**Федеральное государственное бюджетное образовательное учреждение**

**«Сибирский государственный университет телекоммуникаций и информатики»**

Кафедра ВС

**КУРСОВАЯ РАБОТА**

По дисциплине «Архитектура ЭВМ»

Выполнил:

студент гр. ИП-014

Обухов А.И.

Проверил:

Майданов Ю. С.

Новосибирск 2022

**Оглавление**

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

Система команд Simple Computer…....................................................................8

Результаты проведённого исследования.........................................................................................................10

Заключение…………………................................................................................14

Литература………………….................................................................................15

Листинг………………….......................................................................................16

**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.

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

· оперативную память;

· внешние устройства;

· центральный процессор.

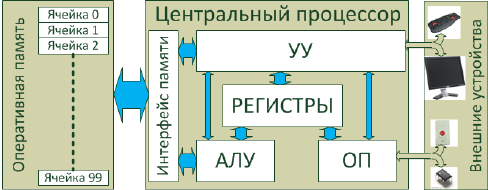


Рисунок 1 – Архитектура вычислительной машины Simple Computer

**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.

**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

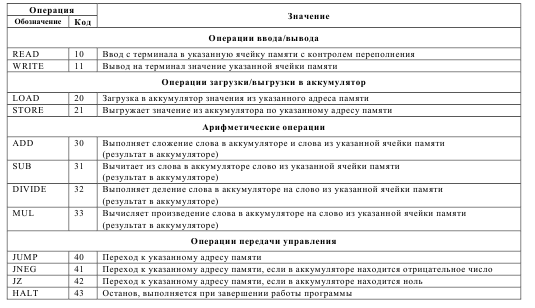


Рисунок 3 - Таблица команд центрального процессора Simple Computer

**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.

**Консоль управления**

Консоль управления содержит следующие области:

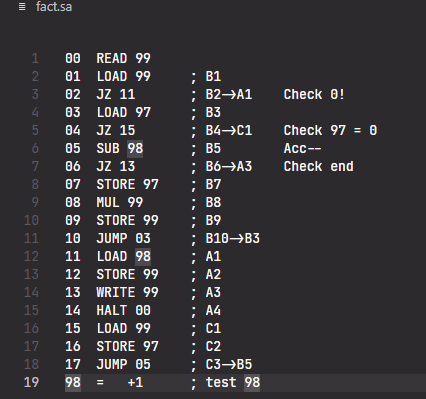
1. *Memory* – содержимое оперативной памяти *Simple Computer*.
2. *Accumulator* – значение, находящееся в аккумуляторе;
3. *instructionCounter* – значение регистра «счетчик команд»;
4. *Operation* – результат декодирования операции;
5. *Flags* – состояние регистра флагов («П» - переполнение при выполнении операции,
6. «0» - ошибка деления на 0, «М» - ошибка выхода за границы памяти, «Т» - игнорирование
7. тактовых импульсов, «Е» - указана неверная команда);
8. *Cell* – значение выделенной ячейки памяти в области *Memory* (используется для редактирования);
9. *Keys* – подсказка по функциональным клавишам;
10. *Input/Otput* – область, используемая *Simple Computer* в процессе выполнения программы для ввода информации с клавиатуры и вывода её 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.

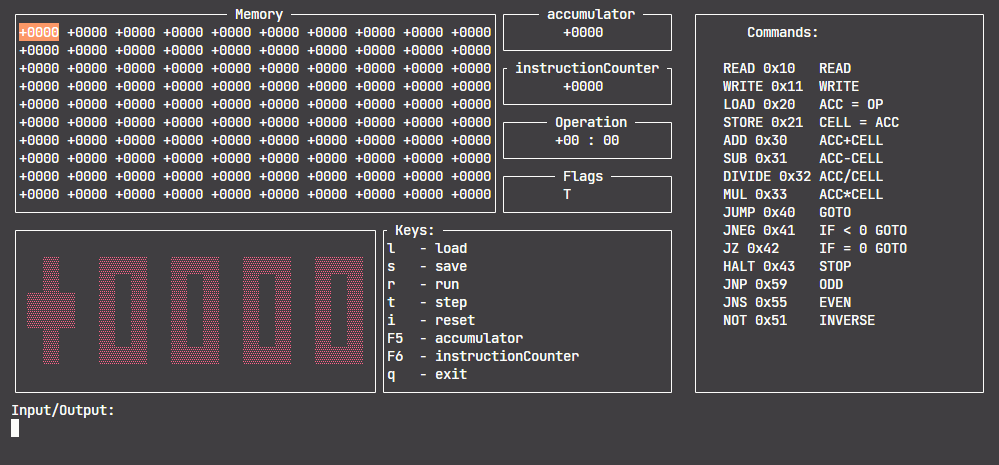
Пользователь имеет возможность с помощью клавиш управления курсора выбирать ячейки оперативной памяти и задавать им значения. Нажав клавишу *F5*, пользователь может задать значение аккумулятору, *F6* – регистру «счётчик команд». Сохранить содержимое памяти (в бинарном виде) в файл или загрузить его обратно 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*.

