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

Кафедра ВС

**КУРСОВАЯ РАБОТА**

По дисциплине «Архитектура ЭВМ»

Вариант 9

Выполнил:

студент гр. ИС-641

Дьячков Д.А.

Проверил:

Майданов Ю. С.

Новосибирск 2018

**Оглавление**

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

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

Результат работы программы...............................................................................11

Заключение…………………................................................................................13

Литература………………….................................................................................14

**Постановка задачи**

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

**Обработка команд центральным процессором**

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

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

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

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

2. полученное значение декодируется как команда;

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

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

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

6. Работа функции завершается.

**Транслятор с языка Simple Assembler**

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

Пример программы на **Simple Assembler:**

00 READ 09 ; (Ввод А)

01 READ 10 ; (Ввод В)

02 LOAD 09 ; (Загрузка А в аккумулятор)

03 SUB 10 ; (Отнять В)

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

05 WRITE 09 ; (Вывод А)

06 HALT 00 ; (Останов)

07 WRITE 10 ; (Вывод В)

08 HALT 00 ; (Останов)

09 = +0000 ; (Переменная А)

10 = +9999 ; (Переменная В)

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

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

**Транслятор с языка Simple Basic**

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

Пример программы на **Simple Basic:**

10 REM Это комментарий

20 INPUT A

30 INPUT B

40 LET C = A – B

50 IF C < 0 GOTO 20

60 PRINT C

70 END

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

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

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

· оперативную память;

· внешние устройства;

· центральный процессор.

·

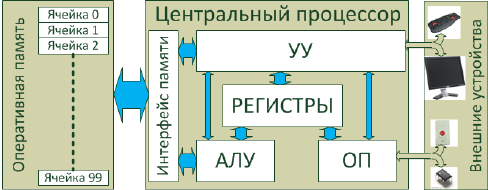


Рисунок 1 – Архитектура вычислительной машины Simple Computer

**Оперативная память**

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

**Внешние устройства**

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

**Центральный процессор**

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

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

· арифметико-логическое устройство (АЛУ);

· управляющее устройство (УУ);

· обработчик прерываний от внешних устройств (ОП);

· интерфейс доступа к оперативной памяти.

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

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

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

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

**Блок схемы используемых алгоритмов**

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

Считываем значение ячейки из памяти

Декодируем в машинную команду и операнд

Нет

Команда и операнд корректны?

Нет

Команда является арифметической операцией?

Вызов ALU для арифметической операции

Выполнение команды

**2.Simple Assembler**

Файл с кодом на Simple **Assembler**

Если данные считались

Нет

Да

Сканируем строку и разбиваем на токены

**3.Simple Basic**

Нет

Сохраняем полученный байт код в выходной файл

Да

Данные корректны?

Кодируем в машинный код

Файл с кодом на Simple **Basic, массив переменных и нерешенных переходов**

Нет

Если данные считались

Сканируем строку и разбиваем на токены

Нет

Данные корректны?

Выбираем блок преобразования в зависимости от команды

Есди в команде есть GOTO с переходом вперёд, то добавляем в список **нерешенных переходов**.

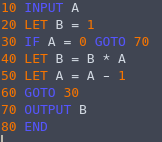
Корректируем нерешенные переходы

Сохраняем полученный код Simple Assembler в файл

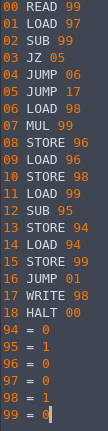
**Результаты проведенного исследования**

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.

factorial.bas:

****

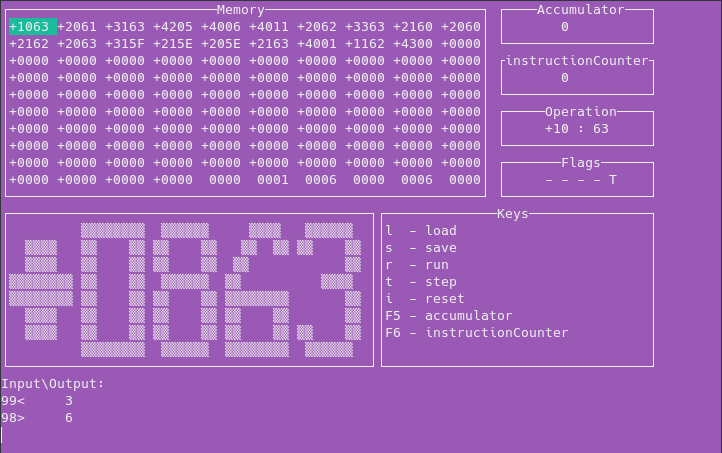
factorial.asm:



Программа загруженная в SimpleComputer:



Выполнение программы вычисления факториала, для числа 3.



