ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ   
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ   
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ   
«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ   
ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

КУРСОВАЯ РАБОТА   
по дисциплине «Архитектура ЭВМ»  
Вариант 7

Выполнил:

Проверил:   
доцент каф. ВС   
Майданов Ю. С.

Новосибирск 2020

Оглавление

1. Постановка задачи…………………………………………………………3
2. Алгоритмы………………………………………………………………… 4
3. Краткое описание работы программы…………………………………....6
4. Результат работы программы………………..…………………………...10
5. Заключение………………………………………………………………..12
6. Листинг программы………………………………………………………13

**1.Постановка задачи**

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

**2.Алгоритмы**

Для выполнения данной курсовой работы было необходимо реализовать несколько алгоритмов, которые обеспечили бы выполнение, поставленных выше, задач.

*Транслятор Simple Basic To Simple Assembler*

Трансляция программы с Simple Basic на Simple Assembler производилась в несколько этапов.

Для начала, требовалось считать имеющийся Simple Basic-код из .bas файла. Для этого проводилось построчное считывание. Из каждой полученной строки считывались номер строки и команда, все последующее считывалось в один общий массив, т.к. дальнейшее разделение параметров команд было невозможно (каждая команда имеет отличные от других параметры).

После получения массива структур, содержащего все данные со считанных строк, происходила, непосредственно, сама трансляция имеющихся команд с Simple Basic на Simple Assembler. В зависимости от того, какая команда содержалась в текущей структуре, выполнялись соответствующие действия.

Ключевой функцией являлось заполнение аналогичного массива структур уже транслированными строками. Стоит заметить, что некоторые команды, представленные в Simple Basic однострочно требовали многострочной записи на Simple Assembler. Так, условия, обрабатываемые оператором IF, обеспечивали вызов пары соответствующих команд Simple Assembler, также при реализации арифметических выражений при операторе LET были задействованы несколько команд Simple Assembler.

Но прежде, было необходимо отсортировать данное выражение в форме обратной польской нотации. Ее принцип заключался в последовательной записи к выходной строке всех переменных и чисел, и параллельной записи знаков арифметических операций и скобок в стек. Значения, записанные в стеке, выносились в выходную строку в следующих случаях: если знак на вершине стека имел одинаковый приоритет со знаком, помещаемым в стек (они «менялись местами»); если в стек заносилась скобка, закрывающая прежде открытые (выносились все, вплоть до изначальной скобки); если считывание из выражения было окончено и в стеке остались элементы.

По завершении обработки каждой строки, имелся массив структур, содержащий в себе все необходимые данные, готовые для записи в .sa файл, что и являлось последним шагом в трансляторе Simple Basic To Simple Assembler.

*Транслятор Simple Assembler в бинарный формат*

Трансляция программы с Simple Assembler на C также проводилась в несколько этапов.

Аналогично считывали имеющуюся программу с .sa файла. Затем, построчно считывая Simple Assembler-код, получали из каждой строки номер ячейки, команду и операнд. После определения команды, происходило, собственно, кодирование. В зависимости от команды или ее отсутствия ячейка соответствующе заполнялась. После полного заполнения массива, он записывался в .o файл.

Таким образом, был реализован транслятор Simple Assembler в бинарный формат.

**Краткое описание работы программы**

Данная программная модель представляет простейшую вычислительную машину Simple Computer.

Для более удобного управления пользователю представлен интерфейс консоли. Его области представляют содержимое оперативной памяти компьютера, значение аккумулятора, значение регистра «Счетчик Команд», результат декодирования операции, состояние регистров флагов, подсказка по функциональным клавишам, а также область, используемая компьютером в процессе выполнения программы для ввода информации с клавиатуры и вывода ее на экран.

Содержимое ячеек памяти и регистров центрального процессора выводится в декодированном виде. При этом, знак «+» соответствует значению 0 в поле «признак команды», следующие две цифры – номер команды и затем операнд в шестнадцатеричной системе счисления.

С оперативной памятью — главной областью действий — предоставляются некоторые возможности взаимодействия. Так, помимо безусловного передвижения курсора по памяти (с помощью стрелок на клавиатуре влево/вправо) и перезаписи значения выделенных ячеек(.), пользователю доступны загрузка(l) и сохранения(s) .o файла, содержащего выполняемую программу, а также, соответственно, запуск программы(r), выполнение текущей команды(t) и обнуление содержимого памяти и регистров(i), стоит отметить и возможность перезаписывать значения аккумулятора(F5) и счетчика команд(F6).

Архитектура Simple Computer - включает следующие функциональные блоки:

· оперативную память;

· внешние устройства;

· центральный процессор.

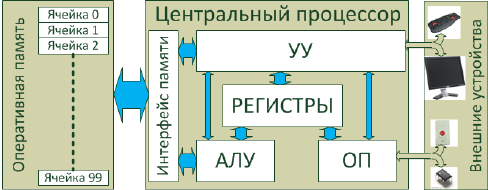


Рисунок 1 – Архитектура вычислительной машины Simple Computer

*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 её адресом и задаётся 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.

Реализация несложных алгоритмов доступна через написание кода на двух транслируемых языках: Basic и Assembler. Первый переводится на второй, а последний уже в бинарный формат, который может быть считан консолью управления.

*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, счётчик команд, указывающий на адрес ячейки памяти, в которой хранится текущая выполняемая команда и регистр флагов, сигнализирующий об определённых событиях. Аккумулятор имеет разрядность 15 бит, счётчика 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, её 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.

*Система команд Simple Computer*

Получив текущую команду из оперативной памяти, устройство управления декодирует её с целью определить номер функции, которую надо выполнить и операнд. Формат команды следующий.



Рисунок 2 – Формат команды центрального процесса Simple Computer

*Выполнение команд центральным процессором Simple Computer*

Команды выполняются последовательно. Адрес ячейки памяти, в которой находится текущая выполняемая команда, задаётся в регистре «Счётчик команд». Устройство управления запрашивает содержимое указанной ячейки памяти и декодирует его согласно используемому формату команд. Получив код операции, устройство управления определяет, является ли эта операция арифметико-логической. Если да, то выполнение операции передаётся в АЛУ. В противном случае операция выполняется устройством управления. Процедура выполняется до тех пор, пока флаг «останов» не будет равен 1.

*Транслятор с языка Simple Assembler*

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*

В языке Simple Basic используются следующие операторы: REM, INPUT, PRINT, GOTO, IT, LET, END.

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

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

**Результат работы программы**

В качестве теста, в файл .bas на Simple Basic был помещен алгоритм факториала:

10 INPUT A   
20 LET B = 1  
30 IF A = 0 GOTO 70  
40 LET B = B \* A   
50 LET A = A - 1   
60 GOTO 30   
70 PRINT B   
80 END

Ниже представлены результаты программной модели при выполнении данной задачи.

