Федеральное агентство по связи и информатизации Российской Федерации

Сибирский Государственный Университет Телекоммуникаций и Информатики

**Курсовая работа**

по дисциплине

«ЭВМ и Периферийные устройства»

на темы:

**«Обработка команд центральным процессом, Транслятор с языка Simple Assembler,**

**Транслятор с языка Simple Basic»**

Выполнил: студент группы

ИП-712

Самсонов Д. В.

Руководитель: Токмашева Е. И.

Проверил: Майданов Ю.С.

Новосибирск

2019г.

СОДЕРЖАНИЕ

1 Текст задания к курсовой работе…….……..……….…………………….3

2 Архитектура Вычислительной машины Simple Computer.……………..5

3 Содержание отчета по проделанной работе……..……………….…….9

3.1 Постановка задачи исследования….......….…………………………9

3.2 Программная реализация….....…………………….………………….9

3.3 Результаты проведённого исследования….....…………………....12

3.4 Выводы….....…………………….……………………………………....13

Список использованной литературы….......……………………....…….…13

4. Приложение………………………………………………………………….14

**Текст задания к курсовой работе**

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 Вычислительной машины Simple Computer**

Архитектура *Simple Computer* включает следующие функциональные блоки:

* оперативную память;
* внешние устройства;
* центральный процессор.

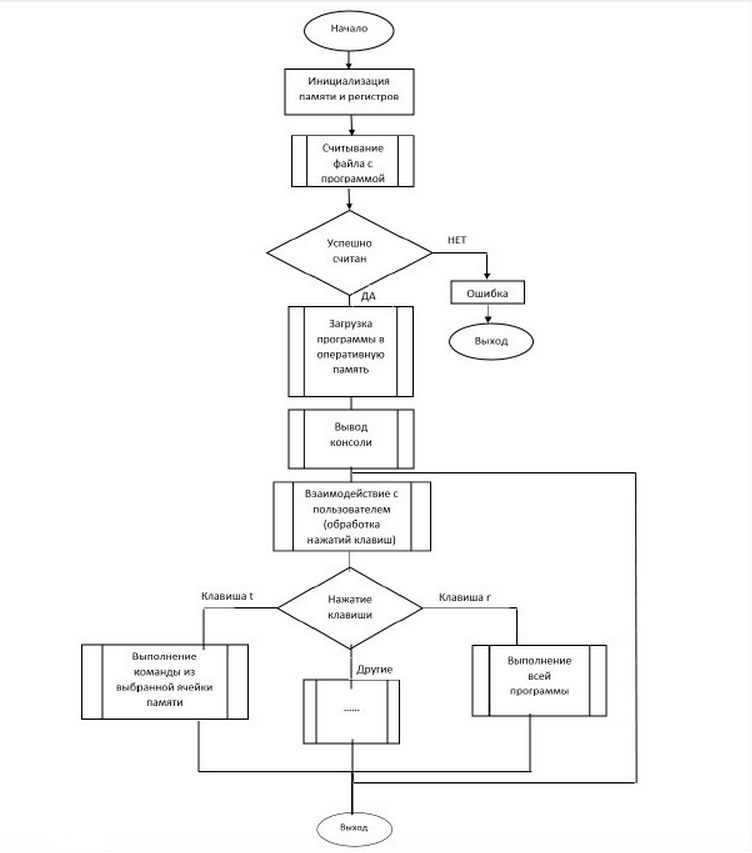


Рисунок 1 – Архитектура вычислительной машины Simple Computer

**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.

**Система команд Simple Computer**

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



Рисунок 2 – Формат команды центрального процесса Simple Computer

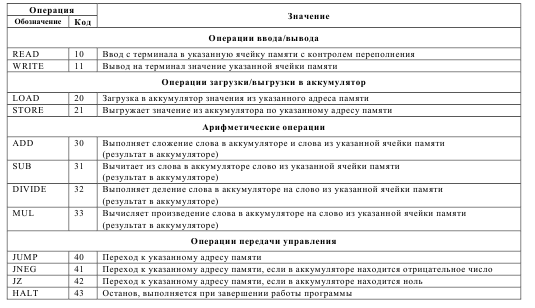


Таблица команд центрального процессора Simple Computer

**Выполнение команд центральным процессором Simple Computer**

Команды выполняются последовательно. Адрес ячейки памяти, в которой находится текущая выполняемая команда, задается в регистре «Счетчик 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 передается в АЛУ. В противном случае операция выполняется устройством управления. Процедура выполняется до тех пор, пока флаг «останов» не будет равен 1.

**Консоль управления**

Консоль управления содержит следующие области:

* *Memory* – содержимое оперативной памяти *Simple Computer*.
* *Accumulator* – значение, находящееся в аккумуляторе;
* *instructionCounter* – значение регистра «счетчик команд»;
* *Operation* – результат декодирования операции;
* *Flags* – состояние регистра флагов («П» - переполнение при выполнении операции,
* «0» - ошибка деления на 0, «М» - ошибка выхода за границы памяти, «Т» - игнорирование
* тактовых импульсов, «Е» - указана неверная команда);
* *Cell* – значение выделенной ячейки памяти в области *Memory* (используется для редактирования);
* *Keys* – подсказка по функциональным клавишам;
* *Input/Otput* – область, используемая *Simple Computer* в процессе выполнения программы для ввода информации с клавиатуры и вывода еѐ 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.