**Заключение**

В рамках курсовой работы была реализованы основные команды УУ и АЛУ, так же реализован алгоритм работы центрального процессора. В том числе были реализованы трансляторы: транслируеющий код SimpleAssembler в бинарный формат, для выполнения на SimpleCompiter, и транслятор с высокоуровневого языка SimpleBasic в код SimpleAssembler.

**Список используемых источников**

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

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

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

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

**Листинг программы**

Репозиторй курсовой работы: https://github.com/DenisDyachkov/SimplePC

**cpu.c:**

#include "cpu.h"

#include "interface.h"

#include <mySimpleComputer.h>

#include <signal.h>

#include <stdio.h>

#include <string.h>

enum eCmdLIST {

READ = 0x10,

WRITE = 0x11,

LOAD = 0x20,

STORE = 0x21,

ADD = 0x30,

SUB = 0x31,

DIV = 0x32,

MUL = 0x33,

JMP = 0x40,

JNE = 0x41,

JZ = 0x42,

HALT = 0x43,

JNC = 0x57,

JNP = 0x59

};

typedef int(\*instr\_callback)(int);

struct stInstructionInfo {

const char\* name;

instr\_callback function;

enum eCmdLIST code;

};

int \_\_read(int operator);

int \_\_write(int operator);

int \_\_load(int operator);

int \_\_store(int operator);

int \_\_add(int operator);

int \_\_sub(int operator);

int \_\_div(int operator);

int \_\_mul(int operator);

int \_\_jmp(int operator);

int \_\_jne(int operator);

int \_\_jz(int operator);

int \_\_halt(int operator);

int \_\_jnc(int operator);

int \_\_jnp(int operator);

static struct stInstructionInfo

cmds[] = {

{"READ", \_\_read, READ},

{"WRITE", \_\_write, WRITE},

{"LOAD", \_\_load, LOAD},

{"STORE", \_\_store, STORE},

{"ADD", \_\_add, ADD},

{"SUB", \_\_sub, SUB},

{"DIV", \_\_div, DIV},

{"MUL", \_\_mul, MUL},

{"JUMP", \_\_jmp, JMP},

{"JNEG", \_\_jne, JNE},

{"JZ", \_\_jz, JZ},

{"HALT", \_\_halt, HALT},

{"JNC", \_\_jnc, JNC},

{"JNP", \_\_jnp, JNP}

};

struct stInstructionInfo\* instruct\_search(int code) {

int r = sizeof(cmds) / sizeof(cmds[0]), l = 0;

while (l < r) {

int m = (r - l) / 2 + l;

if (cmds[m].code == code)

return cmds + m;

else if (cmds[m].code > code)

r = m;

else

l = m + 1;

}

return NULL;

}

int cmd\_search(const char\* cmd) {

int size = sizeof(cmds) / sizeof(cmds[0]), i = 0;

for (; i < size; ++i)

if (!strcmp(cmds[i].name, cmd))

return cmds[i].code;

return -1;

}

int is\_cmd\_exist(int code) {

return instruct\_search(code) != NULL;

}

int is\_cmd\_arithmetic(int code) {

return (code == ADD || code == SUB || code == DIV || code == MUL);

}

int ALU(int code, int operand) {

struct stInstructionInfo\* instruction = instruct\_search(code);

if (!instruction) {

sc\_regSet(FLAG\_INVALID\_COMMAND, 1);

return -1;

}

return instruction->function(operand);

}

int CU() {

int data, code, addr;

if (sc\_memoryGet(registers.instruction\_counter, &data) ||

sc\_commandDecode(data, &code, &addr)) {

sc\_regSet(FLAG\_IGNORE\_CLOCK, 1);

return -1;

