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

Кафедра ВС

**КУРСОВАЯ РАБОТА**

По дисциплине «Архитектура ЭВМ»

Вариант 60

Выполнил:

Проверил:

Майданов Ю. С.

Новосибирск 2019

**Оглавление**

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

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

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

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

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

Листинг………………….......................................................................................15

**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.

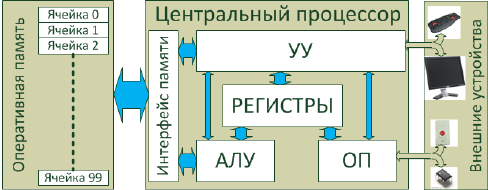


Рисунок 1 – Архитектура вычислительной машины Simple Computer

**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 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

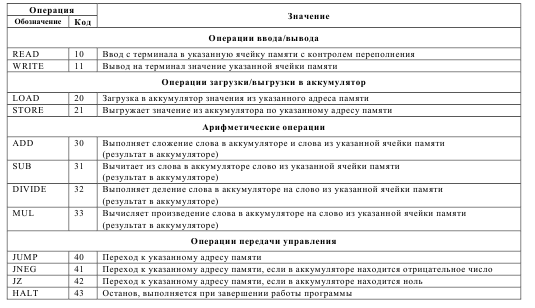


Рисунок 3 - Таблица команд центрального процессора Simple Computer

**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:

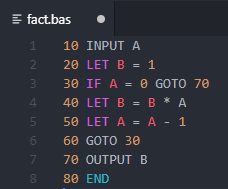
1. *Memory* – содержимое оперативной памяти *Simple Computer*.
2. *Accumulator* – значение, находящееся в аккумуляторе;
3. *instructionCounter* – значение регистра «счетчик команд»;
4. *Operation* – результат декодирования операции;
5. *Flags* – состояние регистра флагов («П» - переполнение при выполнении 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.

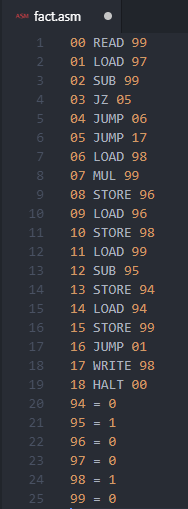
Содержимое ячеек памяти и регистров центрального процессора выводится в декодированном виде. При этом, знак «+» соответствует значению 0 в поле «признак команды», следующие две цифры – номер команды и затем операнд в шестнадцатеричной системе счисления.