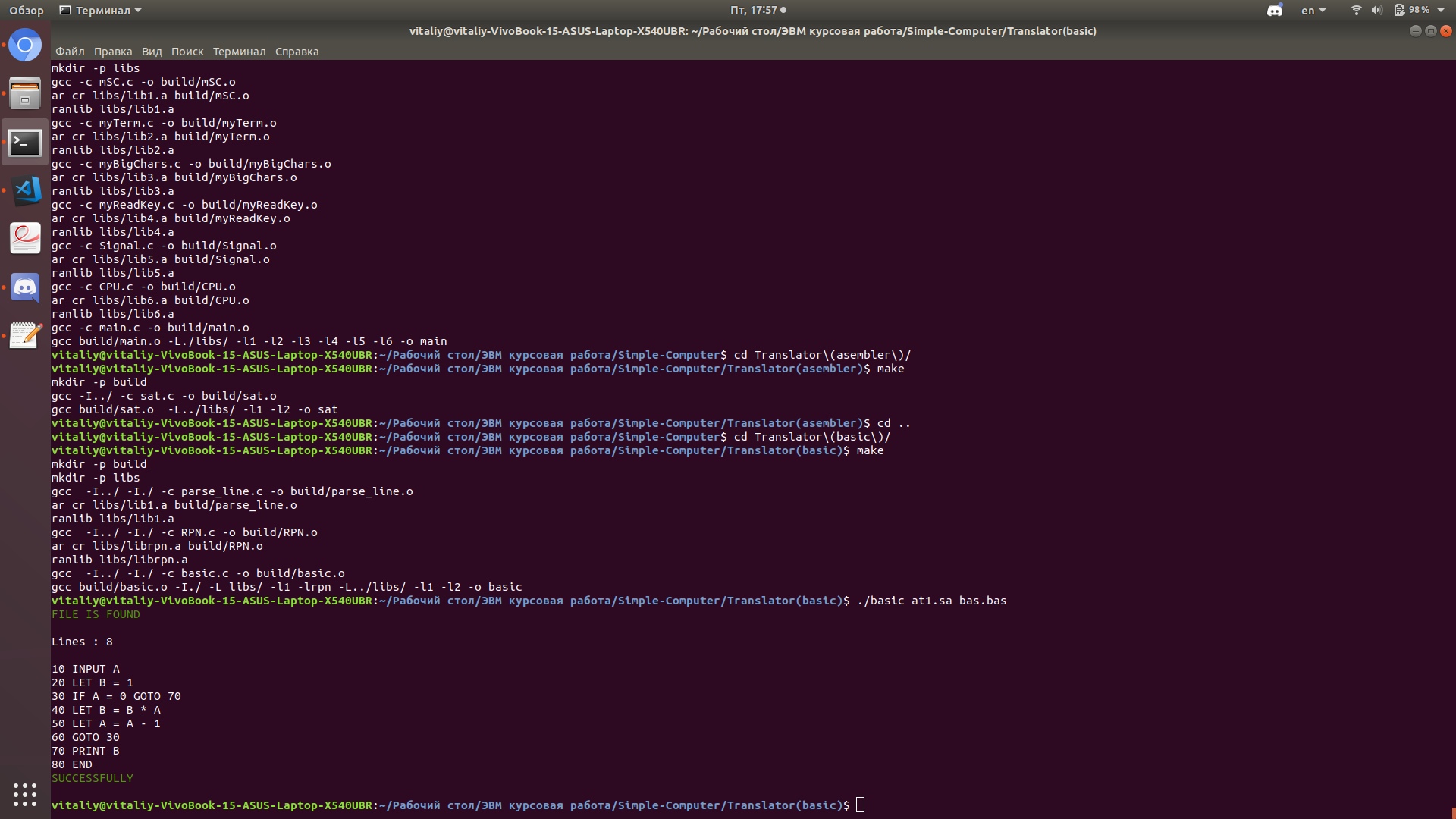


Рис. 1 Вывод результата работы транслятора Simple Basic To Simple Assembler

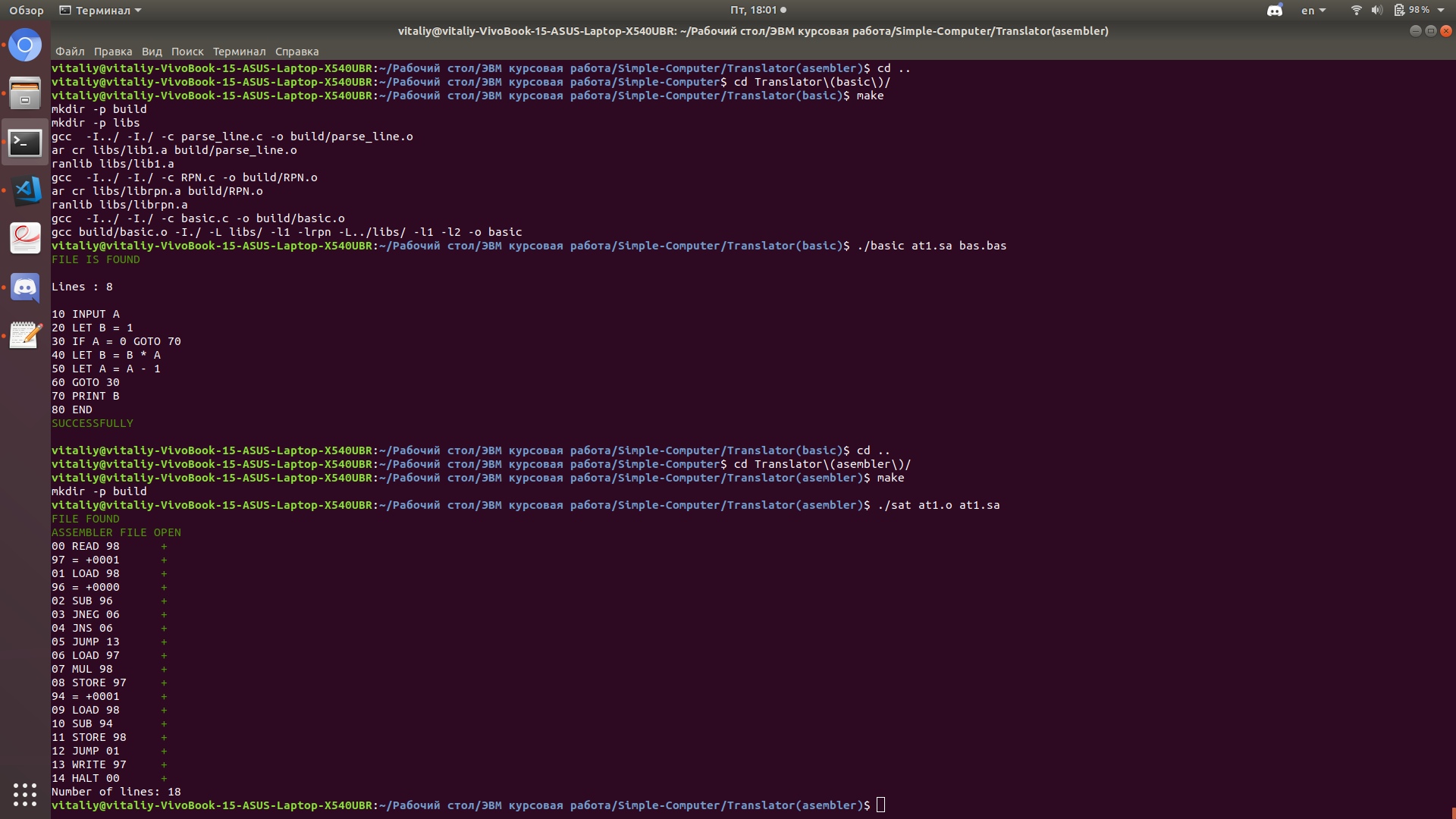


Рис. 2 Вывод результата работы транслятора Simple Assembler в бинарный формат

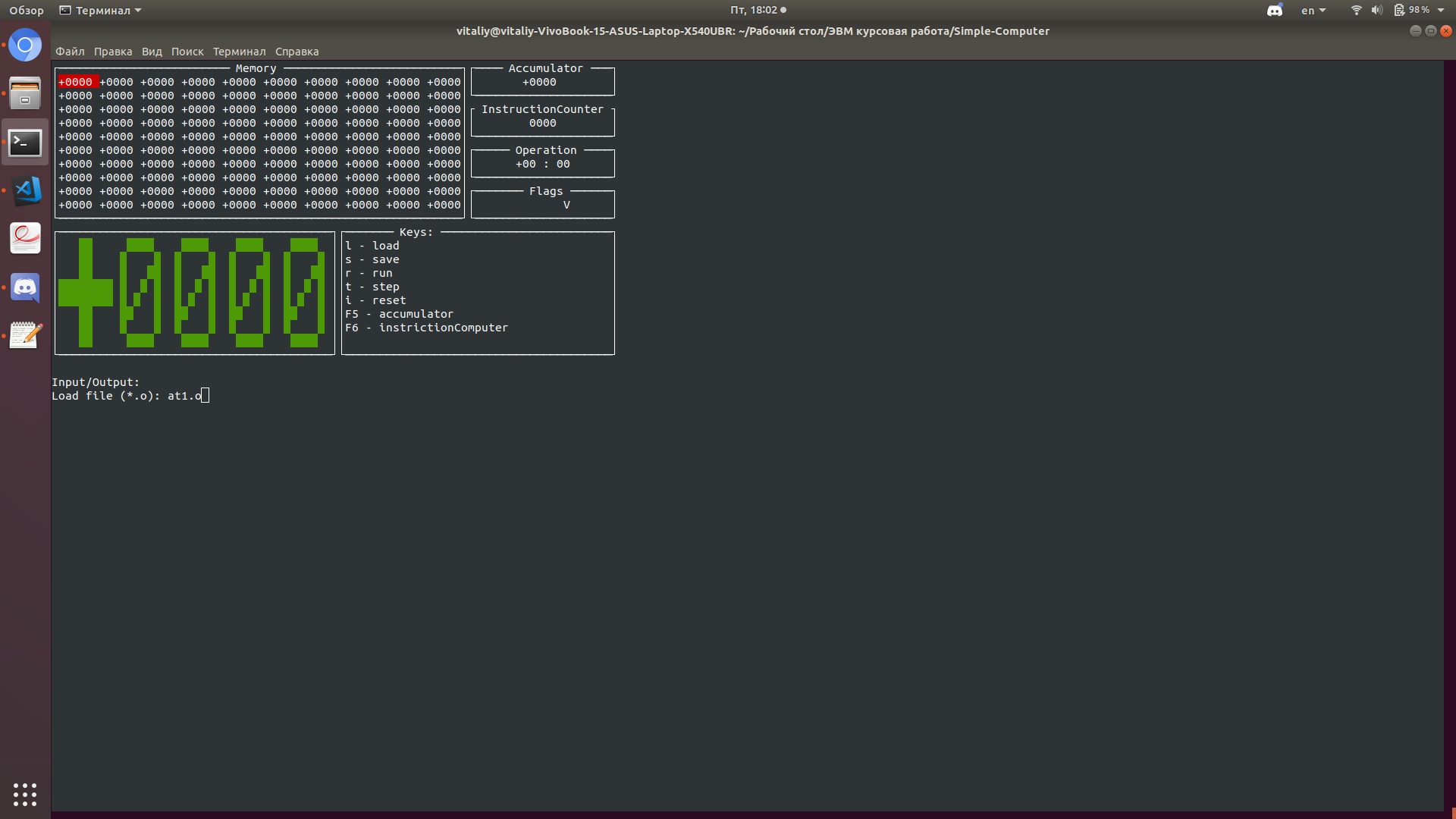


Рис. 3 Ввод имени загружаемого .o файла

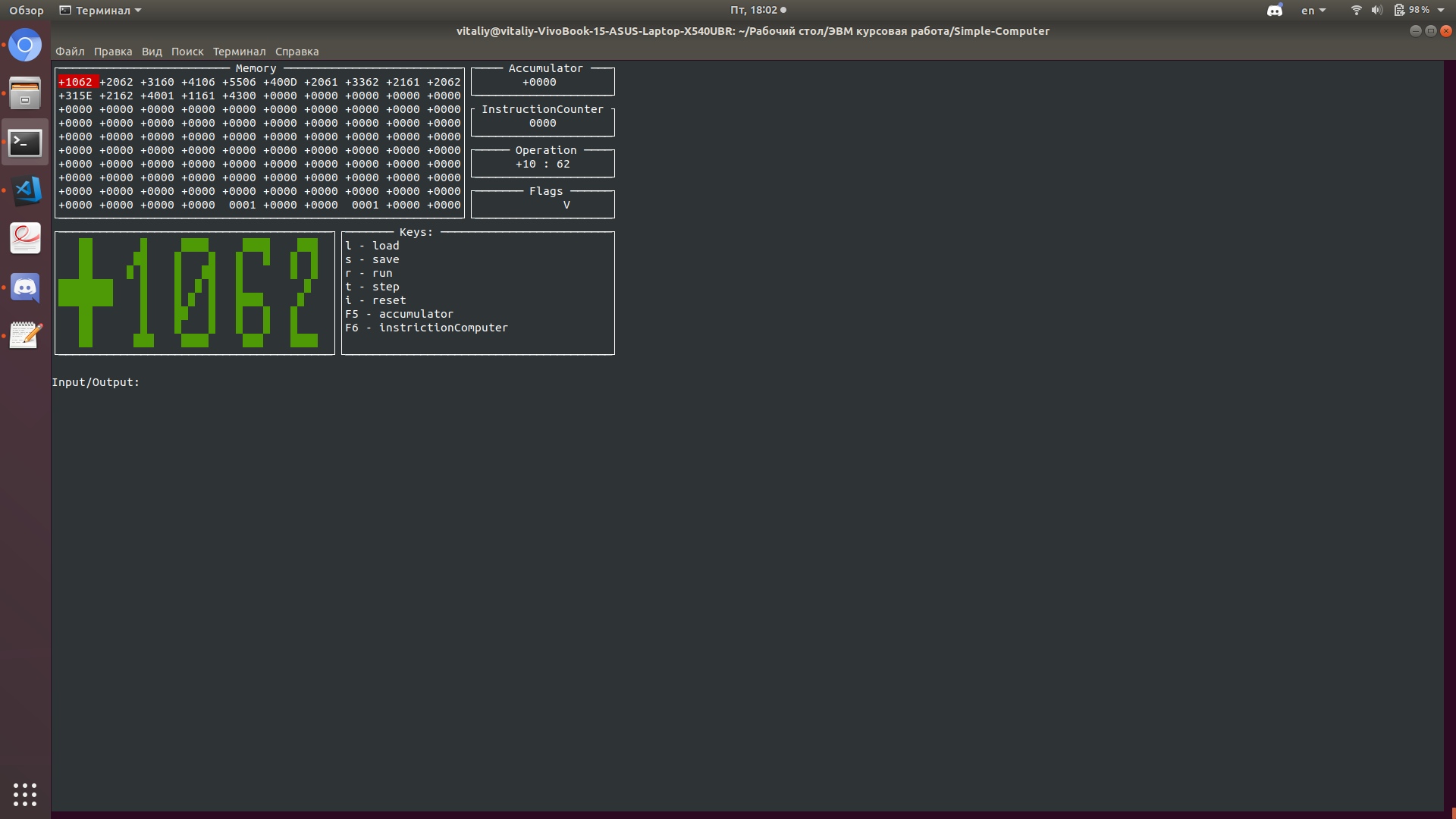


Рис. 4 Файл загружен в модель Simple Computer

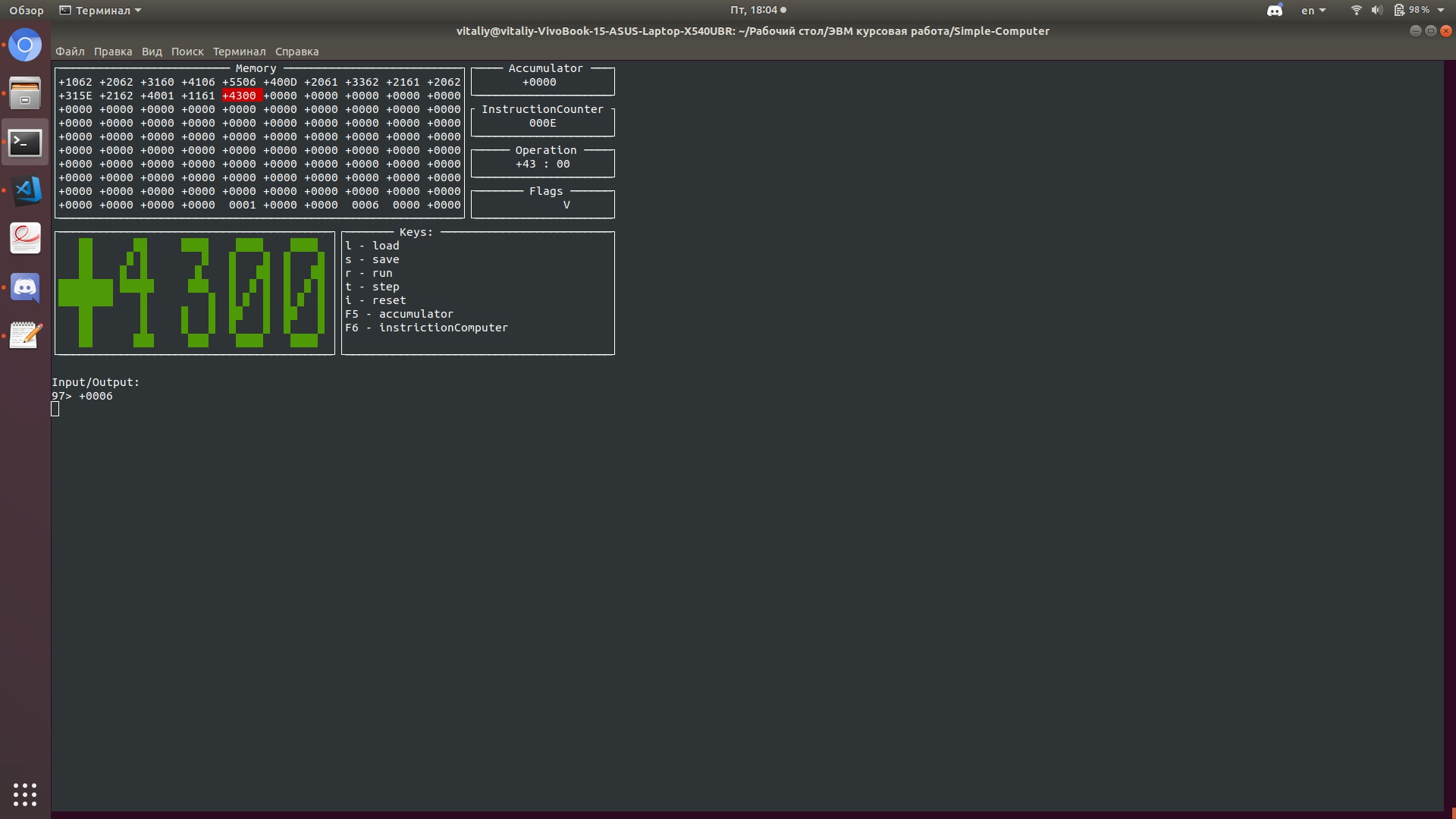


Рис. 5 Результат работы программы – факториал трех

**Заключение**

В рамках данной курсовой работы была реализована модель Simple Computer, выполнение программ которой было осуществлено реализованными функциональными блоками центрального процессора. Для разработки программы были созданы трансляторы с языков Simple Assembler и Simple Basic.