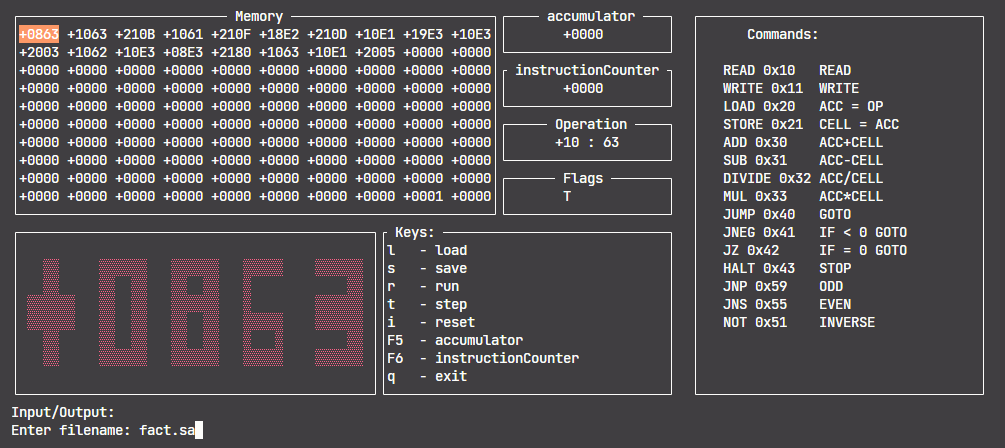
**Результаты проведённого исследования**

В качестве примера была взята программа подсчёта факториала, написанная на *Simple Assembler*.



**Пример работы программы**



Была представлена работа поиска факториала из 4. Полученный ответ представлен в 16-тиричной системе счисления 0x001816 = 2410

**Заключение**

В ходе курсовой работы были реализованы базовые команды CU и ALU. Реализована работа центрального процессора. Был написан транслятор Assembler в бинарный формат. Транслятор был автоматизирован для упрощения работы с программой.

**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**

Asm.c

#include "include/asm.h"

int cmd\_search(const char \*command)

{

    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, "JNP") == 0)

        return 0x59;

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

        return 0x55;

    else if (strcmp(command, "NOT") == 0)

        return 0x51;

    return -1;

}

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

{

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

    if (!fasm)

        return 1;

    int address, operand, code, encode, ignore;

    char cmd[16] = {0};

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

    {

        if (cmd[0] != '=')

        {

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

            code = cmd\_search(cmd);

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

                return 1;

        }

        else

        {

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

        }

        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;

    sc\_memorySave(filename\_object);

    return 0;

}

Cu.c

#include "include/cu.h"

int Cu()

{

    int value = 0;

    int temp = 0;

    sc\_memoryGet(instructionCounter, &value);

    int command;

    int operand;

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

    {

        sc\_regSet(T, 0);

        return 1;

    }

    if (operand >= SIZE || operand < 0)

    {

        sc\_regSet(M, 1);

        sc\_regSet(T, 1);

        return 1;

    }

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

    {

        Alu(command, operand);

    }

    else

    {

        switch (command)

        {

        case READ:

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

            read\_console\_value(operand, &value);

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

            if (value > 0x3FFF)

            {

                sc\_regSet(P, 1);

                break;

            }

            else

            {

                sc\_memorySet(operand, value);

                mt\_gotoXY(24, 1);

                break;

            }

        case WRITE:

            temp = write\_console\_value(operand, value);

            mt\_gotoXY(25, 1);

            printf("%d>\t%0X\n", operand, temp);

            break;

        case LOAD:

            sc\_memoryGet(operand, &accumulator);

            break;

        case STORE:

            sc\_memorySet(operand, accumulator);

            break;

        case JUMP:

            instructionCounter = --operand;

            break;

        case JNEG:

            if (accumulator < 0)

            {

                instructionCounter = --operand;

            }

            break;

        case JNS:

            if (accumulator > 0)

            {

                instructionCounter = --operand;

            }

            break;

        case JZ:

            if (accumulator == 0)

            {

                instructionCounter = --operand;