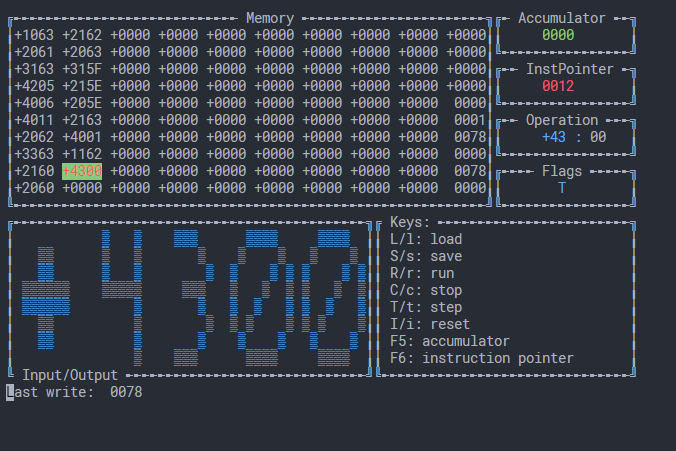
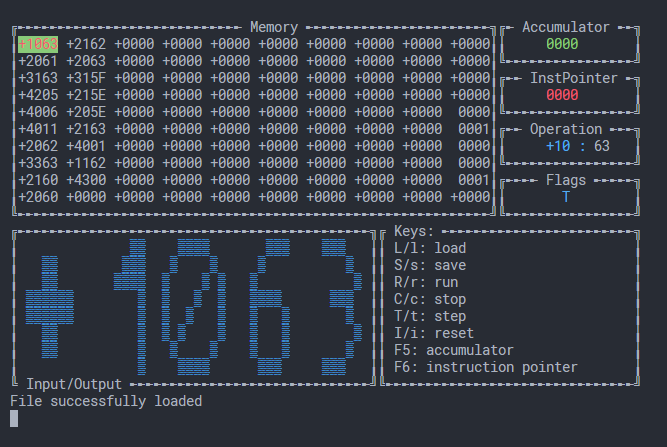
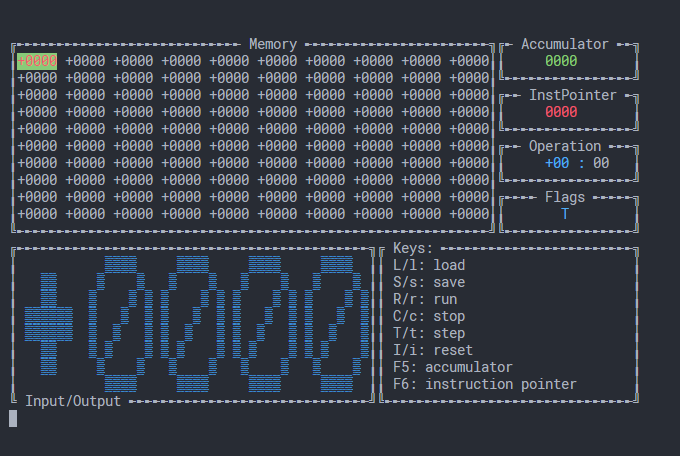
Пользователь имеет возможность с помощью клавиш управления курсора выбирать ячейки оперативной памяти и задавать им значения. Нажав клавишу *F5*, пользователь может задать значение аккумулятору, *F6* – регистру «счётчик команд». Сохранить содержимое памяти (в бинарном виде) в файл или загрузить его обратно 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 Basic*. Чтобы консоль управления смогла распознать высокоуровневый код, мы его транслируем в код *Assembler*, а после, в бинарный формат.





**Пример работы программы**

Была представлена работа поиска факториала из 5. Полученный ответ представлен в 16-тиричной системе счисления 0x007816 = 12010

**Заключение**

В ходе курсовой работы были реализованы базовые команды УУ и АЛУ. Реализована работа центрального процессора. Были написаны трансляторы с Basic в Assembler и Assembler в бинарный формат. Трансляторы были автоматизированы, 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**

**Function main  
main.c**  
#include "../lib/CPU/include/library.h"  
  
extern int accumulator;  
extern int instruction\_pointer;  
extern int writeValue;  
extern int bigChars[];  
extern int scan\_print\_radix;  
extern int position;  
extern int cursorX;  
extern int cursorY;  
  
struct itimerval val, oval;  
  