Пользователь имеет возможность с помощью клавиш управления курсора выбирать ячейки оперативной памяти и задавать им значения. Нажав клавишу *F5*, пользователь может задать значение аккумулятору, *F6* – регистру «счетчик команд». Сохранить содержимое памяти (в бинарном виде) в файл или загрузить его обратно пользователь может, нажав на клавиши «*l*», «*s*» соответственно (после нажатия в поле *Input/Output* пользователю предлагается ввести имя файла). Запустить программу на выполнение (установить значение флага «игнорировать такты таймера» в 0) можно с помощью клавиши *r*. В процессе выполнения программы, редактирование памяти и изменение значений регистров недоступно. Чтобы выполнить только текущую команду пользователь может нажать клавишу *t*. Обнулить содержимое памяти и задать регистрам значения «по умолчанию» можно нажав на клавишу *i*.

**Содержание отчета по проделанной работе**

**Постановка задачи исследования**

Необходимо разработать программную модель простейшей вычислительной машины Simple Computer. Для управления модели (определяющая начальные состояние узлов Simple Computer, запуска команд для выполнения программы, отражения хода выполнения программы) требуется создать консоль. Необходимо реализовать транслятор с языка Simple Basic на Simple Assembler , который транслируется в машинный код для программирования Simple Computer.

**Программная реализация**

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

**main.c (console)**

|  |  |
| --- | --- |
| **Функции** | **Описание** |
| void printall(void) | Вывод рамок и информационного текста |
| void DrawInsCounter() | Перерисовывает содержимое регистра счетчика команд SimpleComputer. |
| void setFlag(enum flags flag) | Задаёт определённый флаг в true |
| void DrawOperationField() | Перерисовывает результат декодирования операции ячейки, адресуемой счетчиком команд |
| void DrawFlags() | Перерисовывает содержимое регистра флагов SimpleComputer. |
| void printBigChar() | Перерисовывает содержимое текущей ячейки памяти |
| void break() | Содержит в себе PrintMemory(), DrawAcc(), DrawInsCounter(), DrawOperationField(), DrawFlags(), DrawBigChars() |
| void printWindow() | Рисует консоль SimpleComputer |
| int ALU(int command, int operand) | Выполняет Арифметико-логические операции |
| int CU() | Выполняет Устройство управления |
| void reset() | Ставит консоль в начальные значения (по умолчанию) |
| void signal handler(int sig) | Обработчик тактовых импульсов |
| int run() | Позыв сигнала каждую секунду , если консоль не занята выполнением команды |
| int main() | Принимает полученный ввод с клавиатуры , а затем с помощью проверок определяет тип команды и действо, которое нужно совершить |

**myBigChars.h**

|  |  |
| --- | --- |
| **Функции** | **Описание** |
| int bc\_printA (char \*str) | Ставит неканоническую работу терминала, передает ей символ и выводит его, после возвращает режим по умолчанию |
| int bc\_box (int x1, int y1, int x2, int y2) | С помощью перевода терминал в неканонический режим и махинацией с выводом символов рисует рамку |
| int bc\_printbigchar (unsigned long long big, int x, int y) | С помощью по битового выставления рисует ячейку памяти большими символами. |

**myReadkey.h**

|  |  |
| --- | --- |
| **Функции** | **Описание** |
| int rk\_readkey (enum keys\* key) | Определяет какая, клавиша была нажата |
| int rk\_mytermsave() | Сохраняет настройки терминала |
| int rk\_mytermrestore() | Возвращает терминалу стандартные настройки |
| int rk\_mytermregime(int regime, int vtime, int vmin, int echo, int sigint) | С помощью махинаций изменяет работу терминала |

**mySimpleComputer.h**

|  |  |
| --- | --- |
| **Функции** | **Описание** |
| int sc\_memoryInit() | Выполняет инициализацию оперативной памяти |
| int sc\_memorySet(int address, int value) | Задает значение ячейки памяти |
| int sc\_memoryGet(int address, int \*value) | Возвращает значение ячейки памяти |
| int sc\_memorySave(char \*filename) | Сохраняет содержимое памяти в файл в бинарном виде |
| int sc\_memoryLoad(char \*filename) | Загружает содержимое из бинарного файла в память |
| int sc\_regInit (void) | Выполняет инициализацию регистра флагов |
| int sc\_regSet(int reg, int value) | Устанавливает значение указанного регистра флагов |
| int sc\_regGet(int reg ,int \*value) | Возвращает значение указанного флага |
| int sc\_commandEncode (int command, int operand, int \* value) | Кодирует команду с указанным номером и операндом |
| int sc\_commandDecode (int value, int \* command, int \* operand) | Декодирует значение как команду Simple Computer |