**Листинг программы**

**main.c:**

#include <stdio.h>

#include <unistd.h>

#include "mySimpleComputer.h"

#include "myBigChars.h"

#include "myTerm.h"

#include "myReadKey.h"

#include <stdlib.h>

#include <string.h>

#include <signal.h>

#include "Signal.h"

#include <sys/time.h>

#include "CPU.h"

int main()

{

enum keys Key = Default;

int paintCount = 0;

int regime = 0;

char file\_name[50];

rk\_mytermregime(1, 1, 0, 1, 0);

signal(SIGUSR1, s\_signalhandler);

signal(SIGALRM, s\_signalhandler);

sc\_memoryInit();

sc\_accInit();

sc\_regInit();

sc\_instrcInit();

sc\_regSet(IGNORING\_TACT\_PULSES, 1);

int flag\_check = 0;

while (Key != KeyQuit)

{

bc\_drawInterfase(paintCount);

sc\_instrcGet(&paintCount);

rk\_readkey(&Key);

sc\_regGet(IGNORING\_TACT\_PULSES, &flag\_check);

if (flag\_check)

{

switch (Key)

{

case KeyLoad:

{

bc\_drawInterfase(paintCount);

printf("Load file (\*.o): ");

rk\_mytermregime(0, 0, 1, 0, 1);

fgets(file\_name, 50, stdin);

file\_name[strlen(file\_name) - 1] = '\0';

if (sc\_memoryLoad(file\_name))

{

printf("\nFile not open/found!");

getchar();

}

Key = Default;

rk\_mytermregime(1, 1, 0, 1, 0);

break;

}

case KeySave:

{

bc\_drawInterfase(paintCount);

printf("Save file: ");

rk\_mytermregime(0, 0, 1, 0, 1);

fgets(file\_name, 50, stdin);

file\_name[strlen(file\_name) - 1] = '\0';

if (sc\_memorySave(file\_name))

{

printf("\nFile not save!");

getchar();

}

Key = Default;

rk\_mytermregime(1, 1, 0, 1, 0);

break;

}

case KeyRight:

if (paintCount < 99)

{

paintCount++;

sc\_instrcSet(paintCount);

}

else

{

paintCount = 0;

sc\_instrcSet(paintCount);

}

Key = Default;

break;

case KeyLeft:

if (paintCount > 0)

{

paintCount--;

sc\_instrcSet(paintCount);

}

else

{

paintCount = 99;

sc\_instrcSet(paintCount);

}

Key = Default;

break;

case KeyUp:

{

if (paintCount - 10 >= 0)

{

paintCount -= 10;

sc\_instrcSet(paintCount);

}

else

{

paintCount = 90 + paintCount % 10;

sc\_instrcSet(paintCount);

}

Key = Default;

break;

}

case KeyDown:

{

if (paintCount + 10 <= 99)

{

paintCount += 10;

sc\_instrcSet(paintCount);

}

else

{

paintCount = paintCount % 10;

sc\_instrcSet(paintCount);

}

Key = Default;

break;

}

case KeySetCellValue:

{

rk\_mytermregime(0, 0, 1, 0, 1);

int instr\_c = 0;

sc\_instrcGet(&instr\_c);

bc\_drawInterfase(instr\_c);

printf("%02d< ", paintCount);

char value[10];

fgets(value, 10, stdin);

int val\_copy = 0;

value[strlen(value) - 1] = '\0';

if (strlen(value) == 5 && (value[0] == '+' || value[0] == '-'))

{

sscanf(value + 1, "%04X", &val\_copy);

if (val\_copy <= 0x3FFF && val\_copy >= -0x3FFF && check16(value) != 1)

{

if (value[0] == '-')

{

val\_copy \*= (-1);

if (val\_copy == 0)

{

val\_copy = val\_copy | (1 << 14);

}

}

else

{

val\_copy = val\_copy | (1 << 14);

}

if (sc\_memorySet(paintCount, val\_copy) == 1)

{

printf("\nBorder overload");

getchar();

}

}

else

{

printf("\nError memory overflow or not number");

getchar();

}

}

else

{

printf("\nFew or many arguments");

getchar();

}

rk\_mytermregime(1, 1, 0, 1, 0);

Key = Default;

break;

}

case KeyAccumulator:

{

rk\_mytermregime(0, 0, 1, 0, 1);

int instr\_c = 0;

sc\_instrcGet(&instr\_c);

bc\_drawInterfase(instr\_c);

printf("Accumulator< ");

char value[10];

fgets(value, 10, stdin);

int val\_copy = 0;

value[strlen(value) - 1] = '\0';

if (strlen(value) == 5 && (value[0] == '+' || value[0] == '-'))

{

sscanf(value + 1, "%04X", &val\_copy);

if (val\_copy <= 0x3FFF && val\_copy >= -0x3FFF && check16(value) != 1)

{

if (value[0] == '-')

{

val\_copy \*= (-1);

}

sc\_accSet(val\_copy);

}

else

{

printf("\nError memory overflow or not number");

getchar();

}

}

else

{

printf("\nFew or many arguments");

getchar();

}

rk\_mytermregime(1, 1, 0, 1, 0);

Key = Default;

break;

}

case KeyInstructionCounter:

{

rk\_mytermregime(0, 0, 1, 0, 1);

bc\_drawInterfase(paintCount);

char wrinstr[10];

printf("Input value InstructionCounter: ");

fgets(wrinstr, 10, stdin);

int instr\_copy = 0;

wrinstr[strlen(wrinstr) - 1] = '\0';

if (strlen(wrinstr) <= 5 && wrinstr[0] == '+' && check16(wrinstr) != 1)

{

sscanf(wrinstr + 1, "%04X", &instr\_copy);

if (instr\_copy >= 0x0000 && instr\_copy <= 0x63)

{

sc\_instrcSet(instr\_copy);

}

else

{

printf("\nError border");

getchar();

}

}

else

{

printf("\nError format number");

getchar();

}

rk\_mytermregime(1, 1, 0, 1, 0);

Key = Default;

break;

}

case KeyStep:

{

CU();

Key = Default;

break;

}

default:

break;

}

}

switch (Key)

{

case KeyReset:

{

raise(SIGUSR1);

Key = Default;

break;

}

case KeyRun:

{

int fl = 0;

sc\_regGet(IGNORING\_TACT\_PULSES, &fl);

if (fl)

{

sc\_regSet(IGNORING\_TACT\_PULSES, 0);

raise(SIGALRM);

}

else

{

alarm(0);

sc\_regSet(IGNORING\_TACT\_PULSES, 1);

}

Key = Default;

break;

}

default:

break;

}

}

rk\_mytermregime(0, 0, 1, 1, 1);

mt\_setbgcolor(reset);

mt\_clrscr();

printf("\n");

return 0;

}

**mSC.h:**

#include "mySimpleComputer.h"

struct mySimpleComputer registers;

int sc\_memoryInit()

{

if (RandomAccessMemory != NULL)

{

for (int i = 0; i < 100; i++)

{

RandomAccessMemory[i] = 0;

}

return 0;

}

else

{

return 1;

}

}

int sc\_printMemory(int numberPaint)

{

enum colors fon = black;

for (int i = 0, stroka = 2; i < 100;)

{

mt\_gotoXY(2, stroka);

for (int j = 0; j < 10; j++, i++)

{

if (i == numberPaint)

{

mt\_setbgcolor(red);

}

else

{

mt\_setbgcolor(fon);

}

int oper = 0, com = 0, bit = 0;

int buffer = RandomAccessMemory[i];

bit = (buffer >> 14) & 1;

if (!bit)

{

com = com | ((buffer >> 7) & 0x7F);

oper = oper | (buffer & 0x7F);

printf("+%02X%02X ", com, oper);

}

else

{

if (buffer < 0x0000)

{

printf("-%04X ", buffer \* (-1));

}

else

{

buffer = buffer & 0x3FFF;

printf(" %04X ", buffer);

}

}

mt\_setbgcolor(black);

}

stroka++;

}

}

int sc\_memorySet(int address, int value)

{

if (RandomAccessMemory != NULL)

{

if (address >= 0 && address <= 99)

{

RandomAccessMemory[address] = value;

return 0;

}

else

{

sc\_regSet(BORDER\_OVERLOAD\_ERROR, 1);

return 1;

}

}

else

{

return 1;

}

}

int sc\_memoryGet(int address, int \*value)

{

if (RandomAccessMemory != NULL)

{

if (address >= 0 && address <= 99)

{

(\*value) = RandomAccessMemory[address];

return 0;

}

else

{

sc\_regSet(BORDER\_OVERLOAD\_ERROR, 1);

return 1;

}

}

else

{

return 1;

}

}

int sc\_memorySave(char \*filename)

{

FILE \*RAMSave = NULL;

RAMSave = fopen(filename, "wb");

if (RAMSave == NULL)

{

return 1;

}

fwrite(&RandomAccessMemory, sizeof(int), 100, RAMSave);

fclose(RAMSave);

return 0;

}

int sc\_memoryLoad(char \*filename)

{

if (RandomAccessMemory != NULL)

{

FILE \*RAMLoad = NULL;

RAMLoad = fopen(filename, "rb");

if (RAMLoad == NULL)

{

return 1;

}

int LoadDataRam[100];

fread(LoadDataRam, sizeof(int), 100, RAMLoad);

fclose(RAMLoad);

for (int i = 0; i < 100; i++)

{

RandomAccessMemory[i] = LoadDataRam[i];

}

return 0;

}

else

{

return 1;

}

}

int sc\_regInit(void)

{

registers.register\_flag = 0;

return 0;

}

int sc\_accInit(void)

{

registers.accumulator = 0;

return 0;

}

int sc\_instrcInit(void)

{

registers.instruction\_counter = 0;

return 0;

}

int sc\_regSet(int reg, int value)

{

if ((reg <= CODE\_ERROR) && (reg >= MEMORY\_OVERFLOW) && (value == 0 || value == 1))

{

if (value == 1)

{

registers.register\_flag = registers.register\_flag | (value << reg);

return 0;

}

else if (value == 0)

{

registers.register\_flag = registers.register\_flag & (~(1 << reg));

return 0;

}

}

return 1;

}

int sc\_accSet(int value)

{

registers.accumulator = value;

return 0;

}

int sc\_instrcSet(int value)

{

registers.instruction\_counter = value;

return 0;

}

int sc\_regGet(int reg, int \*value)

{

if ((reg <= CODE\_ERROR) && (reg >= MEMORY\_OVERFLOW))

{

(\*value) = (registers.register\_flag >> reg) & 1;

return 0;

}

return 1;

}

int sc\_accGet(int \*value)

{

(\*value) = registers.accumulator;

return 0;

}

int sc\_instrcGet(int \*value)

{

(\*value) = registers.instruction\_counter;

return 0;

}

int sc\_commandEncode(int command, int operand, int \*value)

{

if ((command <= 127) && (command >= 0) && (operand <= 99) && (operand >= 0))

{

if ((command >= 0x10 && command <= 0x11) ||

(command >= 0x20 && command <= 0x21) ||

(command >= 0x30 && command <= 0x33) ||

(command >= 0x40 && command <= 0x43) ||

(command >= 0x51 && command <= 0x76))

{

(\*value) = 0;

for (int i = 0, j = 7; i < 7 && j < 15; i++, j++)

{

int checkBitCommand = (command >> i) & 1;

int checkBitOperand = (operand >> i) & 1;

if (checkBitCommand == 1)

{

(\*value) = (\*value) | (1 << j);

}

if (checkBitOperand == 1)

{

(\*value) = (\*value) | (1 << i);

}

}

sc\_regSet(BORDER\_OVERLOAD\_ERROR, 0);

return 0;

}

else

{

sc\_regSet(CODE\_ERROR, 0);

return 1;

}

}

else

{

sc\_regSet(BORDER\_OVERLOAD\_ERROR, 1);

return 1;

}

}

int sc\_commandDecode(int value, int \*command, int \*operand)

{

int checkCom = (value >> 14) & 1;

if (checkCom == 0)

{

(\*operand) = 0;

(\*command) = 0;

for (int i = 0, j = 7; i < 7 && j < 14; i++, j++)

{

int checkBitCommand = (value >> j) & 1;

int checkBitOperand = (value >> i) & 1;

if (checkBitCommand == 1)

{

(\*command) = (\*command) | (1 << i);

}

if (checkBitOperand == 1)

{

(\*operand) = (\*operand) | (1 << i);

}

}

return 0;

}

else

{

return 1;

}

}

void sc\_printRegFlag(void)

{

mt\_gotoXY(64, 11);

for (int i = 0; i < 5; i++)

{

int CheckBitRegister = (registers.register\_flag >> i) & 1;

char f = '\0';

switch (i)

{

case 0:

f = 'P'; //M

break;

case 1:

f = '0'; //D

break;

case 2:

f = 'E'; //B

break;

case 3:

f = 'V'; //I

break;

case 4:

f = 'M'; //C

break;

default:

break;

}

mt\_setbgcolor(black);

if (CheckBitRegister == 1)

{

printf("%c ", f);

}

else

{

printf(" ");

}

}

}

void sc\_printAccum(void)

{

mt\_setbgcolor(black);

mt\_gotoXY(70, 2);

int buffer = registers.accumulator;

if (buffer < 0x0000)

{

printf("-%04X ", buffer \* (-1));

}

else

{

printf("+%04X ", buffer);

}

}

void sc\_printInstrCount(void)

{

mt\_setbgcolor(black);

mt\_gotoXY(71, 5);

printf("%04X", registers.instruction\_counter);

}

void sc\_printOper(void)

{

mt\_setbgcolor(black);

mt\_gotoXY(69, 8);

int command = 0, operand = 0, value = 0;

sc\_memoryGet(registers.instruction\_counter, &value);

sc\_commandDecode(value, &command, &operand);

printf("+%2.2x : %2.2x", command, operand);

}

**mySimpleComputer.h:**

#pragma once

#include <stddef.h>

#include <stdio.h>

#include "myTerm.h"