int main(int argc, char \*\*argv) {  
 sc\_memoryInit();  
 sc\_regInit();  
 cpu\_setSignals();  
 rk\_termSave();  
 sc\_regSet(FLAG\_INTERRUPT, 1);  
 position = 0;  
 int count\_write = 0, exit = 0, refresh\_flag = 0, mRegister = 0, pRegister = 0;  
 enum keys key = KEY\_OTHER;  
 int tmp\_chars[17 \* 2];  
 if (bc\_bigCharRead(open("chars.font", O\_RDONLY), tmp\_chars, 17, &count\_write))  
 bc\_bigCharWrite(open("chars.font", O\_WRONLY | O\_CREAT, S\_IRUSR | S\_IWUSR), (int\*)bigChars, 17);  
 else memcpy(bigChars, tmp\_chars, sizeof(tmp\_chars));  
 mt\_clrscr();  
 cpu\_refreshGuiState(position);  
 while (!exit) {  
 if (!refresh\_flag) {  
 cpu\_refreshGui(position);  
 refresh\_flag = 1;  
 }  
 rk\_readkey(&key);  
 sc\_regGet(FLAG\_INTERRUPT, &pRegister);  
 if (pRegister) {  
 switch (key) {  
 case KEY\_UP:  
 if (cursorY != 0) cursorY--;  
 else cursorY = 9;  
 refresh\_flag = 0;  
 key = KEY\_OTHER;  
 break;  
 case KEY\_DOWN:  
 cursorY = (cursorY + 1) % 10;  
 refresh\_flag = 0;  
 key = KEY\_OTHER;  
 break;  
 case KEY\_LEFT:  
 if (cursorX != 0) cursorX--;  
 else cursorX = 9;  
 refresh\_flag = 0;  
 key = KEY\_OTHER;  
 break;  
 case KEY\_RIGHT:  
 cursorX = (cursorX + 1) % 10;  
 refresh\_flag = 0;  
 key = KEY\_OTHER;  
 break;  
 case KEY\_F5:  
 refresh\_flag = cpu\_changeAccumulator(position);  
 key = KEY\_OTHER;  
 break;  
 case KEY\_F6:  
 refresh\_flag = cpu\_changeInstructionPointer(position);  
 key = KEY\_OTHER;  
 break;  
 case KEY\_ENTER:  
 refresh\_flag = cpu\_changeMemoryUnit(position);  
 key = KEY\_OTHER;  
 break;  
 case KEY\_T:  
 cpu\_CU();  
 cpu\_refreshGui(instruction\_pointer);  
 position = instruction\_pointer;  
 cursorX = instruction\_pointer / 10;  
 cursorY = instruction\_pointer % 10;  
 key = KEY\_OTHER;  
 break;  
 case KEY\_S:  
 cpu\_memorySave(position);  
 refresh\_flag = 1;  
 key = KEY\_OTHER;  
 break;  
 case KEY\_L:  
 cpu\_memoryLoad(position);  
 refresh\_flag = 1;  
 key = KEY\_OTHER;  
 break;  
 case KEY\_I:  
 raise(SIGUSR1);  
 refresh\_flag = 0;  
 cursorX = 0;  
 cursorY = 0;  
 key = KEY\_OTHER;  
 break;  
 case KEY\_Q:  
 exit = 1;  
 key = KEY\_OTHER;  
 break;  
 case KEY\_R:break;  
 case KEY\_X:break;  
 case KEY\_D:break;  
 case KEY\_C:break;  
 case KEY\_ESC:break;  
 case KEY\_OTHER:break;  
 }  
 }  
 if (key == KEY\_C) {  
 position = instruction\_pointer;  
 cursorX = instruction\_pointer / 10;  
 cursorY = instruction\_pointer % 10;  
 refresh\_flag = 0;  
 raise(SIGUSR2);  
 key = KEY\_OTHER;  
 } else if (key == KEY\_ESC) {  
 raise(SIGTERM);  
 key = KEY\_OTHER;  
 } else if (key == KEY\_R) {  
 key = KEY\_OTHER;  
 sc\_regGet(FLAG\_INTERRUPT, &mRegister);  
 if (mRegister) {  
 sc\_regInit();  
 cpu\_frequencyGenerator(START);  
 position = instruction\_pointer;  
 cursorX = instruction\_pointer / 10;  
 cursorY = instruction\_pointer % 10;  
 refresh\_flag = 0;  
 continue;  
 } else {  
 cpu\_frequencyGenerator(STOP);  
 sc\_regSet(FLAG\_INTERRUPT, 1);  
 position = instruction\_pointer;  
 cursorX = instruction\_pointer / 10;  
 cursorY = instruction\_pointer % 10;  
 refresh\_flag = 0;  
 continue;  
 }  
 }  
 position = cursorY + cursorX \* 10;  
 }  
 cpu\_frequencyGenerator(STOP);  
 rk\_termRestore();  
 cpu\_signalsRestore();  
 system("rm -f term\_temp");  
 return 0;  
}  
**libCPU.a  
library.h**  
#ifndef CPU\_LIBRARY\_H  
#define CPU\_LIBRARY\_H  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <assert.h>  
#include <signal.h>  
#include <fcntl.h>  
#include <time.h>  
#include <unistd.h>  
#include <sys/time.h>  
  