            }

            break;

        case HALT:

            sc\_regSet(T, 1);

            return 0;

            break;

        case NOT:

            value = ~accumulator;

            sc\_memorySet(operand, value);

            break;

        case JNP:

            if ((accumulator % 2) != 0)

            {

                instructionCounter = --operand;

            }

            break;

        }

    }

    if (instructionCounter + 1 < SIZE)

    {

        instructionCounter++;

        coord = instructionCounter;

    }

    else

    {

        sc\_regSet(M, 1);

    }

    return 0;

}

int Alu(int command, int operand)

{

    int value = 0;

    sc\_memoryGet(operand, &value);

    switch (command)

    {

    case ADD:

        if (accumulator + value > 0x3FFF)

        {

            sc\_regSet(P, 1);

            break;

        }

        accumulator += value;

        break;

    case SUB:

        if (accumulator - value < -0x3FFF)

        {

            sc\_regSet(P, 1);

            break;

        }

        accumulator -= value;

        break;

    case DIVIDE:

        if (accumulator == 0)

        {

            sc\_regSet(O, 1);

            break;

        }

        accumulator /= value;

        break;

    case MUL:

        if (accumulator \* value > 0x3FFF)

        {

            sc\_regSet(P, 1);

            break;

        }

        accumulator \*= value;

        break;

    default:

        sc\_regSet(E, 1);

        break;

    }

    return 0;

}

Main.c

#include "include/printConsole.h"

int main()

{

    run();

    return 0;

}

myBigChars.c

#include "include/myBigChars.h"

#include "include/myTerm.h"

int bc\_printA(const char \*str)

{

    printf(EN\_MACS);

    if (printf("%s", str) < 0)

    {

        return -1;

    }

    printf(EX\_MACS);

    return 0;

}

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

{

    if (x1 < 0 || y1 < 0)

        return -1;

    mt\_gotoXY(x1, y1);

    bc\_printA(BC\_ANGLE\_LEFT\_UP);

    for (int i = y1 + 1; i < y1 + y2 - 1; i++)

    {

        mt\_gotoXY(x1, i);

        bc\_printA(BC\_LINE\_HORIZONTAL);

    }

    bc\_printA(BC\_ANGLE\_RIGHT\_UP);

    for (int i = x1 + 1; i < x1 + x2 - 1; i++)

    {

        mt\_gotoXY(i, y1);

        bc\_printA(BC\_LINE\_VERTICAL);

    }

    mt\_gotoXY(x1 + x2 - 1, y1);

    bc\_printA(BC\_ANGLE\_LEFT\_DOWN);

    for (int i = x1 + 1; i < x1 + x2 - 1; i++)

    {

        mt\_gotoXY(i, y1 + y2 - 1);

        bc\_printA(BC\_LINE\_VERTICAL);

    }

    for (int i = y1 + 1; i < y1 + y2 - 1; i++)

    {

        mt\_gotoXY(x1 + x2 - 1, i);

        bc\_printA(BC\_LINE\_HORIZONTAL);

    }

    bc\_printA(BC\_ANGLE\_RIGHT\_DOWN);

    printf("\n");

    return 0;

}

int bc\_printbigchar(int arr[2], int x, int y, enum colors b\_col, enum colors f\_col)

{

    int w\_x, w\_y;

    mt\_getscreensize(&w\_x, &w\_y);

    if (x < 0 || y < 0 || x > w\_x || y > w\_y)

        return -1;

    char str[8] = {0};

    mt\_setfgcolor(f\_col);

    mt\_setbgcolor(b\_col);

    for (int i = 0; i < 2; i++)

    {

        for (int j = 0; j < 4; j++)

        {

            int a\_buf = arr[i] >> (j \* 8);

            int buf = a\_buf & 255;

            for (int k = 0; k < 8; k++)

            {

                int buf\_bit = (buf & (1 << k));

                if (buf\_bit != 0)

                {

                    str[k] = BLACK\_CHAR;

                }

                else

                {

                    str[k] = ' ';

                }

            }

            mt\_gotoXY(x + (i \* 4) + j + 1, y);

            bc\_printA(str);

        }

    }

    mt\_defcolor();

    mt\_gotoXY(24, 0);

    return 0;

}

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

{

    if (x < 0 || y < 0 || x > 7 || y > 7 || value > 1 || value < 0)

        return -1;

    int index = 0;

    if (x > 3)

        index = 1;

    if (value == 1)

    {

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

    }

    else if (value == 0)

    {

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

    }

    return 0;

}

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

{

    if (x < 0 || x > 7 || y < 0 || y > 7)

        return -1;

    int ind = 0;

    if (x > 3)

        ind = 1;

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

    return 0;