**myTerm.h**

|  |  |
| --- | --- |
| **Функции** | **Описание** |
| int mt\_clrscr(void) | Очистка экрана |
| int mt\_getscreensize(int \*rows , int \*cols) | Определяет размер экрана терминала |
| int mt\_gotoXY(int x, int y) | Перемещает курсор в указанную позицию |
| int mt\_setfgcolor(int colors ) | Устанавливает цвет последующих выводимых символов |
| int mt\_setbgcolor(int colors ) | Устанавливает цвет фона последующих выводимых символов |
| int mt\_cursorVisible(int flag) | Отключает или включает курсор |
| void colorType(int type) | Задаёт стиль терминалу |
| void CLR() | Очищает строку ввода |

**main (assembler)**

|  |  |
| --- | --- |
| **Функции** | **Описание** |
| int get\_str(int fd, char \*buff) | Получает строку и обрабатывает её |
| Int translator\_assembler(char \*filename) { | Считывает входной файл, изменят команды на понятные машине, после декодирует и записывает их в память, а затем выводит в двоичный файл. |

**main (basic)**

|  |  |
| --- | --- |
| **Функции** | **Описание** |
| int() | Инициализация массива видоизмененного Simple Basic |
| struct GLOBAL\_VARIABLES \*StupidLookup(const char \*name) | Простым пробегом ищет глобальные переменные программы на basic |
| unsigned int \_\_allocate(const char \*variable) | Определяет адресс ячейки памяти в Ассемблере |
| struct GLOBAL\_VARIABLES \*LET(char \*expr)void IF(char \*operand, int line, FILE file2) | Высчитывает значение присваиваемое командой LET |
| void IF(char \*operand, int line, FILE file2) | Выполняет IF |
| char \*trim(char \*trimmed, unsigned) | Удаляет символ пробела из строки |
| long int BracerCalculate(char \*expr) | Калькулятор операций -+/\* |
| int main(int argc, char\*\* argv) часть1 | Транслятор |
| int main(int argc, char\*\* argv) часть2 | Парсер |

**Результаты проведённого исследования**

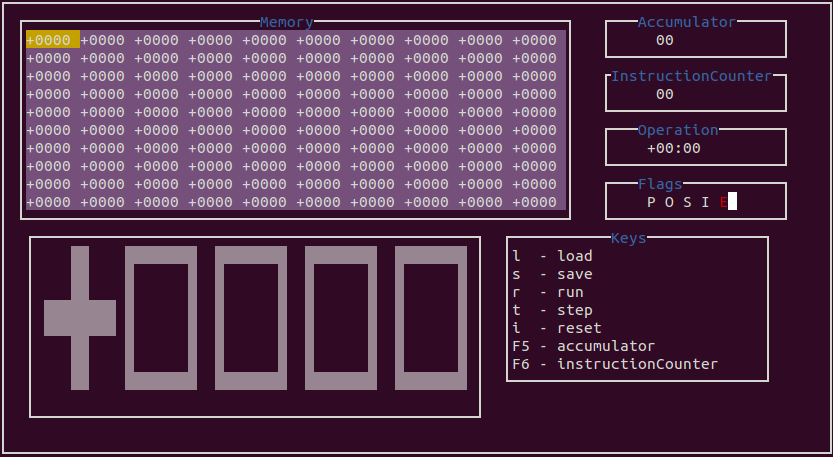


Рисунок 3 – показывает работу Simple Computer в начальной стадии

Рисунок 4 – показывает работу Simple Computer после загрузки в ней НОД

**Приложение (Листинг Кода)**

**Basic translator:**

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <time.h>

#define TRUE 1

#define FALSE 0

#define MAGICAL\_NUMBER 100

char operators[][6] = { "LET", "IF", "OUTPUT", "INPUT", "REM", "GOTO", "END" };

struct GLOBAL\_VARIABLES {

long int value;

char \*name;

unsigned int address;

struct GLOBAL\_VARIABLES \*next;

}\*Global;

int end = 0;

int allocate = 0;

int count = 0;

struct Expression {

long double lval;

char op;

long double rval;

}\*term;

struct Line {

int number;

char operator[6];

char \*operand;

}\*line;

struct Assembler {

unsigned int address;

char operator[6];

int operand;

}\*ass;

char \*trim(char \*trimmed, unsigned int);

unsigned int \_\_allocate(const char \*variable);

struct GLOBAL\_VARIABLES \*StupidLookup(const char \*name);

long double Execute(long double lval, long double rval, char operation)

{

if (operation == '\*')

return lval \* rval;

if (operation == '/') {

if (rval == 0) {

printf("Executing error! diversion by 0");

exit(EXIT\_FAILURE);

}

return lval / rval;

} if (operation == '+')

return lval + rval;

if (operation == '-')

return lval - rval;

return 0;

}

int nextPriority(int \*prty, int size)

{

int i = 0;

for (i = 0; i + 1 < size; i++)

prty[i] = prty[i + 1] - 1;

prty[i] = 0;

return i;

}

int Remove(void \*A, int i, int size)

{

int j = 0;

for (j = i; j + 1 < size; j++)

\*((long double \*)A + j) = \*((long double \*) A + j + 1);

\*((long double \*) A + j) = 0;

return j;