#define MEMORY\_OVERFLOW 0 //переполнение при выполнении операции

#define DIVISION\_ERROR\_BY\_0 1 //ошибка деления на ноль

#define BORDER\_OVERLOAD\_ERROR 2 //ошибка выхода за границы памяти

#define IGNORING\_TACT\_PULSES 3 //игнорирование тактовых импульсов

#define CODE\_ERROR 4 //не верная команда

static int RandomAccessMemory[100];

struct mySimpleComputer

{

int register\_flag : 5;

int accumulator : 15;

int instruction\_counter : 16;

};

int sc\_memoryInit();

int sc\_memorySet(int, int);

int sc\_memoryGet(int, int \*);

int sc\_memorySave(char \*);

int sc\_memoryLoad(char \*);

int sc\_commandEncode(int, int, int \*);

int sc\_commandDecode(int, int \*, int \*);

int sc\_printMemory(int);

void sc\_printRegFlag(void);

int sc\_regInit(void);

int sc\_regSet(int, int);

int sc\_regGet(int, int \*);

int sc\_accInit(void);

int sc\_accSet(int);

int sc\_accGet(int \*);

void sc\_printAccum(void);

int sc\_instrcInit(void);

int sc\_instrcSet(int);

int sc\_instrcGet(int \*);

void sc\_printInstrCount(void);

void sc\_printOper(void);

**myTerm.c:**

#include <stdio.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <sys/ioctl.h>

#include <fcntl.h>

#include "myTerm.h"

int mt\_clrscr(void)

{

printf("\E[H\E[2J\E[H");

return 0;

}

int mt\_gotoXY(int x, int y)

{

int rows = 0, cols = 0;

mt\_getscreensize(&rows, &cols);

if (y <= rows && y >= 0 && x >= 0 && x <= cols)

{

printf("\033[%d;%dH", y, x);

return 0;

}

else

{

return 1;

}

}

int mt\_getscreensize(int \*rows, int \*cols)

{

struct winsize ws;

if (!ioctl(1, TIOCGWINSZ, &ws))

{

(\*rows) = ws.ws\_row;

(\*cols) = ws.ws\_col;

return 0;

}

else

{

return -1;

}

}

int mt\_setfgcolor(enum colors COL)

{

if (COL == 0 || (COL <= 37 && COL >= 30))

{

printf("\033[0;%dm", COL);

return 0;

}

else

{

return -1;

}

}

int mt\_setbgcolor(enum colors COL)

{

if (COL == 0 || (COL <= 37 && COL >= 30))

{

int c = (int)COL + 10;

printf("\033[0;%dm", c);

return 0;

}

else

{

return -1;

}

}

**myTerm.h:**

#pragma once

enum colors

{

black = 30,

red = 31,

green = 32,

yellow = 33,

blue = 34,

magenta = 35,

cyan = 36,

white = 37,

reset = 0

};

int mt\_clrscr(void);

int mt\_gotoXY(int, int);

int mt\_getscreensize(int \*rows, int \*cols);

int mt\_setfgcolor(enum colors);

int mt\_setbgcolor(enum colors);

**myBigChars.c:**

#include "myBigChars.h"

int bc\_printA(char \*str)

{

printf("\E(0%s\E(B", str);

}

int bc\_box(int x1, int y1, int x2, int y2)

{

mt\_setbgcolor(black);

if (x1 == 0)

x1 = 1;

for (int CoordX = x1 + 1; CoordX < x2; CoordX++)

{

mt\_gotoXY(CoordX, y1);

bc\_printA("r");

}

for (int CoordY = y1 + 1; CoordY < y2; CoordY++)

{

mt\_gotoXY(x1, CoordY);

bc\_printA("x");

}

for (int CoordX = x1 + 1; CoordX < x2; CoordX++)

{

mt\_gotoXY(CoordX, y2);

bc\_printA("r");

}

for (int CoordY = y1 + 1; CoordY < y2; CoordY++)

{

mt\_gotoXY(x2, CoordY);

bc\_printA("x");

}

mt\_gotoXY(x1, y1);

bc\_printA("l");

mt\_gotoXY(x1, y2);

bc\_printA("m");

mt\_gotoXY(x2, y1);

bc\_printA("k");

mt\_gotoXY(x2, y2);

bc\_printA("j");

mt\_setbgcolor(black);

}

int bc\_drawInterfase(int paint)

{

//основная рамка

mt\_clrscr();

sc\_printMemory(paint);

int m = 0;

sc\_memoryGet(paint, &m);

bc\_drawdigit(m);

sc\_printRegFlag();

sc\_printAccum();

sc\_printOper();

sc\_printInstrCount();

bc\_box(1, 1, 61, 12);

bc\_box(62, 1, 83, 3);

bc\_box(62, 4, 83, 6);

bc\_box(62, 7, 83, 9);

bc\_box(62, 10, 83, 12);

bc\_box(1, 13, 42, 22);

bc\_box(43, 13, 83, 22);

mt\_gotoXY(27, 1);

printf(" Memory ");

mt\_gotoXY(67, 1);

printf(" Accumulator ");

mt\_gotoXY(63, 4);

printf(" InstructionCounter ");

mt\_gotoXY(68, 7);

printf(" Operation ");

mt\_gotoXY(70, 10);

printf(" Flags ");

mt\_gotoXY(51, 13);

printf(" Keys: ");

mt\_gotoXY(44, 14);

printf("l - load");

mt\_gotoXY(44, 15);

printf("s - save");

mt\_gotoXY(44, 16);

printf("r - run");

mt\_gotoXY(44, 17);

printf("t - step");

mt\_gotoXY(44, 18);

printf("i - reset");

mt\_gotoXY(44, 19);

printf("F5 - accumulator");

mt\_gotoXY(44, 20);

printf("F6 - instrictionComputer\n");

mt\_gotoXY(50, 24);

printf("\n");

mt\_gotoXY(1, 23);

printf("\nInput/Output:\n");

if (TerminalWrite[2] == 1)

{

bc\_printtextterminal();

}

}

int bc\_printbigchar(long int \*a, int x, int y, enum colors colDig, enum colors colBack)

{

if (x < 0 || y < 0)

{

return -1;

}

mt\_gotoXY(x, y);

for (int i = 0; i < 2; i++)

{

for (int l = 0; l < 32;)

{

for (int j = 0; j < 8 && l < 32; j++, l++)

{

if ((a[i] >> (l)) & 0x1)

{

mt\_setbgcolor(colBack);

bc\_printA(" ");

mt\_setfgcolor(reset);

}

else

{

mt\_setbgcolor(colDig);

bc\_printA(" ");

mt\_setbgcolor(reset);

}

}

y++;

mt\_gotoXY(x, y);

}

}

}

int bc\_setbigcharpos(int \*big, int x, int y, int value)

{

if ((y >= 0 && y < 8) && (x >= 0 && x < 8))

{

if (value == 1)

{

if (y < 4)

{

big[0] = big[0] | (1 << (8 \* y + x));

}

if (y > 3)

{

y -= 4;

big[1] = big[1] | (1 << (8 \* y + x));

}

}

else if (value == 0)

{

if (y < 4)

{

big[0] = (~(1 << (8 \* y + x))) & big[0];

}

if (y > 3)

{

y -= 4;

big[1] = (~(1 << (8 \* y + x))) & big[1];

}

}

}

else

{

return -1;

}

}

int bc\_getbigcharpos(int \*big, int x, int y, int \*value)

{

if ((y >= 0 && y < 8) && (x >= 0 && x < 8))

{

if (y < 4)

{

(\*value) = (big[0] >> (y \* 8 + x)) & 0x1;

}

if (y > 3)

{

y -= 4;

(\*value) = (big[1] >> (y \* 8 + x)) & 0x1;

}

}

else

{

return -1;

}

}

int bc\_bigcharwrite(int fd, long int \*big, int count)

{

int result;

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

{

if ((result = write(fd, big, sizeof(long int) \* 2)) == -1)

return -1;

}

}

int bc\_bigcharread(int fd, long int \*big, int need\_count, int \*count)

{

for (int i = 0; i < need\_count; i++, \*count++)

{

int result;

if ((result = read(fd, big, sizeof(long int) \* 2)) == -1)

return -1;

}

}

void bc\_setvaluebigchar(char s, long int \*mas)

{

struct myBigChars BCH = {

{1382171196, 1010976330},

{269752336, 940576784},

{270804024, 1006896136},

{940580924, 943998016},

{1041372184, 940576784},

{1077675068, 943998016},

{67388536, 943998012},

{135274556, 67372036},

{1010975292, 1010975298},

{2084717116, 1010843712},

{4279769112, 404232447},

{1009001496, 606348324},

{1007952924, 1011106884},

{33694744, 405012994},

{1145316380, 472138820},

{470025276, 1006896132},

{470025276, 67372036},

{4278190080, 255}};

switch (s)

{

case '0':

mas[0] = BCH.Zero[0];

mas[1] = BCH.Zero[1];

break;

case '1':

mas[0] = BCH.One[0];

mas[1] = BCH.One[1];

break;

case '2':

mas[0] = BCH.Two[0];

mas[1] = BCH.Two[1];

break;

case '3':

mas[0] = BCH.Three[0];

mas[1] = BCH.Three[1];

break;

case '4':

mas[0] = BCH.Four[0];

mas[1] = BCH.Four[1];

break;

case '5':

mas[0] = BCH.Five[0];

mas[1] = BCH.Five[1];

break;

case '6':

mas[0] = BCH.Six[0];

mas[1] = BCH.Six[1];

break;

case '7':

mas[0] = BCH.Seven[0];

mas[1] = BCH.Seven[1];

break;

case '8':

mas[0] = BCH.Eight[0];

mas[1] = BCH.Eight[1];

break;

case '9':

mas[0] = BCH.Nine[0];

mas[1] = BCH.Nine[1];

break;

case 'A':

mas[0] = BCH.A[0];

mas[1] = BCH.A[1];

break;

case 'B':

mas[0] = BCH.B[0];

mas[1] = BCH.B[1];

break;

case 'C':

mas[0] = BCH.C[0];

mas[1] = BCH.C[1];

break;

case 'D':

mas[0] = BCH.D[0];

mas[1] = BCH.D[1];

break;

case 'E':

mas[0] = BCH.E[0];

mas[1] = BCH.E[1];

break;

case 'F':

mas[0] = BCH.F[0];

mas[1] = BCH.F[1];

break;

case '+':

mas[0] = BCH.Plus[0];

mas[1] = BCH.Plus[1];

break;

case ' ':

mas[0] = 0;

mas[1] = 0;

break;

case '-':

mas[0] = BCH.Minus[0];

mas[1] = BCH.Minus[1];

break;

default:

break;

}

}

void bc\_drawdigit(int digit)

{

char str[4] = {0};

long int BigDigit[2] = {0};

int bit = digit & (1 << 14);

enum colors fon = black;

if (!bit)

{

bc\_setvaluebigchar('+', BigDigit);

bc\_printbigchar(BigDigit, 2, 14, fon, green);

int oper = 0, com = 0;

sc\_commandDecode(digit, &com, &oper);

sprintf(str, "%02X", com);

for (int i = 0, x = 10; i < 2; i++, x += 8)

{

bc\_setvaluebigchar(str[i], BigDigit);

bc\_printbigchar(BigDigit, x, 14, fon, green);

}

sprintf(str, "%02X", oper);

for (int i = 0, x = 26; i < 2; i++, x += 8)

{

bc\_setvaluebigchar(str[i], BigDigit);

bc\_printbigchar(BigDigit, x, 14, fon, green);

}

}

else

{

if (digit < 0)

{

digit = digit \* (-1);

bc\_setvaluebigchar('-', BigDigit);

bc\_printbigchar(BigDigit, 2, 14, fon, green);

}

else

{

digit = digit & 0x3FFF;

bc\_setvaluebigchar(' ', BigDigit);

bc\_printbigchar(BigDigit, 2, 14, fon, green);

}

snprintf(str, 15, "%04X", digit);

for (int i = 0, x = 10; i < 4; i++, x += 8)

{

bc\_setvaluebigchar(str[i], BigDigit);

bc\_printbigchar(BigDigit, x, 14, fon, green);

}

}

}

void bc\_settextterminal(int digit, int number)

{

TerminalWrite[0] = number;

TerminalWrite[1] = digit;

TerminalWrite[2] = 1;

}

void bc\_printtextterminal(void)

{

if (TerminalWrite[1] >= 0)

{

printf("%02d> +%04X\n", TerminalWrite[0], TerminalWrite[1]);

}

else

{

printf("%02d> -%04X\n", TerminalWrite[0], (-1) \* TerminalWrite[1]);

}

}

**myBigChars.h:**

#pragma once

#include "myTerm.h"

#include "mySimpleComputer.h"

#include <unistd.h>

#include <stdio.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

struct myBigChars

{

long int Zero[2]; // = {1382171196, 1010976330};

long int One[2]; // = {269752336, 940576784};

long int Two[2]; // = {270804024, 1006896136};

long int Three[2]; // = {940580924, 943998016};

long int Four[2]; // = {1041372184, 940576784};

long int Five[2]; // = {1077675068, 943998016};

long int Six[2]; // = {67388536, 943998012};

long int Seven[2]; // = {135274556, 67372036};

long int Eight[2]; // = {1010975292, 1010975298};

long int Nine[2]; // = {2084717116, 1010843712};

long int Plus[2]; // = {4279769112, 404232447};

long int A[2]; // = {1009001496, 606348324};

long int B[2]; // = {1007952924, 1011106884};

long int C[2]; // = {33694744, 405012994};

long int D[2]; // = {1145316380, 472138820};

long int E[2]; // = {470025276, 1006896132};

long int F[2]; // = {470025276, 67372036};

long int Minus[2]; // = {4278190080,255};