}

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

{

    if (big < 0 || fd < 0 || count < 0)

        return -1;

    if (write(fd, big, count \* 8) == -1)

        return -1;

    return 0;

}

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

{

    if (fd < 0 || need\_count < 0)

    {

        \*count = 0;

        return -1;

    }

    int result = read(fd, big, need\_count \* 8);

    if (result == -1)

    {

        \*count = 0;

        return -1;

    }

    \*count = result / 8;

    return 0;

}

myReadKey.c

#include "include/myReadkey.h"

int rk\_readkey(enum keys \*key)

{

    rk\_mytermregime(1, 0, 1, 1, 0);

    char buff[8] = {0};

    read(STDIN\_FILENO, buff, 8);

    if (strcmp(buff, "\E[A") == 0)

    {

        \*key = key\_up;

    }

    else if (strcmp(buff, "\E[B") == 0)

    {

        \*key = key\_down;

    }

    else if (strcmp(buff, "\E[C") == 0)

    {

        \*key = key\_right;

    }

    else if (strcmp(buff, "\E[D") == 0)

    {

        \*key = key\_left;

    }

    else if (strcmp(buff, "\E[15~") == 0)

    {

        \*key = key\_f5;

    }

    else if (strcmp(buff, "\E[17~") == 0)

    {

        \*key = key\_f6;

    }

    else if (buff[0] == 'l')

    {

        \*key = key\_l;

    }

    else if (buff[0] == 's')

    {

        \*key = key\_s;

    }

    else if (buff[0] == 'r')

    {

        \*key = key\_r;

    }

    else if (buff[0] == 't')

    {

        \*key = key\_tt;

    }

    else if (buff[0] == 'i')

    {

        \*key = key\_i;

    }

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

    {

        \*key = buff[0] - 48;

    }

    else if (buff[0] == 'q')

    {

        \*key = key\_q;

    }

    else

    {

        \*key = key\_other;

    }

    rk\_mytermrestore();

    return 0;

}

int rk\_mytermsave(void)

{

    if (tcgetattr(STDIN\_FILENO, &atr) != 0)

        return -1;

    return 0;

}

int rk\_mytermrestore(void)

{

    if (tcsetattr(STDIN\_FILENO, TCSAFLUSH, &atr) != 0)

            return -1;

    return 0;

}

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

{

    struct termios newatr;

    rk\_mytermsave();

    newatr = atr;

    if (regime == 0)

    {

        newatr.c\_lflag |= ICANON;

    }

    else if (regime == 1)

    {

        newatr.c\_lflag &= (~ICANON);

        if (echo == 0)

        {

            newatr.c\_lflag |= ECHO;

        }

        else if (echo == 1)

        {

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

        }

        if (sigint == 0)

        {

            newatr.c\_lflag |= ISIG;

        }

        else if (sigint == 1)

        {

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

        }

        newatr.c\_cc[VTIME] = vtime;

        newatr.c\_cc[VMIN] = vmin;

    }

    tcsetattr(STDIN\_FILENO, TCSAFLUSH, &newatr);

    return 0;

}

mySignal.c

#include "include/mySignal.h"

void signalhangle(int signal)

{

    int val;

    sc\_regGet(T, &val);

    if (val == 0)

    {

        Cu();

        resetTerm();

        mt\_gotoXY(24, 1);

        printf("%s", io\_msg);

    }

}

void reset\_signalhagle(int signal)

{

    accumulator = 0;

    instructionCounter = 0;

    setBGColor(0);

    coord = 0;

    setBGColor(1);

    sc\_regInit();

    sc\_regSet(T, 1);

    sc\_memoryInit();

    resetTerm();

}

void create\_timer(double interval)

{

    struct itimerval nval;

    nval.it\_value.tv\_sec = nval.it\_interval.tv\_sec = (long)interval;

    nval.it\_value.tv\_usec = nval.it\_interval.tv\_usec = (long)((interval - (long)interval) \* 1000000);

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

}

mySimpleComputer.c

#include "include/mySimpleComputer.h"

int RAM[100];

int flag;

int sc\_memoryInit()

{

    for (int i = 0; i < SIZE; ++i)

    {

        RAM[i] = 0;

    }

    return 0;

}

int sc\_memorySet(int address, int value)

{

    if (address >= SIZE || address < 0)

    {

        sc\_regSet(M, 1);

        return 1;

    }

    RAM[address] = value;

    return 0;

}

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

{

    if (address >= SIZE || address < 0)

    {

        sc\_regSet(M, 1);

        return 1;

    }

    \*value = RAM[address];

    return 0;

}

int sc\_memorySave(const char \*filename)