}

if (is\_cmd\_exist(code) == 0) {

sc\_regSet(FLAG\_INVALID\_COMMAND, 1);

sc\_regSet(FLAG\_IGNORE\_CLOCK, 1);

return -1;

}

if (addr < 0 || addr >= RAM\_SIZE) {

sc\_regSet(FLAG\_OUT\_RANGE, 1);

sc\_regSet(FLAG\_IGNORE\_CLOCK, 1);

return -1;

}

++registers.instruction\_counter;

if (is\_cmd\_arithmetic(code)) {

if (ALU(code, addr)) {

sc\_regSet(FLAG\_IGNORE\_CLOCK, 1);

return -1;

}

} else

instruct\_search(code)->function(addr);

select\_cell = registers.instruction\_counter;

return 0;

}

int sign\_number\_from\_memory(int value) {

if (value & 0x2000)// negative

value = -1 \* (((~value) & 0x3FFF) + 1);

return value;

}

int sign\_number\_to\_memory(int value) {

if (value < 0)//(value & (~0x3FFF)) == 0x3FFF

value = 0x2000 | (value & 0x3FFF);

return value;

}

int \_\_read(int operator) {

int result;

read\_console\_value(operator, &result);

result = sign\_number\_to\_memory(result);

if (result & (~0x3FFF)) {

sc\_regSet(FLAG\_OVERFLOW, 1);

result &= 0x3FFF;

}

sc\_memorySet(operator, result | 0x4000);

return 0;

}

int \_\_write(int operator) {

int result;

sc\_memoryGet(operator, &result);

write\_console\_value(operator, sign\_number\_from\_memory(result & 0x3FFF));

return 0;

}

int \_\_load(int operator) {

sc\_memoryGet(operator, (int\*)&registers);//accumulator (0 offset)

registers.accumulator = sign\_number\_from\_memory(registers.accumulator & 0x3FFF);

return 0;

}

int \_\_store(int operator) {

int value = sign\_number\_to\_memory(registers.accumulator) | 0x4000;

sc\_memorySet(operator, value);

return 0;

}

void accum\_overflow\_fix() {

if (registers.accumulator > 0 &&

registers.accumulator & (~0x3FFF)) {

sc\_regSet(FLAG\_OVERFLOW, 1);

registers.accumulator &= 0x3FFF;

}

}

int \_\_add(int operator) {

int right;

sc\_memoryGet(operator, &right);

registers.accumulator += sign\_number\_from\_memory(right & 0x3FFF);

accum\_overflow\_fix();

return 0;

}

int \_\_sub(int operator) {

int right;

sc\_memoryGet(operator, &right);

registers.accumulator -= sign\_number\_from\_memory(right & 0x3FFF);

accum\_overflow\_fix();

return 0;

}

int \_\_div(int operator) {

int right;

sc\_memoryGet(operator, &right);

if (right == 0) {

sc\_regSet(FLAG\_DIV\_ZERO, 1);

return -1;

}

registers.accumulator /= sign\_number\_from\_memory(right & 0x3FFF);

return 0;

}

int \_\_mul(int operator) {

int right;

sc\_memoryGet(operator, &right);

registers.accumulator \*= sign\_number\_from\_memory(right & 0x3FFF);

accum\_overflow\_fix();

return 0;

}

int \_\_jmp(int operator) {

registers.instruction\_counter = operator;

return 0;

}

int \_\_jne(int operator) {

if (registers.accumulator < 0)//TODO: старший бит (15) = 1 -> отрицательное

registers.instruction\_counter = operator;

return 0;

}

int \_\_jz(int operator) {

if (registers.accumulator == 0)

registers.instruction\_counter = operator;

return 0;

}

int \_\_halt(int operator) {

sc\_regSet(FLAG\_IGNORE\_CLOCK, 1);

registers.instruction\_counter = 0;

return 0;

}

int \_\_jnc(int operator) {

int of, instruction, cmd, addr;

sc\_regGet(FLAG\_OVERFLOW, &of);

sc\_memoryGet(registers.instruction\_counter - 1, &instruction);

sc\_commandDecode(instruction, &cmd, &addr);

if (cmd == ADD && of)

registers.instruction\_counter = operator;

//if (registers.accumulator & (~0x3FFF))