};

static int TerminalWrite[3];

int bc\_printA(char \*str);

int bc\_box(int x1, int y1, int x2, int y2);

int bc\_printbigchar(long int \*, 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, long int \*big, int count);

int bc\_bigcharread(int fd, long int \*big, int need\_count, int \*count);

int bc\_drawInterfase(int);

void bc\_setvaluebigchar(char, long int \*);

void bc\_drawdigit(int digit);

void bc\_settextterminal(int, int);

void bc\_printtextterminal(void);

**myReadKey.c:**

#include "myReadKey.h"

#include <unistd.h>

#include <stdio.h>

#include <string.h>

struct termios savetty;

int rk\_readkey(enum keys \*Key)

{

char buffer[8] = {0};

//Дескриптор терминала STDIN\_FILENO 0

read(STDIN\_FILENO, buffer, 8);

if (buffer[0] == 'l')

{

(\*Key) = KeyLoad;

}

else if (buffer[0] == 's')

{

(\*Key) = KeySave;

}

else if (buffer[0] == 'r')

{

(\*Key) = KeyRun;

}

else if (buffer[0] == 't')

{

(\*Key) = KeyStep;

}

else if (buffer[0] == 'i')

{

(\*Key) = KeyReset;

}

else if (buffer[0] == '\033')

{

if (buffer[2] == '1' && buffer[3] == '5')

{

(\*Key) = KeyAccumulator;

}

else if (buffer[2] == '1' && buffer[3] == '7')

{

(\*Key) = KeyInstructionCounter;

}

else if (buffer[2] == 'C')

{

(\*Key) = KeyRight;

}

else if (buffer[2] == 'D')

{

(\*Key) = KeyLeft;

}

else if (buffer[2] == 'A')

{

(\*Key) = KeyUp;

}

else if (buffer[2] == 'B')

{

(\*Key) = KeyDown;

}

else

(\*Key) = KeyQuit;

}

else if (buffer[0] == '.')

{

(\*Key) = KeySetCellValue;

}

return 0;

}

int rk\_mytermsave(void)

{

struct termios tty;

if (tcgetattr(STDIN\_FILENO, &tty) == -1)

{

return 1;

}

else

{

savetty = tty;

return 0;

}

}

int rk\_mytermrestore(void)

{

if (tcsetattr(STDIN\_FILENO, TCSANOW, &savetty) == -1)

{

return 1;

}

else

{

return 0;

}

}

int rk\_mytermregime(int regime, int vtime, int vmin, int echo, int sigint)

{

struct termios Settings;

rk\_mytermsave();

Settings = savetty;

if (regime == 1)

{

Settings.c\_lflag &= (~ICANON);

if (sigint == 1)

{

Settings.c\_lflag &= (~ISIG);

}

else if (sigint == 0)

{

Settings.c\_lflag |= ISIG;

}

if (echo == 1)

{

Settings.c\_lflag &= (~ECHO);

}

else if (echo == 0)

{

Settings.c\_lflag |= ECHO;

}

Settings.c\_cc[VMIN] = vmin;

Settings.c\_cc[VTIME] = vtime;

}

else if (regime == 0)

{

Settings.c\_lflag |= (ICANON | ECHO);

}

tcsetattr(0, TCSANOW, &Settings);

return 0;

}

**myReadKey.h:**

#pragma once

#include <fcntl.h>

#include <termios.h>

enum keys

{

KeyLoad, //l

KeySave, //s

KeyRun, //r

KeyStep, //t

KeyReset, //i

KeyAccumulator, //f5

KeyInstructionCounter, //f6

KeyQuit, //q

KeyRight, //->

KeyLeft, //<-

KeyUp, //^

KeyDown, // \/

KeySetCellValue, //.

Default

};

int rk\_readkey(enum keys \*);

int rk\_mytermsave(void);

int rk\_mytermrestore(void);

int rk\_mytermregime(int regime, int vtime, int vmin, int echo, int sigint);

**Signal.c:**

#include "Signal.h"

void s\_signalhandler(int signal)

{

switch (signal)

{

case SIGUSR1:

alarm(0);

sc\_memoryInit();

sc\_regInit();

sc\_accInit();

sc\_instrcInit();

sc\_regSet(IGNORING\_TACT\_PULSES, 1);

break;

case SIGALRM:

{

CU();

int instr = 0, flag = 0;

sc\_instrcGet(&instr);

sc\_regGet(IGNORING\_TACT\_PULSES, &flag);

if (!flag)

{

alarm(1);

instr++;

sc\_instrcSet(instr);

}

}

}

}

**Signal.h:**

#pragma once

#include <signal.h>

#include "mySimpleComputer.h"

#include "myTerm.h"

#include <sys/time.h>

#include <unistd.h>

#include "CPU.h"

void s\_signalhandler(int);

void s\_starttimer(void);

**CPU.c:**

#include "CPU.h"

bool flag\_add = false;

int ALU(int command, int operand)

{

int value\_cell = 0, value\_acc = 0;

sc\_accGet(&value\_acc);

sc\_memoryGet(operand, &value\_cell);

if (value\_cell > 0x0000)

value\_cell = value\_cell & (0x3FFF);

if (value\_acc > 0x0000)

value\_acc = value\_acc & (0x3FFF);

switch (command)

{

case 0x30:

{ //ADD

value\_acc += value\_cell;

if (value\_acc <= 0x3FFF && value\_acc >= -0x3FFF)

{

sc\_accSet(value\_acc);

flag\_add=true;

}

else

{

sc\_regSet(MEMORY\_OVERFLOW, 1);

return 1;

}

break;

}

case 0x31:

{ //SUB

value\_acc -= value\_cell;

if (value\_acc <= 0x3FFF && value\_acc >= -0x3FFF)

{

sc\_accSet(value\_acc);

}

else

{

sc\_regSet(MEMORY\_OVERFLOW, 1);

return 1;

}

break;

}

case 0x32:

{ //DIVIDE

if (value\_cell == 0)

{

sc\_regSet(DIVISION\_ERROR\_BY\_0, 1);

return 1;

}

else

{

value\_acc /= value\_cell;

sc\_accSet(value\_acc);

}

break;

}

case 0x33:

{ //MUL

value\_acc \*= value\_cell;

if (value\_acc <= 0x3FFF && value\_acc >= -0x3FFF)

{

sc\_accSet(value\_acc);

}

else

{

sc\_regSet(MEMORY\_OVERFLOW, 1);

return 1;

}

break;

}

case 0x60:{

//CHL Dima

value\_acc=value\_cell<<1;

if (value\_acc <= 0x3FFF && value\_acc >= -0x3FFF)

{

sc\_accSet(value\_acc);

}

else

{

sc\_regSet(MEMORY\_OVERFLOW, 1);

return 1;

}

break;

}

case 0x61:{

//SHL Vova

value\_acc=value\_cell>>1;

if (value\_acc <= 0x3FFF && value\_acc >= -0x3FFF)

{

sc\_accSet(value\_acc);

}

else

{

sc\_regSet(MEMORY\_OVERFLOW, 1);

return 1;

}

break;

}

default:

break;

}

}

int check16(char \*number)

{

for (int i = 1; i < 5; i++)

{

if (number[i] != '0' && number[i] != '1' && number[i] != '2' && number[i] != '3' &&

number[i] != '4' && number[i] != '5' && number[i] != '6' && number[i] != '7' &&

number[i] != '8' && number[i] != '9' && number[i] != 'A' && number[i] != 'B' &&

number[i] != 'C' && number[i] != 'D' && number[i] != 'E' && number[i] != 'F')

{

return 1;

}

}

return 0;

}

int read\_value(int operand)

{

rk\_mytermregime(0, 0, 1, 0, 1);

int instr\_c = 0;

sc\_instrcGet(&instr\_c);

bc\_drawInterfase(instr\_c);

printf("%02d< ", operand);

char value[10];

fgets(value, 10, stdin);

int val\_copy = 0;

value[strlen(value) - 1] = '\0';

if (strlen(value) == 5 && (value[0] == '+' || value[0] == '-'))

{

sscanf(value + 1, "%04X", &val\_copy);

if (val\_copy <= 0x3FFF && val\_copy >= -0x3FFF && check16(value) != 1)

{

if (value[0] == '-')

{

val\_copy \*= (-1);

if (val\_copy == 0)

{

val\_copy = val\_copy | (1 << 14);

}

}

else

{

val\_copy = val\_copy | (1 << 14);

}

if (sc\_memorySet(operand, val\_copy) == 1)

{

printf("\nBorder overload");

sc\_regSet(IGNORING\_TACT\_PULSES, 1);

rk\_mytermregime(1, 1, 0, 1, 0);

return 0;

}

}

else

{

printf("\nError memory overflow or not number");

getchar();

return 1;

}

}

else

{

printf("\nFew or many arguments");

getchar();

return 1;

}

rk\_mytermregime(1, 1, 0, 1, 0);

return 0;

}

int CU(void)

{

int command = 0, operand = 0, value\_cell = 0, adress\_cell = 0;

sc\_instrcGet(&adress\_cell);

if (sc\_memoryGet(adress\_cell, &value\_cell) == 1)

{

return 1;

}

if (sc\_commandDecode(value\_cell, &command, &operand) == 1)

{

sc\_regSet(CODE\_ERROR, 1);

sc\_regSet(IGNORING\_TACT\_PULSES, 1);

alarm(0);

return 1;

}

else

{

sc\_regSet(CODE\_ERROR, 0);

}

if (command >= 0x30 && command <= 0x33||command == 0x60 || command == 0x61)

{

if (ALU(command, operand))

{

sc\_regSet(IGNORING\_TACT\_PULSES, 1);

return 1;

}

else

{

return 0;

}

}

int value = 0;

switch (command)

{

case 0x10:

{ //READ

int read = 0;

do

{

read = read\_value(operand);

} while (read);

break;

}

case 0x11:

{ //WRITE

sc\_memoryGet(operand, &value);

if (value > 0)

{

value = value & 0x3FFF;

}

bc\_settextterminal(value, operand);

break;

}

case 0x20:

{ //LOAD

sc\_memoryGet(operand, &value);

if (value > 0)

{

value = value & 0x3FFF;

}

sc\_accSet(value);

break;

}

case 0x21:

{ //STORE

sc\_accGet(&value);

if (value >= 0)

{

value = value | (1 << 14);

}

sc\_memorySet(operand, value);

break;

}

case 0x40:

{ //JUMP

sc\_instrcSet(operand - 1);

break;

}

case 0x41:

{ //JNEG

sc\_accGet(&value);

if (value < 0x0000)

sc\_instrcSet(operand - 1);

break;

}

case 0x42:

{ //JZ

sc\_accGet(&value);

if (value == 0)

sc\_instrcSet(operand - 1);

break;

}

case 0x55:

{

//JNS Maks

sc\_accGet(&value);

if (value > 0x0000)

sc\_instrcSet(operand - 1);

break;

}

case 0x43:

{ //HALT

sc\_regSet(IGNORING\_TACT\_PULSES, 1);

break;

}

case 0x58:{

//JP Vitaliy

sc\_accGet(&value);

if(value%2 == 0)

sc\_instrcSet(operand - 1);

break;

}

case 0x57:{

//JC Nastya

if(flag\_add){

flag\_add = false;

sc\_instrcSet(operand - 1);

}

break;

}

default:

{

sc\_regSet(CODE\_ERROR, 1);

sc\_regSet(IGNORING\_TACT\_PULSES, 1);

}

break;

}

}

**CPU.h:**

#pragma once

#include "mySimpleComputer.h"

#include <sys/time.h>

#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stdbool.h>

#include "myReadKey.h"

#include "myBigChars.h"

int ALU(int command, int operand);

int read\_value(int operand);

int CU(void);

int check16(char \*number);

**sat.c:**

#include "sat.h"

int chek\_arg(int argc, char \*argv[])

#include "sat.h"

int chek\_arg(int argc, char \*argv[])

{

if (argc != 3)

{

return INVALID\_NUMBER\_OF\_ARGUMENTS;

}

else

{

size\_t obj = strlen(argv[1]);

size\_t asem = strlen(argv[2]);

if (obj < 3 || asem < 4)

{

return INVALID\_SIZE\_OF\_ARGUMENTS;

}

else

{

char \*expansion\_obj = strrchr(argv[1], '.');

char \*expansion\_asm = strrchr(argv[2], '.');

if (!strcmp(expansion\_asm, ".sa") && !strcmp(expansion\_obj, ".o"))

{

FILE \*fp = fopen(argv[2], "r");

if (fp == NULL)

{

fclose(fp);

return FILE\_NOT\_FOUND;

}

else

{

fclose(fp);

return SUCCESSFULLY;

}

}

else

{

return WRONG\_EXTENSION;

}

}

}

}

void succes(char str[])

{

mt\_setfgcolor(green);

printf("%s", str);

mt\_setfgcolor(reset);

printf("\n");

}

void error\_pr(char str[])

{

mt\_setfgcolor(red);

printf("%s", str);

mt\_setfgcolor(reset);

printf("\n");

exit(EXIT\_FAILURE);

}

int error\_handler(int error)

{

switch (error)

{

case SUCCESSFULLY:

{

succes("FILE FOUND");

return 0;

break;

}

case INVALID\_NUMBER\_OF\_ARGUMENTS:

{

error\_pr("INVALID NUMBER OF ARGUMENTS");

break;