{

    FILE \*mem\_file = fopen(filename, "wb");

    fwrite(RAM, sizeof(int), SIZE, mem\_file);

    fclose(mem\_file);

    return 0;

}

int sc\_memoryLoad(const char \*filename)

{

    FILE \*mem\_file = fopen(filename, "rb");

    fread(RAM, sizeof(int), SIZE, mem\_file);

    fclose(mem\_file);

    return 0;

}

int sc\_regInit()

{

    flag = 0;

    return 0;

}

int sc\_regSet(int reg, int value)

{

    if (reg == P || reg == O || reg == M || reg == T || reg == E)

    {

        if (value == 0)

        {

            flag = flag & ~reg;

        }

        else if (value == 1)

        {

            flag = flag | reg;

        }

        else

        {

            return 0;

        }

    }

    return 0;

}

int sc\_regGet(int reg, int \*value)

{

    if (reg == P || reg == O || reg == M || reg == T || reg == E)

    {

        if ((flag & reg) != 0)

        {

            \*value = 1;

        }

        else

        {

            \*value = 0;

        }

    }

    return 0;

}

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

{

    if ((operand > 0x7F || operand < 0x0) &&

        (command != (0x10 || 0x11 || 0x20 || 0x21 || 0x30 || 0x31 || 0x32 || 0x33 || 0x40 || 0x41 || 0x42 || 0x43 ||

                     0x51 || 0x52 || 0x53 || 0x54 || 0x55 || 0x56 || 0x57 || 0x58 || 0x59 || 0x60 || 0x61 || 0x62 || 0x63 ||

                     0x64 || 0x65 || 0x66 || 0x67 || 0x68 || 0x69 || 0x70 || 0x71 || 0x72 || 0x73 || 0x74 || 0x75 || 0x76)))

    {

        sc\_regSet(E, 1);

        return 1;

    }

    \*value = (command << 7) | operand;

    return 0;

}

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

{

    if ((value >> 14) != 0)

    {

        if ((\*operand > 0x7F || \*operand < 0x0) &&

            (\*command != (0x10 || 0x11 || 0x20 || 0x21 || 0x30 || 0x31 || 0x32 || 0x33 || 0x40 || 0x41 || 0x42 || 0x43 ||

                          0x51 || 0x52 || 0x53 || 0x54 || 0x55 || 0x56 || 0x57 || 0x58 || 0x59 || 0x60 || 0x61 || 0x62 || 0x63 ||

                          0x64 || 0x65 || 0x66 || 0x67 || 0x68 || 0x69 || 0x70 || 0x71 || 0x72 || 0x73 || 0x74 || 0x75 || 0x76)))

        {

            sc\_regSet(E, 1);

            return 1;

        }

    }

    \*command = value >> 7;

    \*operand = value & 0x7F;

    return 0;

}

myTerm.c

#include "include/myTerm.h"

/\*

    clear\_screen=\E[H\E[J

    cursor\_address=\E[%i%p1%d;%p2%dH

    set\_a\_background=\E[4%p1%dm

    set\_a\_foreground=\E[3%p1%dm

    cursor\_invisible=\E[?25l\E[?1c

    cursor\_visible=\E[?25h\E[?8c

\*/

int mt\_clrscr(void)

{

    return printf("\E[H\E[J");

}

int mt\_gotoXY(int y, int x)

{

    return printf("\E[%d;%dH", y, x);

}

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 color)

{

    return printf("\E[3%dm", color);

}

int mt\_setbgcolor(enum colors color)

{

    return printf("\E[4%dm", color);

}

int mt\_defcolor()

{

    printf("\E[0m");

    return 0;

}

printConsole.c

#include "include/printConsole.h"

static int bcint0[2] = {1717992960, 8283750};

static int bcint1[2] = {471341056, 3938328};

static int bcint2[2] = {538983424, 3935292};

static int bcint3[2] = {2120252928, 8282238};

static int bcint4[2] = {2120640000, 6316158};

static int bcint5[2] = {2114092544, 8273984};

static int bcint6[2] = {33701376, 4071998};

static int bcint7[2] = {811630080, 396312};

static int bcint8[2] = {2120646144, 8283750};

static int bcint9[2] = {2087074816, 3956832};

static int bcintA[2] = {2118269952, 4342338};

static int bcintB[2] = {1044528640, 4080194};

static int bcintC[2] = {37895168, 3949058};

static int bcintD[2] = {1111637504, 4080194};

static int bcintE[2] = {2114092544, 8258050};

static int bcintF[2] = {33717760, 131646};

static int bcintp[2] = {2115508224, 1579134};

static int bcintm[2] = {2113929216, 126};

int time = 0;

int instructionCounter = 0;

void getXY(int \*x, int \*y)