}

long int BracerCalculate(char \*expr)

{

struct Expression \*term = malloc(sizeof(term) \* MAGICAL\_NUMBER);

int i = 0, j = 0, len = strlen(expr), digcount,mulprty[len], mulsize = 0, size = 0;

long dig = 0;

long double res = 0, \*values = calloc(sizeof(values), len);

char \*operations = calloc(sizeof(operations), len / 2), \*suf;

for (i = 0; ; i++) {

dig = values[i] = atol(expr);

if (dig < 0)

expr++;

for (digcount = 1; (dig /= 10) != 0; digcount++);

expr += digcount;

if (expr[0] == '\0')

break;

operations[i] = expr[0];

expr++;

if (operations[i] == '\*' || operations[i] == '/') {

mulprty[mulsize] = i;

mulsize++;

} else if (operations[i] != '+' && operations[i] != '-') {

printf("Parsing error! Invalid arithmetic operator: %c", operations[i]);

exit(EXIT\_FAILURE);

}

}

i++;

size = i;

for (; mulsize > 0; ) {

values[mulprty[0]] = Execute(values[mulprty[0]], values[mulprty[0] + 1], operations[mulprty[0]]);

size = Remove(values, mulprty[0] + 1, size);

suf = strdup(operations + mulprty[0] + 1);

operations[mulprty[0]] = '\0';

strcat(operations, suf);

if ((mulsize = nextPriority(mulprty, mulsize)) == 0)

break;;

}

for (j = 0; strlen(operations) != 0; ) {

values[0] = Execute(values[0], values[1], operations[0]);

size = Remove(values, 1, size);

operations++;

}

res = values[0];

return res;

}

struct GLOBAL\_VARIABLES \*LET(char \*expr)

{

char \*var = calloc(sizeof(var), MAGICAL\_NUMBER);

int varPos;

expr = strchr(expr, '=');

for (varPos = 0, expr--; expr[0] == ' '; expr--, varPos++);

var[0] = expr[0];

expr += varPos + 2;

char \*bracer, \*rside, \*startbrace, \*endbrace;

struct GLOBAL\_VARIABLES \*found = NULL;

while ((startbrace = strrchr(expr, '(')) != NULL) {

if ((endbrace = strchr(startbrace, ')')) != NULL) {

rside = strdup(endbrace + 1);

endbrace[0] = '\0';

bracer = strdup(startbrace + 1);

startbrace[0] = '\0';

} else {

printf("Parsing error! Wasn't expected closing bracer");

exit(EXIT\_FAILURE);

}

sprintf(expr, "%s%ld%s", expr, BracerCalculate(bracer), rside);

}

if ((startbrace = strrchr(expr, '(')) == NULL) {

if ((found = StupidLookup(var)) == NULL) {

Global[end].address = \_\_allocate(var);

Global[end - 1].value = BracerCalculate(expr);

printf("%s = %ld\n", var, Global[end - 1].value);

return &Global[end - 1];

} else {

found->value = BracerCalculate(expr);

printf("%s = %ld\n", var, found->value);

return found;

}

}

return NULL;

}

unsigned int is\_Word(const char \*word)

{

return (word[0] >= 'A' && word[0] <= 'z');

}

struct GLOBAL\_VARIABLES \*StupidLookup(const char \*name)

{

int i;

for (i = 0; i < end; i++) {

if (strcmp(Global[i].name, name) == 0)

return &Global[i];

}

return NULL;

}

void IF(char \*operand, int line, FILE file2)

{

char \*condition = calloc(sizeof(condition), MAGICAL\_NUMBER), \*number, cmpr = 0;

struct GLOBAL\_VARIABLES \*found = NULL;

long int lrvals[2];

int len = 0, i = 0, j = 0;

while (1) {

for (i = 0; operand[i] != '>' && operand[i] != '<' && operand[i] != '=' &&

strlen(condition) > 0; i++)

condition[i] = operand[i];

if (is\_Word(condition)) {

if ((found = StupidLookup(condition)) != NULL) {

lrvals[j] = found->value;

j++;

}else

printf("Translation error! \"%s\" is Uninitialized variable", condition);

exit(EXIT\_FAILURE);

} else if (condition[0] == '<') {

cmpr = '<';

if (lrvals[0] < lrvals[1]) {

number = operand + strlen(operand);

ass[count].operand = atoi(number);

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

strcpy(ass[count].operator, operand);

}

} else if (condition[0] == '>') {

cmpr = '>';

if (lrvals[0] > lrvals[1]) {

number = operand + strlen(operand);

ass[count].operand = atoi(number);

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

strcpy(ass[count].operator, operand);

}

} else if (condition[0] == '=') {

cmpr = '=';

if (lrvals[0] == lrvals[1]) {

number = operand + strlen(operand);

ass[count].operand = atoi(number);

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

strcpy(ass[count].operator, operand);

}

} else {

lrvals[j] = atol(condition);

j++;

printf("Wrong compare operator");

return;

//operand += (operand - condition) + len;

}

if (cmpr != 0 && j >= 1 ) //complete condition

break;

}

}

char \*trim(char \*trimmed, unsigned int let)