}

case INVALID\_SIZE\_OF\_ARGUMENTS:

{

error\_pr("INVALID SIZE OF ARGUMENTS");

break;

}

case WRONG\_EXTENSION:

{

error\_pr("WRONG EXTENSION");

break;

}

case FILE\_NOT\_FOUND:

{

error\_pr("ASSEMBLER FILE NOT FOUND");

break;

}

case EMPTY\_FILE:

{

error\_pr("EMPTY FILE");

break;

}

case COMMAND\_NOT\_FOUND:

{

error\_pr("COMMAND NOT FOUND");

break;

}

case ERROR\_ENCODE:

{

error\_pr("ERROR ENCODE");

break;

}

case INCORRECT\_NUMBER\_FORMAT:

{

error\_pr("INCORRECT NUMBER FORMAT");

break;

}

case TOO\_FEW\_COMMANDS:

{

error\_pr("TOO FEW COMMANS");

break;

}

case DIGIT\_TOO\_BIG:

{

error\_pr("DIGIT TOO BIG");

break;

}

default:

{

return 1;

break;

}

}

}

int ComCPU(char \*command)

{

if (strcmp(command, "=") == 0)

{

return 0x12;

}

else if (strcmp(command, "READ") == 0)

{

return 0x10;

}

else if (strcmp(command, "WRITE") == 0)

{

return 0x11;

}

else if (strcmp(command, "LOAD") == 0)

{

return 0x20;

}

else if (strcmp(command, "STORE") == 0)

{

return 0x21;

}

else if (strcmp(command, "ADD") == 0)

{

return 0x30;

}

else if (strcmp(command, "SUB") == 0)

{

return 0x31;

}

else if (strcmp(command, "DIVIDE") == 0)

{

return 0x32;

}

else if (strcmp(command, "MUL") == 0)

{

return 0x33;

}

else if (strcmp(command, "JUMP") == 0)

{

return 0x40;

}

else if (strcmp(command, "JNEG") == 0)

{

return 0x41;

}

else if (strcmp(command, "JZ") == 0)

{

return 0x42;

}

else if (strcmp(command, "HALT") == 0)

{

return 0x43;

}

else if (strcmp(command, "JP") == 0)

{

return 0x58;

}

else if (strcmp(command, "JNS") == 0)

{

return 0x55;

}

else if (strcmp(command, "JNC") == 0)

{

return 0x57;

}

else if (strcmp(command, "JP") == 0)

{

return 0x58;

}

else if (strcmp(command, "JNS") == 0)

{

return 0x55;

}

else if (strcmp(command, "CHL") == 0)

{

return 0x60;

}

else if (strcmp(command, "SHR") == 0)

{

return 0x61;

}

else

error\_handler(COMMAND\_NOT\_FOUND);

}

void EncodeLine(int adress\_cell, int command, int operand)

{

int cell = 0;

if (command != 0x12)

{

if (sc\_commandEncode(command, operand, &cell) == 1)

{

error\_handler(ERROR\_ENCODE);

}

else

{

RAM[adress\_cell] = cell;

}

}

else

{

if (operand <= 0x3FFF && operand >= -0x3FFF)

{

if (operand > 0)

{

operand = operand | (1 << 14);

}

RAM[adress\_cell] = operand;

}

else

{

error\_handler(DIGIT\_TOO\_BIG);

}

}

}

int asm\_parser(char \*line)

{

int adress\_cell = 0, operand = 0;

char com[10], operand\_str[10];

sscanf(line, "%d%s%s", &adress\_cell, com, operand\_str);

if (strlen(operand\_str) == 0)

{

error\_handler(TOO\_FEW\_COMMANDS);

}

if (operand\_str[0] == '+' || operand\_str[0] == '-')

{

if (strlen(operand\_str) == 5)

{

//operand\_str[0] = ' ';

sscanf(operand\_str + 1, "%4X", &operand);

if (operand\_str[0] == '-')

{

operand \*= (-1);

}

}

else

{

error\_handler(INCORRECT\_NUMBER\_FORMAT);

}

}

else

{

sscanf(operand\_str, "%d", &operand);

}

int dig\_com = ComCPU(com);

EncodeLine(adress\_cell, dig\_com, operand);

}

void interpretation(char \*asembler, char \*object)

{

FILE \*asem\_file;

char str[256], command[256];

asem\_file = fopen(asembler, "rt");

if (asem\_file == NULL)

{

mt\_setfgcolor(red);

printf("COULD NOT OPEN OR CREATE FILE");

mt\_setfgcolor(reset);

printf("\n");

fclose(asem\_file);

exit(EXIT\_FAILURE);

}

else

{

succes("ASSEMBLER FILE OPEN");

}

int number\_str = 0;

while (fgets(str, sizeof(char) \* 256, asem\_file))

{

int count\_empty = 0;

for (int i = 0; i < 256 && str[i] != '\0' && str[i] != ';' && str[i] != '\n'; i++)

{

if (str[i] == ' ')

{

count\_empty++;

}

command[i] = str[i];

command[i + 1] = '\0';

}

if (strtok(command, "\t") != NULL && count\_empty != strlen(command))

{

printf("%s\t", command);

asm\_parser(command);

mt\_setfgcolor(green);

printf("+");

mt\_setfgcolor(reset);

printf("\n");

number\_str++;

}

}

if (number\_str == 0)

{

error\_handler(EMPTY\_FILE);

}

else

{

printf("Number of lines: %d\n", number\_str);

}

fclose(asem\_file);

}

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

{

FILE \*obj\_file = NULL;

char name\_file\_o[256];

strcpy(name\_file\_o, "../");

int warning = chek\_arg(argc, argv);

if (!error\_handler(warning))

{

interpretation(argv[2], argv[1]);

strcat(name\_file\_o, argv[1]);

obj\_file = fopen(name\_file\_o, "w");

if (obj\_file == NULL)

{

error\_pr("COULD NOT OPEN OR CREATE FILE");

}

fwrite(RAM, 1, 100 \* sizeof(int), obj\_file);

fclose(obj\_file);

}

else

{

exit(EXIT\_FAILURE);

}

}

**sat.h:**

#pragma once

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stddef.h>

#include "mySimpleComputer.h"

#include "myTerm.h"

#define SUCCESSFULLY 0

#define INVALID\_NUMBER\_OF\_ARGUMENTS 1

#define INVALID\_SIZE\_OF\_ARGUMENTS 2

#define WRONG\_EXTENSION 3

#define FILE\_NOT\_FOUND 4

#define FILE\_OPEN 5

#define COULD\_NOT\_OPEN\_OR\_CREATE\_FILE 6

#define EMPTY\_FILE 7

#define COMMAND\_NOT\_FOUND 8

#define ERROR\_ENCODE 9

#define INCORRECT\_NUMBER\_FORMAT 10

#define TOO\_FEW\_COMMANDS 11

#define DIGIT\_TOO\_BIG 12

int RAM[100];

struct command\_system

{

int operand : 16;

int com : 7;

};

**basic.c:**

#include "basic.h"

struct asm\_line asm\_code[100];

int RAM[100] = {0};

int perem[26] = {0};

int perem\_adress = 98;

int number\_line = 0;

int RAM\_adress = 0;

int \*buf\_obm = NULL;

int end\_programm = 0;

void error\_print(char text[])

{

mt\_setfgcolor(red);

printf("%s\n", text);

mt\_setfgcolor(reset);

exit(1);

}

void succes(char \*text)

{

mt\_setfgcolor(green);

printf("%s\n", text);

mt\_setfgcolor(reset);

}

void error\_handler(int error)

{

switch (error)

{

case TOO\_FEW\_ARGUMENTS:

{

error\_print("TOO FEW ARGUMENTS");

break;

}

case INVALID\_SIZE\_OF\_ARGUMENTS:

{

error\_print("INVALID SIZE OF NAME FILES\n");

break;

}

case FILE\_NOT\_FOUND:

{

error\_print("FILE NOT FOUND\n");

break;

}

case SUCCESSFULLY:

{

succes("SUCCESSFULLY\n");

break;

}

case WRONG\_EXTENSION:

{

error\_print("WRONG EXTENSION\n");

break;

}

case FILE\_IS\_FOUND:

{

succes("FILE IS FOUND\n");

break;

}

case FILE\_IS\_NOT\_OPEN:

{

error\_print("FILE IS NOT OPEN\n");

break;

}

case MEMORY\_ERROR:

{

error\_print("COILD NOT ALLOCATE MEMORY\n");

break;

}

case WRONG\_LINE\_ORDER:

{

error\_print("WRONG LINE ORDER\n");

break;

}

case MISSING\_COMMAND\_OR\_PARAMETERS:

{

error\_print("MISSING COMMAND OR PARAMETERS\n");

break;

}

case WRONG\_REGISTER:

{

error\_print("WRONG\_REGISTER\n");

break;

}

case COMMAND\_NOT\_FOUND:

{

error\_print("COMMAND\_NOT\_FOUND\n");

break;

}

case MEMORY\_OVERLOAD:

{

error\_print("MEMORY OVERLOAD\n");

break;

}

case LINE\_NOT\_FOUND:

{

error\_print("LINE NOT FOUND\n");

break;

}

case TOO\_LONG\_VARIABLE:

{

error\_print("TOO LONG VARIABLE\n");

break;

}

case END\_ERROR:

{

error\_print("'END' SHOULD BE USED ONCE");

break;

}

}

}

int check\_name\_file(int argc, char \*argv[])

{

if (argc == 3)

{

size\_t asem = strlen(argv[1]);

size\_t bas = strlen(argv[2]);

if (bas < 5 || asem < 4)

{

error\_handler(INVALID\_SIZE\_OF\_ARGUMENTS);

}

else

{

char \*expansion\_asm = strrchr(argv[1], '.');

char \*expansion\_bas = strrchr(argv[2], '.');

if (!strcmp(expansion\_asm, ".sa") && !strcmp(expansion\_bas, ".bas"))

{

FILE \*fp = fopen(argv[2], "r");

if (fp == NULL)

{

error\_handler(FILE\_NOT\_FOUND);

}

else

{

fclose(fp);

error\_handler(FILE\_IS\_FOUND);

}

}

else

{

error\_handler(WRONG\_EXTENSION);

}

}

}

else

{

error\_handler(TOO\_FEW\_ARGUMENTS);

}

}

int empty\_str(const char \*str)

{

int counter = 0;

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

{

if (str[i] == ' ' || str[i] == '\t' || str[i] == '\n')

{

counter++;

}

}

if (strlen(str) == counter)

{

return 1;

}

else

{

return 0;

}

}

int amount\_line\_in\_file(char \*bas\_name)

{

FILE \*file\_bas = NULL;

file\_bas = fopen(bas\_name, "rt");

char str[256];

if (file\_bas == NULL)

{

error\_handler(FILE\_IS\_NOT\_OPEN);

}

int amount = 0;

while (fgets(str, sizeof(char) \* 256, file\_bas))

{

if (!empty\_str(str))

{

amount++;

}

}

fclose(file\_bas);

return amount;

}

char \*get\_parameters(char \*a)

{

int i = 0, j = 0, space = 0, flag = 0, stop = 2;

while (a[i] != '\n' && a[i] != '\0')

{

if (space < stop)

{

if (a[i] == ' ' || a[i] == '\t')

flag = 1;

else if (flag == 1)

{

space++;

flag = 0;

}

}

if (space == stop)

{

return (&a[i]);

}

++i;

}

return NULL;

}

int check\_str\_syntax(char \*command, char \*parameters)

{

if (strlen(command) == 0 || strlen(parameters) == 0 && strcmp(command, "END") != 0)

{

error\_handler(MISSING\_COMMAND\_OR\_PARAMETERS);

}

for (int i = 0; i < strlen(command); i++)

{

if (command[i] < 'A' || command[i] > 'Z')

{

error\_handler(WRONG\_REGISTER);

}

}

}

void check\_syntax(struct basic\_line \*basic\_code, size\_t n)

{

int prev\_number = 0;

for (size\_t i = 0; i < n; i++)

{

if (i > 0)

{

if (basic\_code[i].number\_line <= basic\_code[i - 1].number\_line)

{

error\_handler(WRONG\_LINE\_ORDER);

}

}

check\_str\_syntax(basic\_code[i].command, basic\_code[i].parameters);

}

}

int check\_command(char \*command)

{

if (strcmp(command, "INPUT") == 0)

{

return 0x0014;

}

else if (strcmp(command, "PRINT") == 0)

{

return 0x0015;

}

else if (strcmp(command, "GOTO") == 0)

{

return 0x0016;

}

else if (strcmp(command, "IF") == 0)

{

return 0x0017;

}

else if (strcmp(command, "LET") == 0)

{

return 0x0018;

}

else if (strcmp(command, "REM") == 0)

{

return 0x0019;

}

else if (strcmp(command, "END") == 0)

{

return 0x001A;

}

else

{

error\_handler(COMMAND\_NOT\_FOUND);

}

}

int is\_digit(char \*dig)

{

int i = 0;

if (dig[0] == '-')

{

i++;

}

for (; i < strlen(dig); i++)

{

if (dig[i] < '0' || dig[i] > '9')

{

return -1;

}

}

return 0;

}

int \*add\_perem(char Litter, bool digit)

{

int pos = Litter - 'A';

if (pos >= 0 && pos < 26 && Litter <= 'Z' && Litter >= 'A' && !digit)

{

if (perem[pos] == 0)

{

perem[pos] = perem\_adress;

RAM[perem\_adress] = perem\_adress;

perem\_adress--;

}

return &perem[pos];