{

    \*x = coord % 10;

    \*y = coord / 10;

}

int printInstructionCounterValue()

{

    mt\_gotoXY(5, 70);

    printf("+%.4X", instructionCounter);

    return 0;

}

int printAccumulatorValue()

{

    mt\_gotoXY(2, 70);

    if (accumulator < 0)

    {

        printf("-%.4X", -accumulator);

    }

    else

    {

        printf("+%.4X", accumulator);

    }

    return 0;

}

int setBGColor(int ind)

{

    int x, y;

    getXY(&x, &y);

    if (ind == 1)

    {

        mt\_setbgcolor(BLUE);

        mt\_gotoXY(y + 2, (6 + 6 \* x) - 4);

        if (RAM[coord] < 0)

        {

            printf("-%.4X", -RAM[coord]);

        }

        else

        {

            printf("+%.4X", RAM[coord]);

        }

        printf("\E[0m");

    }

    else if (ind == 0)

    {

        mt\_setbgcolor(LIGHT\_GRAY);

        mt\_gotoXY(y + 2, (6 + 6 \* x) - 4);

        if (RAM[coord] < 0)

        {

            printf("-%.4X", -RAM[coord]);

        }

        else

        {

            printf("+%.4X", RAM[coord]);

        }

    }

    else

    {

        return -1;

    }

    return 0;

}

int resetBG()

{

    setBGColor(1);

    return 0;

}

int resetTerm()

{

    mt\_clrscr();

    printAll();

    printAccumulatorValue();

    printInstructionCounterValue();

    resetBG();

    printFlags();

    printOperation();

    printCaseBig();

    mt\_gotoXY(25, 1);

    for (int i = 0; i < 8; i++)

    {

        for (int i = 0; i < 83; i++)

        {

            printf(" ");

        }

        printf("\n");

    }

    mt\_gotoXY(23, 1);

    printf("Input/Output:\n");

    fflush(stdout);

    return 0;

}

void log\_console(const char \*msg)

{

    strcat(io\_msg, msg);

    strcat(io\_msg, "\n");

}

void read\_console\_value(int addr, int \*value)

{

    mt\_gotoXY(24, 1);

    printf("Enter: ");

    scanf("%X", value);

    char print[16];

    sprintf(print, "%d<\t%0X", addr, \*value);

    log\_console(print);

}

int write\_console\_value(int addr, int value)

{

    char print[16];

    int operand = 0, com = 0;

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

    int tmp = 0;

    sc\_memoryGet(operand, &tmp);

    sprintf(print, "%d>\t%0X", addr, tmp);

    log\_console(print);

    return tmp;

}

int run()

{

    enum keys key;

    resetTerm();

    sc\_memoryInit();

    sc\_regInit();

    accumulator = 0;

    coord = 0;

    signal(SIGUSR1, reset\_signalhagle);

    signal(SIGALRM, signalhangle);

    printAll();

    sc\_regSet(T, 1);

    while (1)

    {

        resetTerm();

        rk\_readkey(&key);

        int val;

        if (key == key\_q)

        {

            break;

        }

        else if (!sc\_regGet(T, &val) && val)

        {

            strcpy(io\_msg, "");

            switch (key)

            {

            case key\_l:

                keyLoad();

                break;

            case key\_s:

                keySave();

                break;

            case key\_r:

                sc\_regSet(T, 0);

                instructionCounter = 0;

                coord = 0;

                resetTerm();

                create\_timer(0.1);

                break;

            case key\_tt:

                Cu();

                break;

            case key\_i:

                create\_timer(0);

                raise(SIGUSR1);

                break;

            case key\_f5:

                keyF5();

                break;

            case key\_f6:

                keyF6();

                break;

            case key\_up:

                keyUp();

                break;

            case key\_down:

                keyDown();

                break;

            case key\_right:

                keyRight();

                break;

            case key\_left:

                keyLeft();

                break;

            default:

                key\_num(key);

                break;

            }

        }

    }

    rk\_mytermregime(1, 0, 1, 0, 0);

    return 0;

}

void keySave()

{

    char file[35] = {0};

    mt\_gotoXY(25, 1);

    printf("Input filename: ");

    scanf("%s", file);

    sc\_memorySave(file);

    resetTerm();

}

void keyLoad()

{

    char filename[64];

    mt\_gotoXY(24, 1);

    printf("Enter filename: ");

    rk\_mytermregime(0, 0, 1, 1, 1);

    fgets(filename, 63, stdin);

    filename[strlen(filename) - 1] = 0;

    char \*ptr1 = strrchr(filename, '.');

    if (ptr1 != NULL)