#include "./../../MemComReg/include/sc\_register.h"  
#include "./../../MemComReg/include/sc\_memory.h"  
#include "./../../MemComReg/include/sc\_command.h"  
#include "./../../Term/include/library.h"  
#include "./../../BigChars/include/library.h"  
#include "./../../ReadKey/include/library.h"  
#include "./../../Assets/include/library.h"  
#include "./../../TranslateAsm/include/library.h"  
#include "./../../TranslateBasic/include/library.h"  
  
/\* CPU \*/  
  
#ifndef TOKENSYMB  
#define TOKENSYMB ' '  
#endif  
  
#ifndef TOKENSYMBS  
#define TOKENSYMBS " "  
#endif  
  
#ifndef SIZE\_BUFFER  
#define SIZE\_BUFFER 256  
#endif  
  
#define TIMER ITIMER\_REAL  
#define REFRESH 100  
#define START 1  
#define STOP 0  
  
#define BIGCHARS\_COLOR\_FG mt\_cyan  
#define BIGCHARS\_COLOR\_BG mt\_default  
  
#define TEXT\_COLOR\_FG mt\_default  
#define TEXT\_COLOR\_BG mt\_green  
  
#define KEY\_COLOR\_FG mt\_default  
#define INST\_REG\_COLOR\_FG mt\_red  
#define REG\_COLOR\_FG mt\_blue  
#define COMMAND\_COLOR\_FG mt\_blue  
#define OPERAND\_COLOR\_FG mt\_light\_blue  
#define ACCUM\_COLOR\_FG mt\_green  
  
#define MKR(X) (X \* 1000000)  
#define TIME\_SLEEP\_USEC 0.05  
#define TIME\_SLEEP\_SEC 0  
  
void cpu\_CU();  
int cpu\_ALU(int command, int operand);  
  
void cpu\_frequencyGenerator(int status);  
  
void cpu\_refreshGuiState(int position);  
void cpu\_refreshGui(int position);  
void cpu\_printBoxes();  
void cpu\_printPointer();  
void cpu\_printAccumulator();  
void cpu\_printKeys(int x, int y);  
void cpu\_printLabels();  
void cpu\_printOperation(int position);  
void cpu\_printFlags(int x, int y);  
int cpu\_printMemoryUnit(int \*bigChars, int position);  
void cpu\_printLine(int size);  
void cpu\_printMemory(int x, int y, int position);  
  
void cpu\_setSignals();  
void cpu\_signalsRestore();  
void cpu\_setIgnoreAlarm();  
void cpu\_restoreIgnoreAlarm();  
void cpu\_setEchoMode();  
void cpu\_restoreEchoMode();  
  
int cpu\_changeAccumulator(int position);  
int cpu\_changeInstructionPointer(int position);  
int cpu\_changeMemoryUnit(int position);  
int scanNum(int \*flag, int \*n);  
  
int cpu\_memorySave(int position);  
int cpu\_memoryLoad(int position);  
  
static void cpu\_printWriteValue();  
  
int cpu\_commandHandler(int command, int operand);  
  
int cpu\_readMemoryUnit(int position);  
  
void cpu\_timerHand();  
void cpu\_urSignalHand();  
void cpu\_urSignalHand2();  
void cpu\_windHand();  
void cpu\_killHand();  
  
#endif //CPU\_LIBRARY\_H

**global.c**  
#include "library.h"  
  
int accumulator = 0;  
int instruction\_pointer = 0;  
int bigChars[34] = {  
 0x7e181800,  
 0x0018187e,//+  
  