{

int i = 0, j = 0, len = strlen(trimmed), reslen = len, blen = 0;

if (len == 0)

return trimmed;

char \*buf = calloc(sizeof(buf), len), \*\*tokens = calloc(sizeof(tokens), MAGICAL\_NUMBER), \*pos = NULL;

for (i = 0; len != 0; i++) {

sscanf(trimmed, "%s", buf);

pos = strstr(trimmed, buf);

blen = strlen(buf);

tokens[i] = strdup(buf);

trimmed -= trimmed - pos - blen;

len = strlen(trimmed);

}

trimmed = calloc(sizeof(trimmed), reslen);

for (j = 0; j < i; j++) {

if (!let)

strcat(tokens[j], " ");

strcat(trimmed, tokens[j]);

}

return trimmed;

}

int getrand(int min, int max)

{

srand(rand() + clock());

return (double)rand() / (RAND\_MAX + 1.0) \* (max - min) + min;

}

unsigned int \_\_allocate(const char \*variable/\*, long int value\*/)

{

struct GLOBAL\_VARIABLES \*found = NULL;

if ((found = StupidLookup(variable)) == NULL) {

Global[end].name = strdup(variable);

//Global[end].value = value;

Global[end].address = allocate;

allocate++;

end++;

return Global[end - 1].address;

} else {

return found->address;

}

return 0;

}

void printAssembler(FILE\* output, struct Assembler \*ass)

{

if (output == NULL) {

printf("Output file writing error!");

exit(EXIT\_FAILURE);

}

int i;

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

fprintf(output, "%d %s %d\n", ass[i].address, ass[i].operator, ass[i].operand);

}

int main()

{

FILE \*basic, \*output;

basic = fopen("input.basic", "r");

output = fopen("output.basic", "w+");

Global = malloc(sizeof(Global) \* MAGICAL\_NUMBER);

struct Line \*line= malloc(sizeof(line) \* MAGICAL\_NUMBER);

struct Assembler \*ass = malloc(sizeof(ass) \* MAGICAL\_NUMBER);

char \*expr = calloc(sizeof(expr), MAGICAL\_NUMBER);

struct GLOBAL\_VARIABLES \*found = NULL;

int len = 0, i = 0, j = 0, ended = FALSE; //j - memory access

for (i = 0; !feof(basic); i++) {

fgets(expr, MAGICAL\_NUMBER, basic);

len = strlen(expr);

if (len < 6)

continue;

if (expr[len - 1] == '\n')

expr[len - 1] = '\0';

sscanf(expr, "%d%s", &line[i].number, line[i].operator);

if (i > 0) {

if (line[i - 1].number >= line[i].number) {

printf("Translation error! The %d number of line is less that previous %d number of line!",

line[i].number, line[i - 1].number);

exit(EXIT\_FAILURE);

}

}

len = strlen(line[i].operator);

//Cases

if (strcmp(line[i].operator, operators[4]) == 0) { //REM

continue;

} else if (ended) {

printf("Parsing error! Was expected the token \"%s\" after the operator END",

line[i].operator);

exit(EXIT\_FAILURE);

}

expr += 4 + len;

//if (strlen(expr) > 0)

if (strcmp(line[i].operator, "LET") == 0) { //LET

expr = trim(expr, TRUE);

line[i].operand = strdup(expr);

found = LET(line[i].operand);

ass[count].address = \_\_allocate(found->name);

strcpy(ass[count].operator, "READ");

ass[count].operand = found->value;

count++;end++;

} else if (strcmp(line[i].operator, "IF") == 0) { //IF

strcpy(ass[count].operator, "JUMP");

ass[count].address = j;

ass[count].operand = 20;

j++;

count++;

} else if (!strcmp(line[i].operator, "OUTPUT")) {//OUTPUT

expr = trim(expr, TRUE);

line[i].operand = strdup(expr);

ass[count].address = j;

j++;

strcpy(ass[count].operator, "WRITE");

if ((found = StupidLookup(line[i].operand)) == NULL) {

printf("Translating error!! Trying to write unitilized variable %s",

line[i].operand);

exit(EXIT\_FAILURE);

} else

ass[count].operand = found->address;

count++;//ass[j].operand

} else if (strcmp(line[i].operator, "INPUT") == 0) {

expr = trim(expr, TRUE);

line[i].operand = strdup(expr);

ass[count].address = j;

j++;

strcpy(ass[count].operator, "READ");

ass[count].operand = \_\_allocate(line[i].operand);

count++;

} else if (strcmp(line[i].operator, "GOTO") == 0) {

expr = trim(expr, TRUE);

line[i].operand = strdup(expr);

ass[count].address = j;

strcpy(ass[count].operator, "JUMP");

ass[count].operand = atoi(line[i].operand);

j++;count++;

} else if (strcmp(line[i].operator, operators[6]) == 0) { //END

strcpy(ass[count].operator, "HALT");

ass[count].address = j;

j++;

strcpy(ass[count].operator, "HALT");

ass[count].operand = 0;

ended = TRUE;

count++;

continue;