    {

        if (strcmp(ptr1, ".sa") == 0)

        {

            char \*ptr = NULL;

            int size = strlen(filename);

            ptr = malloc(sizeof(char) \* size);

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

                ptr[i] = filename[i];

            ptr[size - 1] = 'o';

            ptr[size] = '\0';

            asm\_to\_object(filename, ptr);

            sc\_memoryLoad(ptr);

            resetTerm();

        }

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

                ptr[i] = filename[i];

            ptr[size - 1] = 'a';

            ptr[size] = '\0';

            filename[size - 1] = 'o';

            filename[size] = '\0';

            asm\_to\_object(ptr, filename);

            sc\_memoryLoad(filename);

        }

        else if (strcmp(ptr1, ".so") == 0)

        {

            sc\_memoryLoad(filename);

            resetTerm();

        }

    }

}

void key\_num(enum keys key)

{

    int x, y;

    getXY(&x, &y);

    if (key > 0 && key < 10)

    {

        int value;

        sc\_memoryGet(coord, &value);

        if (key + value > 0x3FFF)

        {

            sc\_regSet(P, 1);

        }

        else

        {

            sc\_memorySet(coord, key + value);

        }

        resetTerm();

    }

}

void printOperation()

{

    int x, y;

    getXY(&x, &y);

    int value;

    sc\_memoryGet(coord, &value);

    mt\_gotoXY(8, 69);

    int command, operand;

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

    printf("+%.2X : %.2X", command, operand);

}

void keyF5()

{

    int value = 0;

    sc\_memoryGet(coord, &value);

    accumulator = value;

    resetTerm();

    fflush(stdout);

}

void keyF6()

{

    int value = 0;

    sc\_memoryGet(coord, &value);

    instructionCounter = coord;

    resetTerm();

    fflush(stdout);

}

void keyRight()

{

    int x, y;

    getXY(&x, &y);

    setBGColor(0);

    if (x != 9)

    {

        x++;

    }

    else if (x == 9 && y != 9)

    {

        x = 0;

        y++;

    }

    setBGColor(1);

    coord = y \* 10 + x;

    resetTerm();

}

void keyLeft()

{

    int x, y;

    getXY(&x, &y);

    setBGColor(0);

    if (x != 0)

    {

        x--;

    }

    else if (x == 0 && y != 0)

    {

        x = 9;

        y--;

    }

    setBGColor(1);

    coord = y \* 10 + x;

    resetTerm();

}

void keyUp()

{

    int x, y;

    getXY(&x, &y);

    if (y != 0)

    {

        setBGColor(0);

        y--;

        setBGColor(1);

    }

    coord = y \* 10 + x;

    resetTerm();

}

void keyDown()

{

    int x, y;

    getXY(&x, &y);

    if (y != 9)

    {

        setBGColor(0);

        y++;

        setBGColor(1);

    }

    coord = y \* 10 + x;

    resetTerm();

}

int printMemory()

{

    bc\_box(1, 1, BOX\_ROW\_MEMORY, BOX\_COLUMN\_MEMORY);

    mt\_gotoXY(1, 28);

    printf(" Memory ");

    for (int i = 0; i < 10; i++)

    {

        mt\_gotoXY(2 + i, 2);

        for (int j = 0; j < 10; j++)

        {

            if (j != 0)

                printf(" ");

            if (RAM[i \* 10 + j] < 0)

            {

                printf("-%.4X", -RAM[i \* 10 + j]);

            }

            else

            {

                printf("+%.4X", RAM[i \* 10 + j]);

            }

        }

    }

    printf("\n");

    return 0;

}

int printFlags()

{

    int reg\_p, reg\_o, reg\_m, reg\_t, reg\_e;

    char r\_P = ' ', r\_O = ' ', r\_M = ' ', r\_T = ' ', r\_E = ' ';

    sc\_regGet(P, &reg\_p);

    sc\_regGet(O, &reg\_o);

    sc\_regGet(M, &reg\_m);

    sc\_regGet(T, &reg\_t);

    sc\_regGet(E, &reg\_e);

    if (reg\_p == 1)

        r\_P = 'P';

    if (reg\_o == 1)

        r\_O = 'O';

    if (reg\_m == 1)

        r\_M = 'M';

    if (reg\_t == 1)

        r\_T = 'T';

    if (reg\_e == 1)

        r\_E = 'E';

    mt\_gotoXY(11, 64);

    printf("%c %c %c %c %c\n", r\_P, r\_O, r\_M, r\_T, r\_E);

    mt\_gotoXY(11, 62);

    bc\_box(10, BOX\_COLUMN\_MEMORY + 1, MINI\_BOX\_ROW, MINI\_BOX\_COLUMN);

    mt\_gotoXY(10, 69);

    printf(" Flags ");

    return 0;