}

else

{

if (digit)

{

RAM[perem\_adress] = perem\_adress;

int copy = perem\_adress;

perem\_adress--;

return &RAM[copy];

}

}

return NULL;

}

int add\_command(char command\_asm[], int \*operand, struct basic\_line \*basic\_l, int number\_c, int value, bool str\_b)

{

asm\_code[RAM\_adress].number\_cell = number\_c;

strcpy(asm\_code[RAM\_adress].command, command\_asm);

asm\_code[RAM\_adress].operand = operand;

if (value >= -0x3FFF && value <= 0x3FFF)

{

asm\_code[RAM\_adress].value = value;

}

else

{

exit(1);

}

if (str\_b)

{

(\*basic\_l).operand = number\_line;

}

RAM\_adress++;

if (command\_asm[0] != '=')

number\_line++;

}

int check\_sing(char sing)

{

if (sing == '\*' || sing == '/' || sing == '+' || sing == '-')

{

return 1;

}

else

{

return 0;

}

}

int lever\_sign(int \*operand, char sign, struct basic\_line \*basic\_line, int number, bool flag\_obm)

{

switch (sign)

{

case '+':

add\_command("ADD", operand, &basic\_line[number], number\_line, 0, false);

break;

case '-':

if (flag\_obm)

{

add\_command("STORE", NULL, &basic\_line[number], number\_line, 99, false);

add\_command("LOAD", operand, &basic\_line[number], number\_line, 0, false);

add\_command("SUB", NULL, &basic\_line[number], number\_line, 99, false);

}

else

{

add\_command("SUB", operand, &basic\_line[number], number\_line, 0, false);

}

break;

case '/':

if (flag\_obm)

{

add\_command("STORE", NULL, &basic\_line[number], number\_line, 99, false);

add\_command("LOAD", operand, &basic\_line[number], number\_line, 0, false);

add\_command("DIVIDE", NULL, &basic\_line[number], number\_line, 99, false);

}

else

{

add\_command("DIVIDE", operand, &basic\_line[number], number\_line, 0, false);

}

break;

case '\*':

add\_command("MUL", operand, &basic\_line[number], number\_line, 0, false);

break;

default:

break;

}

}

void overload\_mem()

{

if (RAM\_adress + (100 - perem\_adress) > 100)

{

error\_handler(MEMORY\_ERROR);

}

}

int add\_asm\_line(char command[], char parameters[], struct basic\_line \*basic\_line, const int number, const size\_t n, bool fl\_b)

{

int code\_com = check\_command(command);

char A[20], Sign[20], B[20];

int \*adress\_d;

int adr = 0;

char com\_if[256] = {0}, param\_if[256] = {0}, let\_line[256] = {0}, get\_st[256] = {0}, p1[256] = {0}, p2[256] = {0}, sign;

struct stack \*let = NULL, \*calculate = NULL;

switch (code\_com)

{

case 0x0014:

{

/\* INPUT \*/

if (parameters[0] >= 'A' && parameters[0] <= 'Z')

{

add\_command("READ", add\_perem(parameters[0], false), &basic\_line[number], number\_line, 0, fl\_b);

}

else

{

error\_handler(WRONG\_REGISTER);

}

break;

}

case 0x0015:

{

/\* PRINT \*/

if (parameters[0] >= 'A' && parameters[0] <= 'Z')

{

add\_command("WRITE", add\_perem(parameters[0], false), &basic\_line[number], number\_line, 0, fl\_b);

}

else

{

if ((parameters[0] >= '0' && parameters[0] <= '9' ||

(parameters[0] == '-' && parameters[1] >= '1' && parameters[1] <= '9')) &&

is\_digit(parameters) == 0)

{

adress\_d = add\_perem(' ', true);

add\_command("=", NULL, &basic\_line[number], \*adress\_d, atoi(parameters), false);

add\_command("WRITE", adress\_d, &basic\_line[number], number\_line, 0, fl\_b);

}

else

{

error\_handler(WRONG\_REGISTER);

}

}

break;

}

case 0x0016:

{

/\* GOTO \*/

adr = atoi(parameters);

bool flag = false;

for (int j = 0; j < n; j++)

{

if (basic\_line[j].number\_line == adr)

{

add\_command("JUMP", &basic\_line[j].operand, &basic\_line[number], number\_line, 0, fl\_b);

flag = true;

break;

}

}

if (!flag)

{

error\_handler(LINE\_NOT\_FOUND);

}

break;

}

case 0x0017:

{

/\* IF \*/

sscanf(parameters, "%s %s %s", A, Sign, B);

int begin\_if = -1;

if (is\_digit(A) == 0)

{

adress\_d = add\_perem(' ', true);

add\_command("=", NULL, &basic\_line[number], \*adress\_d, atoi(A), fl\_b);

if (fl\_b)

begin\_if = \*adress\_d;

add\_command("LOAD", adress\_d, &basic\_line[number], number\_line, 0, false);

}

else

{

if (strlen(A) == 1)

{

adress\_d = add\_perem(A[0], false);

add\_command("LOAD", adress\_d, &basic\_line[number], number\_line, 0, false);

if (fl\_b)

begin\_if = number\_line - 1; //!!!

}

else

{

error\_handler(TOO\_LONG\_VARIABLE);

}

}

if (is\_digit(B) == 0)

{

adress\_d = add\_perem(' ', true);

add\_command("=", NULL, &basic\_line[number], \*adress\_d, atoi(B), false);

add\_command("SUB", adress\_d, &basic\_line[number], number\_line, 0, false);

}

else

{

if (strlen(B) == 1)

{

adress\_d = add\_perem(B[0], false);

add\_command("SUB", adress\_d, &basic\_line[number], number\_line, 0, false);

}

else

{

error\_handler(TOO\_LONG\_VARIABLE);

}

}

if (strlen(Sign) > 1)

{

exit(1);

}

switch (Sign[0])

{

case '>':

{

add\_command("JNEG", &basic\_line[number + 1].operand, basic\_line, number\_line, 0, false);

add\_command("JZ", &basic\_line[number + 1].operand, basic\_line, number\_line, 0, false);

break;

}

case '<':

{

add\_command("JNS", &basic\_line[number + 1].operand, basic\_line, number\_line, 0, false);

add\_command("JZ", &basic\_line[number + 1].operand, basic\_line, number\_line, 0, false);

break;

}

case '=':

{

add\_command("JNEG", &basic\_line[number + 1].operand, basic\_line, number\_line, 0, false);

add\_command("JNS", &basic\_line[number + 1].operand, basic\_line, number\_line, 0, false);

break;

}

default:

exit(1);

break;

}

if\_pars(com\_if, param\_if, parameters);

int number\_com = check\_command(com\_if);

add\_asm\_line(com\_if, param\_if, basic\_line, number, n, false);

if (fl\_b)

basic\_line[number].operand = begin\_if;

break;

}

case 0x0018:

/\* LET \*/

sscanf(parameters, "%s %s", A, B);

let\_merge(parameters, let\_line);

buf\_obm = add\_perem(parameters[0], false);

mt\_setfgcolor(red);

if (calcpars(let\_line) == 1)

{

mt\_setfgcolor(reset);

exit(1);

}

mt\_setfgcolor(reset);

conversion\_rpn(parameters, &let);

int size\_st = size\_stack(let);

int \*store\_adr1 = NULL, \*store\_adr2 = NULL, \*store\_adr = NULL, \*adr\_acc = NULL, begin\_let = -1;

bool flag\_begin = false, flag\_acc = false;

if (size\_st == 1)

{

pop\_del(&let, get\_st, &store\_adr);

basic\_line[number].operand = number\_line;

add\_command("=", NULL, &basic\_line[number], \*buf\_obm, atoi(get\_st), false);

break;

}

while (!pop\_del(&let, get\_st, &store\_adr))

{

if (!check\_sing(get\_st[0]))

{

push(get\_st, &calculate, NULL);

}

else

{

sign = get\_st[0];

pop\_del(&calculate, p1, &store\_adr1);

pop\_del(&calculate, p2, &store\_adr2);

int \*adr\_p1 = NULL, \*adr\_p2 = NULL;

if ('A' <= p1[0] && p1[0] <= 'Z')

{

adr\_p1 = add\_perem(p1[0], false);

}

else

{

if ('0' <= p1[0] && p1[0] <= '9')

{

adr\_p1 = add\_perem(' ', true);

add\_command("=", NULL, &basic\_line[number], \*adr\_p1, atoi(p1), false);

if (!flag\_begin)

begin\_let = \*adr\_p1;

}

else

{

if (strcmp(p1, "acc") == 0 && store\_adr1 != NULL)

{

adr\_p1 = store\_adr1;

}

}

}

if ('A' <= p2[0] && p2[0] <= 'Z')

{

adr\_p2 = add\_perem(p2[0], false);

}

else

{

if ('0' <= p2[0] && p2[0] <= '9')

{

adr\_p2 = add\_perem(' ', true);

add\_command("=", NULL, &basic\_line[number], \*adr\_p2, atoi(p2), false);

if (!flag\_begin)

begin\_let = \*adr\_p2;

}

else

{

if (strcmp(p2, "acc") == 0 && store\_adr2 != NULL)

{

adr\_p2 = store\_adr2;

}

}

}

if ('0' <= p2[0] && p2[0] <= '9' && '0' <= p1[0] && p1[0] <= '9')

{

if (adr\_acc != NULL)

{

add\_command("STORE", adr\_acc, &basic\_line[number], number\_line, 0, false);

}

}

if (strcmp(p1, "acc") != 0 && strcmp(p2, "acc") != 0)

{

if (!flag\_acc)

{

adr\_acc = add\_perem(' ', true);

flag\_acc = true;

add\_command("LOAD", adr\_p2, &basic\_line[number], number\_line, 0, false);

if (!flag\_begin)

begin\_let = number\_line;

lever\_sign(adr\_p1, sign, basic\_line, number, false);

push("acc", &calculate, adr\_acc);

}

else

{

add\_command("STORE", adr\_acc, &basic\_line[number], number\_line, 0, false);

add\_command("LOAD", adr\_p2, &basic\_line[number], number\_line, 0, false);

adr\_acc = add\_perem(' ', true);

push("acc", &calculate, adr\_acc);

lever\_sign(adr\_p1, sign, basic\_line, number, false);

flag\_acc = false;

}

}

else

{

if (strcmp(p1, "acc") == 0)

{

lever\_sign(adr\_p2, sign, basic\_line, number, true);

push("acc", &calculate, store\_adr1);

}

else

{

lever\_sign(adr\_p1, sign, basic\_line, number, false);

push("acc", &calculate, store\_adr1);

}

}

}

}

add\_command("STORE", buf\_obm, &basic\_line[number], number\_line, 0, false);

if (fl\_b)

basic\_line[number].operand = begin\_let - 1;

break;

case 0x0019:

/\* REM \*/

break;

case 0x001A:

/\* END \*/

add\_command("HALT", NULL, &basic\_line[number], number\_line, 0, fl\_b);

if (end\_programm == 0)

{

end\_programm++;

}

else

{

error\_handler(END\_ERROR);

}

break;

default:

error\_handler(COMMAND\_NOT\_FOUND);

break;

}

overload\_mem();

}

void parser(struct basic\_line \*basic\_line, size\_t n)

{

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

{

add\_asm\_line(basic\_line[i].command, basic\_line[i].parameters, basic\_line, i, n, true);

}

}

void dragging\_out\_line(char \*name\_file\_asm, char \*name\_file\_bas)

{

FILE \*file\_bas = NULL;

char str[256];

int amount = amount\_line\_in\_file(name\_file\_bas);

printf("Lines : %d\n\n", amount);

struct basic\_line \*basic\_code = (struct basic\_line \*)calloc(amount + 1, sizeof(struct basic\_line));

if (basic\_code == NULL)

{

error\_handler(MEMORY\_ERROR);

}

file\_bas = fopen(name\_file\_bas, "rt");

if (file\_bas == NULL)

{

error\_handler(FILE\_IS\_NOT\_OPEN);

}

int index = 0;

while (fgets(str, sizeof(char) \* 256, file\_bas))

{

char com[15] = {' '}, \*param = NULL;

int n = 0;

if (!empty\_str(str))

{

printf("%s", str);

sscanf(str, "%d%s", &basic\_code[index].number\_line, basic\_code[index].command);

param = get\_parameters(str);

if (param != NULL)

{

int i = 0;

for (i = 0; i < strlen(param); i++)

{

basic\_code[index].parameters[i] = param[i];

}

basic\_code[index].parameters[i - 1] = '\0';

basic\_code[index].operand = -1;

}

else

{

basic\_code[index].parameters[0] = '\0';

}

index++;

}

}

fclose(file\_bas);

check\_syntax(basic\_code, amount);

parser(basic\_code, amount);

}

int fill\_file(char name[])

{

char name\_file\_sa[256];

strcpy(name\_file\_sa,"../Translator(asembler)/");

strcat(name\_file\_sa,name);

FILE \*as\_file = NULL;

as\_file = fopen(name\_file\_sa, "w");

if (as\_file == NULL)

{

return 1;

}

for (int i = 0; i < 100 && asm\_code[i].command[0] != 0; i++)

{

if (asm\_code[i].operand != NULL)

{

fprintf(as\_file, "%02d %s %02d\n", asm\_code[i].number\_cell, asm\_code[i].command, \*asm\_code[i].operand);

}

else

{

if (asm\_code[i].command[0] == 'H')

{

fprintf(as\_file, "%02d %s %02d\n", asm\_code[i].number\_cell, asm\_code[i].command, asm\_code[i].value);