// registers.instruction\_counter = operator;

return 0;

}

int \_\_jnp(int operator) {

if (registers.accumulator & 1)

registers.instruction\_counter = operator;

return 0;

}

**Asm.c:**

#include "asm.h"

#include <mySimpleComputer.h>

#include <stdio.h>

#include <string.h>

#include <cpu.h>

int asm\_to\_object(const char\* filename\_asm, const char\* filename\_object) {

FILE \*fasm = fopen(filename\_asm, "r");

//FILE \*fobj = fopen(filename".o", "wb");

if (!fasm)// || !fobj)

return 1;

int address, operand, code, encode, ignore;

char cmd[16] = { 0 };

while (fscanf(fasm,"%d %[=a-zA-Z]", &address, cmd) != 0) {//"%d %s %d"

if (cmd[0] != '=') {

fscanf(fasm, "%d", &operand);

code = cmd\_search(cmd);

if (code == -1 || sc\_commandEncode(code, operand, &encode))

return 1;

} else {

//char plus;

//fscanf(fasm, "%c%x", &plus, &operand);

fscanf(fasm, "%x", &encode);

encode = 0x4000 | (encode & 0x3FFF);

}

//fwrite(&encode, sizeof(encode), 1, fobj);

sc\_memorySet(address, encode);

do { ignore = fgetc(fasm); }

while (ignore != '\n' && ignore != EOF);

if (ignore == EOF)

break;

}

fclose(fasm);

if (cmd[0] == 0)

return 1;

//fclose(fobj);

/\*char object\_file[strlen(filename)];

strcpy(object\_file, filename);

object\_file[strlen(filename) - 3] = 0;

strcat(object\_file, "o");\*/

sc\_memorySave(filename\_object);

return 0;

}

**Rpn.c:**

#include "rpn.h"

struct stack\_t \*stack\_push(struct stack\_t \*head, char a) {

struct stack\_t \*ptr;

if((ptr = malloc(sizeof(struct stack\_t))) == NULL) {

perror("Out of memory\n");

exit(1);

}

ptr->c = a;

ptr->next = head;

return ptr;

}

char stack\_pop(struct stack\_t \*\*head) {

struct stack\_t \*ptr;

char a;

if(\*head == NULL)

return '\0';

ptr = \*head;

a = ptr->c;

\*head = ptr->next;

free(ptr);

return a;

}

int get\_prior(char c) {

switch(c) {

case '\*':

case '/':

return 3;

case '-':

case '+':

return 2;

case '(':

return 1;

default:break;

}

}

void translate\_to\_rpn(char \*outstr, char \*a) {

struct stack\_t \*opers = NULL;

int k, point;

k = 0;

point = 0;

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

if(a[k] == ')') {

while((opers->c) != '(')

outstr[point++]=stack\_pop(&opers);

stack\_pop(&opers);

}

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

outstr[point++] = a[k];

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

outstr[point++] = a[k];

else if(a[k] == '(')

opers = stack\_push(opers, '(');

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

if(opers == NULL)

opers = stack\_push(opers, a[k]);

else

if(get\_prior(opers->c) < get\_prior(a[k]))

opers = stack\_push(opers, a[k]);

else {

while((opers != NULL) && (get\_prior(opers->c) >= get\_prior(a[k])))

outstr[point++] = stack\_pop(&opers);

opers = stack\_push(opers, a[k]);

}

}

k++;

}

while(opers != NULL)

outstr[point++] = stack\_pop(&opers);

outstr[point] = '\0';

}

**Basic.c:**

#include "basic.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#include "rpn.h"

struct stLines {

unsigned line\_number;

unsigned start\_address;

};

struct stVariables {

unsigned address;

int init\_value;

char name;

};

struct stGotoConflict {

unsigned instratuction\_address;

unsigned goto\_line;

char calc;

};

int command\_type(const char \*cmd) {

if (!strcmp(cmd, "REM"))

return 1;

if (!strcmp(cmd, "INPUT"))

return 2;

if (!strcmp(cmd, "OUTPUT"))

return 3;

if (!strcmp(cmd, "GOTO"))

return 4;

if (!strcmp(cmd, "IF"))

return 5;

if (!strcmp(cmd, "LET"))

return 6;

if (!strcmp(cmd, "END"))

return 7;

return 0;