}

int printAccumulator()

{

    bc\_box(1, BOX\_COLUMN\_MEMORY + 1, MINI\_BOX\_ROW, MINI\_BOX\_COLUMN);

    mt\_gotoXY(1, 67);

    printf(" accumulator ");

    return 0;

}

int printInstructionCounter()

{

    bc\_box(4, BOX\_COLUMN\_MEMORY + 1, MINI\_BOX\_ROW, MINI\_BOX\_COLUMN);

    mt\_gotoXY(4, 63);

    printf(" instructionCounter ");

    return 0;

}

int printOperationbox()

{

    bc\_box(7, BOX\_COLUMN\_MEMORY + 1, MINI\_BOX\_ROW, MINI\_BOX\_COLUMN);

    mt\_gotoXY(7, 68);

    printf(" Operation ");

    return 0;

}

int printKeys()

{

    int column = 37;

    int row = 10;

    bc\_box(13, 47, row, column);

    mt\_gotoXY(13, 48);

    printf(" Keys: ");

    mt\_gotoXY(14, 48);

    printf("l   - load");

    mt\_gotoXY(15, 48);

    printf("s   - save");

    mt\_gotoXY(16, 48);

    printf("r   - run");

    mt\_gotoXY(17, 48);

    printf("t   - step");

    mt\_gotoXY(18, 48);

    printf("i   - reset");

    mt\_gotoXY(19, 48);

    printf("F5  - accumulator");

    mt\_gotoXY(20, 48);

    printf("F6  - instructionCounter");

    mt\_gotoXY(21, 48);

    printf("q   - exit\n\n");

    return 0;

}

int printCaseBig()

{

    int value;

    int rank[4];

    int x, y;

    int column = 46;

    int row = 10;

    bc\_box(BOX\_ROW\_MEMORY + 1, 1, row, column);

    getXY(&x, &y);

    sc\_memoryGet(y \* 10 + x, &value);

    if (value < 0)

    {

        bc\_printbigchar(bcintm, 13, 2, BLACK, BLUE);

        value \*= -1;

    }

    else

    {

        bc\_printbigchar(bcintp, 13, 2, BLACK, BLUE);

    }

    for (int i = 0; i < 4; ++i)

    {

        rank[i] = value % 16;

        value /= 16;

    }

    for (int i = 38, j = 0; i >= 11; i -= 9, j++)

        printBigChars(rank[j], i);

    return 0;

}

int printBigChars(int val, int k)

{

    switch (val)

    {

    case 0:

        bc\_printbigchar(bcint0, 13, k, BLACK, BLUE);

        break;

    case 1:

        bc\_printbigchar(bcint1, 13, k, BLACK, BLUE);

        break;

    case 2:

        bc\_printbigchar(bcint2, 13, k, BLACK, BLUE);

        break;

    case 3:

        bc\_printbigchar(bcint3, 13, k, BLACK, BLUE);

        break;

    case 4:

        bc\_printbigchar(bcint4, 13, k, BLACK, BLUE);

        break;

    case 5:

        bc\_printbigchar(bcint5, 13, k, BLACK, BLUE);

        break;

    case 6:

        bc\_printbigchar(bcint6, 13, k, BLACK, BLUE);

        break;

    case 7:

        bc\_printbigchar(bcint7, 13, k, BLACK, BLUE);

        break;

    case 8:

        bc\_printbigchar(bcint8, 13, k, BLACK, BLUE);

        break;

    case 9:

        bc\_printbigchar(bcint9, 13, k, BLACK, BLUE);

        break;

    case 10:

        bc\_printbigchar(bcintA, 13, k, BLACK, BLUE);

        break;

    case 11:

        bc\_printbigchar(bcintB, 13, k, BLACK, BLUE);

        break;

    case 12:

        bc\_printbigchar(bcintC, 13, k, BLACK, BLUE);

        break;

    case 13:

        bc\_printbigchar(bcintD, 13, k, BLACK, BLUE);

        break;

    case 14:

        bc\_printbigchar(bcintE, 13, k, BLACK, BLUE);

        break;

    case 15:

        bc\_printbigchar(bcintF, 13, k, BLACK, BLUE);

        break;

    }

    return 0;

}

int printAll()

{

    printMemory();

    printInstructionCounter();

    printOperationbox();

    printAccumulator();

    printFlags();

    printKeys();

    printAccumulatorValue();

    printInstructionCounterValue();

    return 0;

}