}

else

{

if (asm\_code[i].value < 0)

{

fprintf(as\_file, "%02d %s -%04X\n", asm\_code[i].number\_cell, asm\_code[i].command, (-1) \* asm\_code[i].value);

}

else

{

fprintf(as\_file, "%02d %s +%04X\n", asm\_code[i].number\_cell, asm\_code[i].command, asm\_code[i].value);

}

}

}

}

fclose(as\_file);

return 0;

}

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

{

check\_name\_file(argc, argv);

dragging\_out\_line(argv[1], argv[2]);

fill\_file(argv[1]);

error\_handler(SUCCESSFULLY);

}

**basic.h:**

#include <stdio.h>

#include <stdlib.h>

//#include <stddef.h>

#include <string.h>

#include "myTerm.h"

#include <stdbool.h>

#include "parse\_line.h"

#include "RPN.h"

#define TOO\_FEW\_ARGUMENTS 0

#define INVALID\_SIZE\_OF\_ARGUMENTS 1

#define FILE\_NOT\_FOUND 2

#define SUCCESSFULLY 3

#define WRONG\_EXTENSION 4

#define FILE\_IS\_FOUND 5

#define FILE\_IS\_NOT\_OPEN 6

#define MEMORY\_ERROR 7

#define WRONG\_LINE\_ORDER 8

#define MISSING\_COMMAND\_OR\_PARAMETERS 9

#define WRONG\_REGISTER 10

#define COMMAND\_NOT\_FOUND 11

#define MEMORY\_OVERLOAD 12

#define LINE\_NOT\_FOUND 13

#define TOO\_LONG\_VARIABLE 14

#define END\_ERROR 15

#define INPUT 0x0014

#define PRINT 0x0015

#define GOTO 0x0016

#define IF 0x0017

#define LET 0x0018

#define REM 0x0019

#define END 0x001A

struct basic\_line

{

int number\_line;

char command[256];

char parameters[256];

int operand;

};

struct asm\_line

{

int number\_cell;

char command[256];

int \*operand;

int value;

};

struct equally{

int operand;

int value;

};

**parse\_line.c:**

#include "parse\_line.h"

int if\_pars(char \*com, char \*par, char \*a)

{

// com - куда запишем вытянутую команду, par - параметры, a - строка с

// параметрами IF

int i = -1, j = 0, space = 0, flag = 0, if\_uslovie = 0, stop = 1;

do

{

++i;

if (a[i] != ' ' && a[i] != '\t')

{

++if\_uslovie;

}

} while (if\_uslovie != 3); //пока не считается три параметра условия

if (a[i] >= '0' && a[i] <= '9' || a[i] == '-')

{

while (a[i] != ' ' && a[i] != '\t')

{

++i;

}

} //если последний параметр - число, то идем до его конца

if (a[i] >= 'A' && a[i] <= 'Z')

{

++i;

}

char add[256] = {0};

for (i; i < strlen(a); ++i, ++j)

{

add[j] = a[i];

} //все оставшееся(add) - то, из чего будем вытаскивать команду и ее параметры

sscanf(add, "%s", com); // вытащили команду(com)

flag = 0;

i = 0;

if (add[0] == ' ' || add[0] == '\t')

stop++;

while (stop != 0)

{

if ((add[i] == ' ' || add[i] == '\t') && flag == 0)

{

stop--;

flag = 1;

}

else if (add[i] != ' ' && add[i] != '\t')

{

flag = 0;

}

++i;

} //идем до конца команды(для этого нам нужно пройти stop пропусков, т.е.

//каждый пропуск stop--)

for (i, j = 0; i < strlen(add); ++i)

{

// if (add[i] != ' ' && add[i] != '\t') {

par[j] = add[i];

++j;

// }

} //переписываем параметры команды БЕЗ ПРОБЕЛОВ в другой массив(par)

return 0;

}

int calcpars(char \*a)

{

if (a[0] < 'A' || a[0] > 'Z')

{

printf("NOT A LETTER\n");

return 1;

//ОШИБКА: НЕ БУКВА

}

if (a[1] != '=')

{

printf("NO '=' OR TOO LONG VARIABLE \n");

return 1;

//ОШИБКА: ОТСУТСТВУЕТ ЗНАК РАВЕНСТВА ИЛИ СЛИШКОМ ДЛИННАЯ ПРИСВАИВАЕМАЯ

//ПЕРЕМЕННАЯ

}

if (a[2] == '+' || a[2] == '\*' || a[2] == '/' || a[2] == ')' ||

a[strlen(a) - 1] == '+' || a[strlen(a) - 1] == '-' ||

a[strlen(a) - 1] == '\*' || a[strlen(a) - 1] == '/' ||

a[strlen(a) - 1] == '(')

{

printf("YOU CANT START OR END IT WITH THIS SIGN\n");

return 1;

//ОШИБКА: ПЕРВЫЙ ИЛИ ПОСЛЕДНИЙ СИМВОЛ НЕ МОЖЕТ ЯВЛЯТЬСЯ ЗНАКОМ ИЛИ

//НЕСООТВЕТСТВУЮЩЕЙ СКОБКОЙ

}

int flag = 0,

p\_flag = 0; // flag для счета количества разрядов в числе(чтобы выкинуть

// ошибку в случае, если оно больше допустимого), p\_flag для

// счета открытых скобок - если после цикла она не равна 0,

// то значит, что где-то не закрыта скобка

for (int i = 2; i < strlen(a); i++)

{

if ((a[i] == '+' || a[i] == '-' || a[i] == '\*' || a[i] == '/' ||

a[i] == '(') &&

(a[i + 1] == '+' || a[i + 1] == '\*' || a[i + 1] == '/' ||

a[i + 1] == ')'))

{

printf("TWO SIGNS IN A ROW\n");

return 1;

//ОШИБКА: подряд идущие знаки

}

if (((a[i] >= 'A' && a[i] <= 'Z') &&

(a[i + 1] >= '0' && a[i + 1] <= '9')) ||

((a[i + 1] >= 'A' && a[i + 1] <= 'Z') &&

(a[i] >= '0' && a[i] <= '9')) ||

((a[i + 1] >= 'A' && a[i + 1] <= 'Z') &&

(a[i] >= 'A' && a[i] <= 'Z')) ||

(a[i] == ')' && (a[i + 1] >= '0' && a[i + 1] <= '9')) ||

(a[i] == '(' && (a[i] >= '0' && a[i] <= '9')))

{

printf("TWO NON-COMPLIMENT SIGNS IN A ROW OR TOO LONG VARIABLE\n");

return 1;

//ОШИБКА: подряд идущие переменная и число или слишком длинная

//переменная

}

if (a[i] >= 'A' && a[i] <= 'Z')

{

}

else if (a[i] >= '0' && a[i] <= '9')

{

flag++;

if (flag > 5)

{

printf("NOT AVAILABLE VALUE\n");

return 1;

//ОШИБКА: НЕДОПУСТИМОЕ ЗНАЧЕНИЕ

}

else if (flag == 5)

{

int check = (a[i - 4] - '0') \* 10000 + (a[i - 3] - '0') \* 1000 +

(a[i - 2] - '0') \* 100 + (a[i - 1] - '0') \* 10 +

(a[i] - '0');

if (check > 16383)

{

printf("NOT AVAILABLE VALUE\n");

return 1;

//ОШИБКА: НЕДОПУСТИМОЕ ЗНАЧЕНИЕ

}

}

}

else if (a[i] == '+' || a[i] == '-' || a[i] == '/' || a[i] == '\*')

{

flag = 0;

}

else if (a[i] == ')' || a[i] == '(')

{

if (a[i] == '(')

{

p\_flag++;

}

else if (a[i] == ')' && p\_flag != 0)

{

flag = 0;

p\_flag--;

}

else

{

printf("NOT ENOUGH SCOBKI\n");

return 1;

//ОШИБКА: не закрыты скобки

}

}

else

{

printf("UNKNOWN SYMBOl\n");

return 1;

//ОШИБКА: используется неизвестный символ

}

}

if (p\_flag != 0)

{

printf("NOT ENOUGH PARENTHESES\n");

return 1;

//ОШИБКА: не закрыты скобки

}

return 0;

}

int let\_merge(const char \*from\_let\_line, char \*to\_let\_line)

{

for (int i = 0, j = 0; i < strlen(from\_let\_line); i++)

{

if (from\_let\_line[i] != ' ' && from\_let\_line[i] != '\n' && from\_let\_line[i] != '\t')

{

to\_let\_line[j] = from\_let\_line[i];

j++;

}

}

}

**parse\_line.h:**

#pragma once

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

int if\_pars(char \*com, char \*par, char \*a);

int calcpars(char \*a);

int let\_merge(const char \*from\_let\_line, char \*to\_let\_line);

**RPN.c:**

#include "RPN.h"

int priority(char Sign)

{

switch (Sign)

{

case '+':

return 1;

case '-':

return 1;

case '\*':

return 2;

case '/':

return 2;

case '(':

return 0;

case ')':

return 0;

default:

return 0;

}

}

int conversion\_rpn(const char \*fromcalculate, struct stack \*\*let)

{

struct stack \*SingStack = NULL;

struct stack \*Calculate = NULL;

int \*store = NULL;

char copy[256][256];

int j = 0;

for (int i = 0; i < strlen(fromcalculate); j++, i++)

{

int k = 0;

while (fromcalculate[i] != ' ' && fromcalculate[i] != '\t' && i < 255)

{

if (j > 1)

{

copy[j - 2][k] = fromcalculate[i];

}

i++;

k++;

copy[j - 2][k] = '\0';

}

}

bool flag = false;

for (int i = 0; i < j - 2; i++)

{

if ((copy[i][0] >= 'A' && copy[i][0] <= 'Z') || (copy[i][0] >= '0' && copy[i][0] <= '9'))

{

push(copy[i], &Calculate, NULL);

}

else

{

if (copy[i][0] == '\*' || copy[i][0] == '/' || copy[i][0] == '+' || copy[i][0] == '-')

{

char get\_st[256] = {0};

pop\_get(SingStack, get\_st);

if (strlen(get\_st) == 1)

{

int pr1 = priority(get\_st[0]);

int pr2 = priority(copy[i][0]);

if (pr1 == pr2)

{

push(get\_st, &Calculate, NULL);

pop\_del(&SingStack, get\_st, &store);

}

push(copy[i], &SingStack, NULL);

}

else

{

push(copy[i], &SingStack, NULL);

}

}

else

{

if (copy[i][0] == '(')

{

push(copy[i], &SingStack, NULL);

}

else

{

if (copy[i][0] == ')')

{

char get\_st[256] = {0};

do

{

if (strlen(get\_st) == 1)

{

push(get\_st, &Calculate, NULL);

}

pop\_del(&SingStack, get\_st, &store);

} while (get\_st[0] != '(');

}

}

}

}

}

char get\_st[256] = {0};

while (!pop\_del(&SingStack, get\_st, &store))

{

push(get\_st, &Calculate, NULL);

}

del(let);

while (!pop\_del(&Calculate, get\_st, &store))

{

push(get\_st, let, NULL);

}

}

int push(char data[], struct stack \*\*head\_stack, int \*operand)

{

struct stack \*Node = (struct stack \*)malloc(sizeof(struct stack));

if (Node == NULL)

{

return 1;

}

Node->prev = \*head\_stack;

Node->store\_acc = operand;

strcpy(Node->Litter, data);

\*head\_stack = Node;

return 0;

}

int pop\_del(struct stack \*\*head\_stack, char ret[], int \*\*store)

{

struct stack \*Node = NULL;

if (\*head\_stack != NULL)

{

Node = \*head\_stack;

strcpy(ret, (\*head\_stack)->Litter);

\*store = (\*head\_stack)->store\_acc;

(\*head\_stack) = (\*head\_stack)->prev;

free(Node);

return 0;

}

else

{

return 1;

}

}

int pop\_get(struct stack \*head\_stack, char ret[])

{

if (head\_stack != NULL)

{

strcpy(ret, head\_stack->Litter);

return 0;

}

else

{

return 1;

}

}

void print\_stack(const struct stack \*head)

{

while (head)

{

printf("%s ", head->Litter);

head = head->prev;

}

}

int del(struct stack \*\*head\_stack)

{

if (\*head\_stack != NULL)

{

while (\*head\_stack != NULL)

{

struct stack \*Node;

Node = \*head\_stack;

\*head\_stack = (\*head\_stack)->prev;

free(Node);

}

return 0;

}

return 1;

}

int check\_empty\_stack(struct stack \*calculate)

{

if (calculate == NULL)

{

return 1;

}

else

return 0;

}

int size\_stack(struct stack \*head\_stack)

{

int size = 0;

while (head\_stack)

{

size++;

head\_stack = head\_stack->prev;

}

return size;

}

**RPN.h:**

#pragma once

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stdbool.h>

struct stack

{

char Litter[256];

int operand;

struct stack \*prev;

int \*store\_acc;

};

int conversion\_rpn(const char \*fromcalculate, struct stack \*\*head\_stack);

int priority(char Sign);

int push(char data[], struct stack \*\*head\_stack, int \*operand);

int pop\_del(struct stack \*\*head\_stack, char ret[], int \*\*store);

int pop\_get(struct stack \*head\_stack, char ret[]);

void print\_stack(const struct stack \*head);

int del(struct stack \*\*head\_stack);

void fill\_operand(struct stack \*\*head\_stack);

int check\_empty\_stack(struct stack \*calculate);

int size\_stack(struct stack \*head\_stack);