} else {

printf("Translating error! Unknown command: \"%s\"", line[i].operator);

exit(EXIT\_FAILURE);

}

}

fclose(basic);

fclose(output);

return 0;

}

**Simple computer:**

#include <stdio.h>

#include <string.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <stdlib.h>

#include <fcntl.h>

#include <termios.h>

#include <sys/ioctl.h>

#include "lab1.h"

#include "lab2.h"

#include "lab3.h"

#include "lab4.h"

#include <signal.h>

#include <sys/time.h>

#define MAX(x, y) (x >= y) ? x : y

int idx = 0, AKKUM = 0;

int BIG[18][2]={

{0xFF818181,0x818181FF}, //0

{0x1010101,0x1010101}, //1

{0xFF010101,0xFF8080FF}, //2

{0xFF0101FF,0x10101FF}, //3

{0x818181FF,0x1010101}, //4

{0xFF8080FF,0x10101FF}, //5

{0xFF8080FF,0x818181FF}, //6

{0xFF010101,0x1010101}, //7

{0xFF8181FF,0x818181FF}, //8

{0xFF8181FF,0x10101FF}, //9

{0x18244281,0xFF818181}, //A

{0xFE8181FE,0x818181FE}, //B

{0xFF808080,0x808080FF}, //C

{0xFE818181,0x818181FE}, //D

{0xFF8080FF,0x808080FF}, //E

{0xFF8080FF,0x80808080}, //F

{0x181818FF,0xFF181818}, //+

{0xFF,0xFF000000} //-

};

void printInfo();

struct itimerval nval, oval;

enum flags {P, O, S, I, E, NONE};

int cols, rows, braked = 0;

void printFlags();

void printall(void)

{

int XUI[4][2] = {{0x81422418,0x18244281}, {0x81422418,0x10204080},{0x18838589,0x91A1C181},{0x0,0x11172421}};

mt\_setfgcolor(white);

bc\_box(2,1, 92, 25); //main frame

printf("\n");

bc\_box(3, 3, 61, 11); //Memory

bc\_box(15, 4, 50, 10); //bigchar field

bc\_box(15, 54 + 3, 29, 8); //keys

bc\_box(3, 68, 20, 2); //Accumulator

bc\_box(6, 68, 20, 2); //Instruction counters

bc\_box(9, 68, 20, 2); //Operation

bc\_box(12, 68, 20, 2); //Flags

mt\_setfgcolor(blue);

mt\_gotoXY(3, 63 / 2);

write(1,"Memory", 7);

mt\_gotoXY(3, 73 );

write(1,"Accumulator", 12);

mt\_gotoXY(6, 70);

write(1,"InstructionCounter", 19);

mt\_gotoXY(9, 73);

write(1,"Operation", 10);

mt\_gotoXY(12, 73 );

write(1,"Flags", 6);

mt\_gotoXY(15, 70);

write(1,"Keys", 4);

mt\_setfgcolor(white);

mt\_gotoXY(16, 59);

write(1,"l - load", 9);

mt\_gotoXY(17, 59);

write(1,"s - save", 9);

mt\_gotoXY(18, 59);

write(1,"r - run", 8);

mt\_gotoXY(19, 59);

write(1,"t - step", 9);

mt\_gotoXY(20, 59);

write(1,"i - reset", 11);

mt\_gotoXY(21 , 59);

write(1,"F5 - accumulator", 16);

mt\_gotoXY(22, 59);

write(1,"F6 - instructionCounter", 24);

mt\_gotoXY (cols - 5, rows);

}

void Break (int sig);

void setFlag(enum flags flag)

{

mt\_setbgcolor(NAN + 1);

mt\_gotoXY(13, 74 + flag + flag);

mt\_setfgcolor(red);

switch (flag) {

case P : write(1, "P", 1);

break;

case O : write(1, "0", 1);

break;

case S : write(1, "S", 1);

break;

case I : write(1, "I", 1);

break;

case E : write(1, "E", 1);

break;

case NONE : Break(0);

break;

}

mt\_setfgcolor(white);

}

void printBIG\_Char(int value)

{

int command = 0, operand = 0, pos[4];

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

pos[1] = command & 0xF;

pos[0] = (command >> 4) & 0xF;

pos[3] = operand & 0xF;

pos[2] = (operand >> 4) & 0xF;

bc\_printbigchar(BIG[16], 16, 6, NAN + 1, NAN + 1);

bc\_printbigchar(BIG[pos[0]], 16, 15, NAN + 1, NAN + 1);

bc\_printbigchar(BIG[pos[1]], 16, 25, NAN + 1, NAN + 1);

bc\_printbigchar(BIG[pos[2]], 16, 35, NAN + 1, NAN + 1);

bc\_printbigchar(BIG[pos[3]], 16, 45, NAN + 1, NAN + 1);

}

void printWindow()

{

if (idx > 99) {

printFlags();

setFlag(S);

idx = 99;

}

else

printFlags();

char \*chvalue = calloc(sizeof(chvalue), 4);

//int cols, rows;

//mt\_getscreensize(&cols, &rows);

int i = 0, j = 1, x = 4, y = 5;//cols - (cols / 2) + cols / 6;

int value = 0;

mt\_setfgcolor(white);

mt\_gotoXY(x, y);