}

unsigned variable\_id(const struct stVariables \*vars, unsigned max, char var) {

unsigned id = 0;

for (; id < max; ++id)

if (vars[id].name == var)

return id;

return max;

}

#define getVarID(toID, varName) \

if (isdigit(varName)) { \

toID = var\_id; \

var[toID].name = tmp\_var; \

var[toID].address = 99 - toID; \

var[toID].init\_value = atoi(&varName); \

++var\_id; \

++tmp\_var; \

} else { \

toID = variable\_id(var, var\_id, varName); \

if (toID == var\_id) { \

var[toID].name = varName; \

var[toID].address = 99 - toID; \

var[toID].init\_value = 0; \

++var\_id; \

} \

}

int basic\_to\_asm(const char\* filename\_bas, const char\* filename\_asm) {

FILE \*fbas = fopen(filename\_bas, "r");

if (!fbas) return 1;

char \*asm\_code = (char \*) malloc(100 \* 16);

asm\_code[0] = 0;

struct stLines \*lines = (struct stLines \*) malloc(sizeof(\*lines) \* 100);

struct stVariables \*var = (struct stVariables \*) malloc(sizeof(\*var) \* 52);;

struct stGotoConflict \*\_goto = (struct stGotoConflict \*) malloc(sizeof(\*\_goto) \* 100);;

char \*buffer = (char \*) malloc(128);

unsigned address = 0;

unsigned line\_id = 0;

unsigned goto\_id = 0;

unsigned var\_id = 0;

unsigned line = 0;

char fail = 0, end = 0, tmp\_var = 'a';

//A-Z - basic variables

//a-z - temp variables

while (end == 0 && fscanf(fbas, "%u %[A-Z] ", &line, buffer) != 0) {

if (line\_id != 0 && line <= lines[line\_id].line\_number) {

end = fail = 1;

break;

}

int type = command\_type(buffer);

lines[line\_id].line\_number = line;

lines[line\_id].start\_address = address;

++line\_id;

type\_check:

switch (type) {

case 0:

fail = 1;

break;

case 1:

continue;

case 2: {

fscanf(fbas, "%[A-Z]", buffer);

if (buffer[1] != 0 || !(buffer[0] >= 'A' && buffer[0] <= 'Z')) {

end = fail = 1;

break;

}

unsigned id = variable\_id(var, var\_id, buffer[0]);

if (id == var\_id) {

var[id].name = buffer[0];

var[id].address = 99 - id;

var[id].init\_value = 0;

++var\_id;

}

sprintf(buffer, "%02u READ %02u\n", address, var[id].address);

strcat(asm\_code, buffer);

}

break;

case 3: {

fscanf(fbas, "%[A-Z]", buffer);

if (buffer[1] != 0 || !(buffer[0] >= 'A' && buffer[0] <= 'Z')) {

end = fail = 1;

break;

}

unsigned id = variable\_id(var, var\_id, buffer[0]);

if (id == var\_id) {

var[id].name = buffer[0];

var[id].address = 99 - id;

var[id].init\_value = 0;

++var\_id;

}

sprintf(buffer, "%02u WRITE %02u\n", address, var[id].address);

strcat(asm\_code, buffer);

}

break;

case 4: {

sprintf(buffer, "%02u JUMP ", address);

strcat(asm\_code, buffer);

fscanf(fbas, "%u", &line);

if (line > lines[line\_id - 1].line\_number) {

\_goto[goto\_id].goto\_line = line;

\_goto[goto\_id].instratuction\_address = strlen(asm\_code);

\_goto[goto\_id].calc = 0;

strcat(asm\_code, "00\n");

++goto\_id;

++address;

continue;

}

int id = line\_id - 1;

while (id >= 0 && line != lines[id].line\_number)

--id;

if (id < 0) {

end = fail = 1;

break;

}

sprintf(buffer, "%02u\n", lines[id].start\_address);

strcat(asm\_code, buffer);

