITE 98
24 HALT 00
```

factorial.o

```
1379 9984 2786 2659 4063 5271 9985 2786 2659 4062
5399 9985 2785 2657 4067 5655 2658 4321 2786 2657
3936 2785 5133 1506 5504 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0 0 0
```

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_____/_____