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

if (i % 10 == 0 && i != 0) {

mt\_gotoXY(x + j, y);

j++;

}

sc\_memoryGet(i, &value);

if (idx == i) {

//mt\_gotoXY(7, cols - cols / 4);

mt\_setbgcolor(yellow);

//mt\_gotoXY(x + j, y);

} else

mt\_setbgcolor(purple);

int len = sprintf(chvalue, "+%.2X%.2X " , (value >> 8), (value & 0x7F));

write(1, chvalue, len);

mt\_setfgcolor(white);

}

}

void printFlags()

{

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

mt\_setbgcolor(NAN + 1);

mt\_setfgcolor(white);

mt\_gotoXY(13, 74 + P);

write(1, "P", 1);

mt\_gotoXY(13, 75 + O);

write(1, "O", 1);

mt\_gotoXY(13, 76 + S);

write(1, "S", 1);

mt\_gotoXY(13, 77 + I);

write(1, "I", 1);

mt\_gotoXY(13, 78 + E);

write(1, "E", 1);

sc\_regSet(TF, 0);

sc\_memoryGet(idx, &value);

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

sc\_regGet(TF, &value);

if(value)

setFlag(E);

}

void Break (int sig)

{

if (!braked)

idx = 0;

else

idx--;

printWindow();

printInfo();

printFlags();

}

void signalhandler (int signo)

{

printWindow();

setFlag(I);

printInfo();

idx++;}

void run (void)

{

int i;

signal (SIGALRM, signalhandler);

signal (SIGUSR1, Break);

nval.it\_interval.tv\_sec = 1; //And set the interval to 3second, 500 usecond repeatly

nval.it\_interval.tv\_usec = 200;

nval.it\_value.tv\_sec = 1;//from 1 to 0 second and 0 usecond

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

setitimer (ITIMER\_REAL, &nval, &oval);

enum keys key;

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

for (i = idx; i <= 100; i++) {

rk\_readkey(&key);

if (key == key\_esc) {

braked = 1;

break;

}

pause();

}

alarm(0);

raise(SIGUSR1);

}

void printInfo()

{

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

char \*counter = calloc(sizeof(counter), 10);

mt\_setbgcolor(NAN + 1);

mt\_setfgcolor(white);

mt\_gotoXY(4, 75);

int len = sprintf(counter, "%.2X", AKKUM);

write(1, counter, len);

mt\_gotoXY(7, 75);

len = sprintf(counter, "%2.2d", idx);

mt\_setbgcolor(NAN + 1);

mt\_setfgcolor(white);

write(1, counter, len);

sc\_memoryGet(idx, &value);

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

mt\_gotoXY(10, 74);

len = sprintf(counter, "+%.2X:%.2X", command, operand);

mt\_setbgcolor(NAN + 1);

mt\_setfgcolor(white);

write(1, counter, len);

/\*mt\_gotoXY(13, cols - cols / 5 + P);

write(1, "P", 1);

mt\_gotoXY(13, cols - cols / 5 + O + 1);

write(1, "O", 1);

mt\_gotoXY(13, cols - cols / 5 + S + 2);

write(1, "S", 1);

mt\_gotoXY(13, cols - cols / 5 + I + 3);

write(1, "I", 1);

mt\_gotoXY(13, cols - cols / 5 + E + 4);

write(1, "E", 1);\*/

printBIG\_Char(value);

mt\_gotoXY(cols, rows);

}

/\*

void loadMemory(int sig)

{

char \*filename = calloc(sizeof(filename), 100);

mt\_gotoXY(30, 2);

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

write(1, "Input name of file: ", sizeof("Input name of file: "));

mt\_gotoXY(30, 10);

scanf("%s", filename)read(0, filename, sizeof(filename));

if (sc\_memoryLoad(filename) == -1)

printf("Load error");

}\*/

void reset()

{

sc\_memoryInit();

mt\_clrscr();

printall();

printWindow();

printInfo();

printFlags();

mt\_gotoXY(30,5);

}

int alu(int command, int operand){

int t;

int i, value = 0;

switch(command){

case ADD:

sc\_memoryGet(operand-1,&value);

if (AKKUM + value <= 0x7FFF)

AKKUM += value;

else

sc\_regSet(S,1);

break;

case SUB:

sc\_memoryGet(operand - 1, &value);

if (AKKUM - value <= 0x7FFF)

AKKUM-=value;

else

sc\_regSet(S,1);

break;

case DIVIDE:

sc\_memoryGet(operand - 1, &value);

if (value == 0)

setFlag(O);

if (AKKUM/value<=0x7FFF)

AKKUM/=value;

break;

case MUL:

sc\_memoryGet(operand-1,&value);

if (AKKUM \* value <= 0x7FFF)

AKKUM\*=value;

else

sc\_regSet(P,1);

break;

case(CHL):

sc\_memoryGet(operand, &t);

AKKUM = t<<1;

break;

case(SHR):

sc\_memoryGet(operand, &t);