}

break;

case 5: {

char op1[8], op2[8];

fscanf(fbas, "%[0-9A-Z] %1[<=>] %[0-9A-Z]", op1, buffer, op2);

if (isdigit(op1[0]) && isdigit(op2[0])) {

int result = 0;

switch (buffer[0]) {

case '<':

result = atoi(op1) < atoi(op2);

break;

case '>':

result = atoi(op1) > atoi(op2);

break;

default:

result = atoi(op1) == atoi(op2);

break;

}

if (result == 0)

continue;

fscanf(fbas, "%[A-Z]", buffer);

type = command\_type(buffer);

goto type\_check;

}

//Определяем индексы переменных и регистрируем если требуется (могут быть времененные)

int id1 = var\_id, id2 = var\_id;

if (isdigit(op1[0])) {

var[id1].name = tmp\_var;

var[id1].address = 99 - var\_id;

var[id1].init\_value = atoi(op1);

++var\_id;

++tmp\_var;

id2 = variable\_id(var, var\_id, op2[0]);

if (id2 == var\_id) {

var[id2].name = op2[0];

var[id2].address = 99 - var\_id;

var[id2].init\_value = 0;

++var\_id;

}

} else if (isdigit(op2[0])) {

var[id2].name = tmp\_var;

var[id2].address = 99 - var\_id;

var[id2].init\_value = atoi(op2);

++var\_id;

++tmp\_var;

id1 = variable\_id(var, var\_id, op1[0]);

if (id1 == var\_id) {

var[id1].name = op1[0];

var[id1].address = 99 - var\_id;

var[id1].init\_value = 0;

++var\_id;

}

} else {

id1 = variable\_id(var, var\_id, op1[0]);

if (id1 == var\_id) {

var[id1].name = op1[0];

var[id1].address = 99 - var\_id;

var[id1].init\_value = 0;

++var\_id;

}

id2 = variable\_id(var, var\_id, op2[0]);

if (id2 == var\_id) {

var[id2].name = op2[0];

var[id2].address = 99 - var\_id;

var[id2].init\_value = 0;

++var\_id;

}

}

switch (buffer[0]) {

case '<':

//Если при вычитание из 2 1го число отрицательное, значит оно больше

sprintf(buffer, "%02u LOAD %02u\n%02u SUB %02u\n%02u JNEG 00\n",

address, var[id2].address,

address + 1, var[id1].address,

address + 2);

address += 3;

\_goto[goto\_id].goto\_line = line + 1;

\_goto[goto\_id].instratuction\_address = strlen(asm\_code) + 29;

\_goto[goto\_id].calc = 1;

++goto\_id;

break;

case '>':

//Если при вычитание из 1 2e число отрицательное, значит оно больше

sprintf(buffer, "%02u LOAD %02u\n%02u SUB %02u\n%02u JNEG 00\n",

address, var[id1].address,

address + 1, var[id2].address,

address + 2);

address += 3;

\_goto[goto\_id].goto\_line = line + 1;

\_goto[goto\_id].instratuction\_address = strlen(asm\_code) + 29;

\_goto[goto\_id].calc = 1;

++goto\_id;

break;

default:

//Если при вычитание из 2 1го число 0 - равны

sprintf(buffer, "%02u LOAD %02u\n%02u SUB %02u\n%02u JZ %02u\n%02u JUMP 00\n",

address, var[id2].address,

address + 1, var[id1].address,

address + 2, address + 4,

address + 3);

address += 4;

\_goto[goto\_id].goto\_line = line + 1;

\_goto[goto\_id].instratuction\_address = strlen(asm\_code) + 38;

\_goto[goto\_id].calc = 1;

++goto\_id;

break;

}

strcat(asm\_code, buffer);

fscanf(fbas, " %[A-Z] ", buffer);

type = command\_type(buffer);

goto type\_check;

}

break;

case 6: {

fscanf(fbas, "%[A-Z]", buffer);

if (buffer[1] != 0 || !(buffer[0] >= 'A' && buffer[0] <= 'Z')) {

end = fail = 1;

break;

}

char toVar = buffer[0];

char exp[256] = "\0";

char rpn[256];

fgets(exp, 255, fbas);

translate\_to\_rpn(rpn, exp);

if (strlen(rpn) > 1) {

char stack[100] = "\0";

int pos = 0, flg = 0;

for (size\_t i = 0; rpn[i]; i++) {

if (pos > 1 && isalnum(rpn[i]) == 0) {

unsigned id;

char var1 = stack[pos - 2];

getVarID(id, var1);

sprintf(buffer, "%02u LOAD %02u\n", address, var[id].address);

strcat(asm\_code, buffer);

++address;

char var2 = stack[pos - 1];

getVarID(id, var2);

if (rpn[i] == '+')

sprintf(buffer, "%02u ADD %02u\n", address, var[id].address);

else if (rpn[i] == '-')

sprintf(buffer, "%02u SUB %02u\n", address, var[id].address);

else if (rpn[i] == '\*')

sprintf(buffer, "%02u MUL %02u\n", address, var[id].address);

else if (rpn[i] == '/')

sprintf(buffer, "%02u DIV %02u\n", address, var[id].address);

strcat(asm\_code, buffer);

++address;

id = var\_id;

var[id].name = tmp\_var;

var[id].address = 99 - id;

var[id].init\_value = 0;

++tmp\_var;

++var\_id;

sprintf(buffer, "%02u STORE %02u\n", address, var[id].address);

strcat(asm\_code, buffer);

++address;

pos--;

/\*if (islower(stack[pos])) {

--tmp\_var;

--var\_id;

}

if (islower(stack[pos - 1])) {

--tmp\_var;

--var\_id;

}\*/

stack[pos - 1] = tmp\_var - 1;

} else {

stack[pos] = rpn[i];

pos++;

}

}

sprintf(buffer, "%02u LOAD %02u\n", address, var[var\_id - 1].address);

strcat(asm\_code, buffer);

++address;

int id;

getVarID(id, toVar);

sprintf(buffer, "%02u STORE %02u\n", address, var[id].address);

strcat(asm\_code, buffer);

} else {

int id = var\_id;

if (isdigit(rpn[0])) {

id = var\_id;

++var\_id;

if (variable\_id(var, var\_id, toVar) == var\_id) {

var[id].name = toVar;

var[id].address = 99 - id;

var[id].init\_value = atoi(rpn);

--address;

break;

} else {

var[id].name = tmp\_var;

var[id].address = 99 - id;

var[id].init\_value = atoi(rpn);

++tmp\_var;

}

} else {

id = variable\_id(var, var\_id, rpn[0]);

if (id == var\_id) {

var[id].name = rpn[0];

var[id].address = 99 - id;

var[id].init\_value = 0;

++var\_id;

}

}

sprintf(buffer, "%02u LOAD %02u\n", address, var[id].address);

strcat(asm\_code, buffer);

++address;

id = variable\_id(var, var\_id, toVar);

if (id == var\_id) {

var[id].name = toVar;

var[id].address = 99 - id;

var[id].init\_value = 0;

++var\_id;

}

sprintf(buffer, "%02u STORE %02u\n", address, var[id].address);

strcat(asm\_code, buffer);

}

}

break;

case 7:

sprintf(buffer, "%02u HALT 00", address);

strcat(asm\_code, buffer);

end = 1;

break;

}

++address;

if (type == 6)

continue;

int ignore;

do { ignore = fgetc(fbas); }

while (ignore != '\n' && ignore != EOF);

if (ignore == EOF)

break;

}

//Запись в память начальных значений + констатнт

//Проход по конфликтным переходам и заполнение их

if (address + var\_id > 99)

fail = 1;

if (fail == 0) {

int id = 0;

for (; id < goto\_id; ++id) {

int lid = 0;

while (lid < line\_id &&

lines[lid].line\_number < \_goto[id].goto\_line)

++lid;

if (lid != line\_id) {

sprintf(asm\_code + \_goto[id].instratuction\_address, "%02u", lines[lid].start\_address);

asm\_code[\_goto[id].instratuction\_address + 2] = '\n';

}

}

for (id = var\_id - 1; id >= 0; --id) {

sprintf(buffer, "\n%02u = %x", var[id].address, var[id].init\_value);

strcat(asm\_code, buffer);

}

FILE \*fasm = fopen(filename\_asm, "w");

if (fasm != NULL) {

fputs(asm\_code, fasm);

fclose(fasm);

} else

fail = 1;

}

free(buffer);

free(\_goto);

free(var);

free(lines);

free(asm\_code);

return fail == 1;

}