AKKUM = t>>1;

break;

case(RCL):

sc\_memoryGet(operand, &t);

if( t >> 14 & 0x1)

AKKUM = t>>1|0x4000;

else

AKKUM = t>>1;

break;

case(RCR):

sc\_memoryGet(operand, &t);

if( t & 0x1)

AKKUM = t<<1|0x4000;

else

AKKUM = t<<1;

break;

case(NEG):

sc\_memoryGet(operand, &t);

break;

}

return 0;

}

void cu(){

int command, operand;

int t;

sc\_memoryGet(idx, &t);

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

switch(command){

case(10):

sc\_regInit();

mt\_gotoXY(0,24);

sc\_regSet(I, 1);

mt\_gotoXY(0,24);

printf("Р’РІРµРґРёС‚Рµ Р·РЅР°С‡РµРЅРёРµ СЏС‡РµР№РєРё в„– %d - ", operand);

scanf("%x", &t);

sc\_memorySet(operand, t);

break;

case(11):

sc\_regInit();

mt\_gotoXY(0,24);

sc\_regSet(I, 1);

sc\_memoryGet(operand, &t);

mt\_gotoXY(0,24);

printf("Р—РЅР°С‡РµРЅРёРµ СЏС‡РµР№РєРё в„– %d = 0x%x, %d", operand, t, t);

sc\_regSet(I, 0);

break;

case(20):

sc\_memoryGet(operand, &AKKUM);

break;

case(21):

sc\_memorySet(operand, AKKUM);

break;

case(30): case(31):case(32):case(33):

alu(command, operand);

break;

case(40):

idx = operand-1;

break;

case(41):

if (AKKUM < 0)

idx = operand;

break;

case(42):

if (AKKUM == 0)

idx = operand;

break;

case(43):

sc\_regSet(I, 1);

return;

break;

case(51):case(52):case(53):case(54):case(55):case(56):case(57):case(58):case(59):

case(60):case(61):case(62):case(63):case(64):case(65):case(66):case(67):case(68):case(69):

case(70):case(71):case(72):case(73):case(74):case(75):case(76):

alu(command, operand);

break;

default:

sc\_regSet(I, 1);

sc\_regSet(E, 1);

}

idx++;

}

int main()

{

int flag = 0, value = 0;

mt\_getscreensize(&cols, &rows);

mt\_clrscr();

sc\_regInit();

printall();

printWindow();

printInfo();

printFlags();

sc\_memorySet(44, 0x6F05);

enum keys key;

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

while(key != key\_esc) {

flag = rk\_readkey(&key);

if (flag == 0){

switch (key) {

case key\_L : {

if (sc\_memoryLoad("output.bas") == -1) {

mt\_gotoXY(30, 2);

write(1, "Load error", sizeof("Load error"));

}

printWindow();

printInfo();

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

}

break;

case key\_S : {

if (sc\_memorySave("savefile.dat") == -1) {

mt\_gotoXY(30, 2);

write(1, "Save error", sizeof("Save error"));

}

printWindow();

printInfo();

}

break;

case key\_R : run();

break;

case key\_T : {

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

cu();

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

mt\_clrscr();

printall();

printWindow();

printInfo();

}

break;

case key\_I : reset();

break;

case key\_F5 : {

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

printf("Input Accumulator: ");

scanf("%X", &value);

AKKUM = value;

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

mt\_clrscr();

printall();

printWindow();

printInfo();

}

break;

case key\_F6 : {

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

printf("Input InstructionCounter: ");

scanf("%d", &idx);

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

mt\_clrscr();

printall();

printWindow();

printInfo();

}

break;

case key\_up : {

if (idx / 10 == 0) idx += 90;

else

idx -= 10;

printWindow();

printInfo();

}

break;

case key\_down : {

if (idx / 10 == 9) {

idx -= 90;

} else

idx += 10;

printWindow();

printInfo();

}

break;

case key\_left : {

if (idx < 0 || idx % 10 == 0) idx += 9;

else

idx--;

printWindow();

printInfo();

}

break;

case key\_right : {

if ((idx - 9) % 10 == 0) idx -= 9;

else

idx++;

printWindow();

printInfo();

}

break;

case key\_enter : {

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

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

mt\_gotoXY(30, 40);

printf("\nInput command: ");

scanf("%X", &command);

printf("Input operand: ");

scanf("%X", &operand);

if (command > 0x7F || operand > 0x7F)

setFlag(P);

int e = sc\_commandEncode(command, operand, &value);

if (e) {

mt\_clrscr();

printall();

printWindow();

printInfo();

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

break;

}

sc\_memorySet(idx, value);

sc\_regGet(TF, &value);

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

mt\_clrscr();

printall();

printWindow();

printInfo();

if(value) {

setFlag(E);

}

}

break;

}

flag = 1;

}

}

mt\_gotoXY(cols, rows);

return 0;

}

**Выводы**

1. Для реализации транслятора с Simple Basic ->Simple Assembler мне пришлось оптимизировать код и некоторые вычисления.
2. Написав эмуляцию ЭВМ я приобрёл опыт работы с ней и инструментами оболочки BASH и UNIX

**Список использованной литературы**

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

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

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

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