 0x91A1423C,  
 0x3C428589,// 0  
  
 0x40787060,// 1  
 0x40404040,  
  
 0x20424438,  
 0x7e040810,// 2  
  
 0x3840201C,  
 0x1C204020,//3  
  
 0x7c444444,  
 0x40404040,//4  
  
 0x1C04047C,  
 0x1C204020,//5  
  
 0x3C040870,  
 0x38444444,//6  
  
 0x2020407C,  
 0x08081010,//7  
  
 0x38444438,  
 0x38444444,//8  
  
 0x78444438,  
 0x1C204040,//9  
  
 0x7e42423c,  
 0x42424242,// A  
  
 0x3E22221E,  
 0x3E424242,//B  
  
 0x02020478,  
 0x78040202,// C  
  
 0x4242221e,  
 0x1e224242,//D  
  
 0x3e02027e,  
 0x7e020202,//E  
  
 0x1e02027e,  
 0x02020202 //F  
};  
  
int writeUse = 0;  
int writeValue = 0;  
int scan\_print\_radix = 16;  
int readUse = 0;  
int position = 0;  
int cursorX = 0;  
int cursorY = 0;  
**change.c**  
#include "library.h"  
  
extern int accumulator;  
extern int instruction\_pointer;  
extern int scan\_print\_radix;  
static struct sigaction act[5];  
static struct sigaction old[5];  
static sigset\_t sigset;  
static struct sigaction new\_alarm;  
static struct sigaction old\_alarm;  
static struct termios basic\_term;  
static int echoIgn = 0;  
static int alarmIgn = 0;  
extern int readUse;  
static struct itimerval val;  
static struct itimerval oval;  
  
void cpu\_frequencyGenerator(int status) {  
 if (status == START) {  
 val.it\_interval.tv\_sec = TIME\_SLEEP\_SEC;  
 val.it\_interval.tv\_usec = MKR(TIME\_SLEEP\_USEC);  
 val.it\_value.tv\_sec = TIME\_SLEEP\_SEC;  
 val.it\_value.tv\_usec = MKR(TIME\_SLEEP\_USEC);  
 setitimer(TIMER, &val, &oval);  
 } else if (status == STOP) setitimer(TIMER, NULL, NULL);  
}  
  
void cpu\_setSignals() {  
 for (size\_t i = 0; i < 5; i++) {  
 memset(&(act[i]), 0, sizeof(act[i]));  
 memset(&(old[i]), 0, sizeof(old[i]));  
 }  
 sigemptyset(&sigset);  
 sigaddset(&sigset, SIGUSR1);  
 sigaddset(&sigset, SIGALRM);  
 act[0].sa\_handler = cpu\_timerHand;  
 act[0].sa\_mask = sigset;  
  
 sigemptyset(&sigset);  
 sigaddset(&sigset, SIGUSR1);  
 act[1].sa\_handler = cpu\_urSignalHand;  
 act[1].sa\_mask = sigset;  
  
 sigemptyset(&sigset);  
 sigaddset(&sigset, SIGUSR1);  
 sigaddset(&sigset, SIGUSR2);  
 act[2].sa\_handler = cpu\_urSignalHand2;  
 act[2].sa\_mask = sigset;  
  
 sigemptyset(&sigset);  
 sigaddset(&sigset, SIGUSR1);  
 sigaddset(&sigset, SIGWINCH);  
 act[3].sa\_handler = cpu\_windHand;  
 act[3].sa\_mask = sigset;  
  
 sigemptyset(&sigset);  
 sigaddset(&sigset, SIGUSR1);  
 sigaddset(&sigset, SIGTERM);  
 act[4].sa\_handler = cpu\_killHand;  
 act[4].sa\_mask = sigset;  
  
 sigaction(SIGALRM, &(act[0]), &(old[0]));  
 sigaction(SIGUSR1, &(act[1]), &(old[1]));  
 sigaction(SIGUSR2, &(act[2]), &(old[2]));  
 sigaction(SIGWINCH, &(act[3]), &(old[3]));  
 sigaction(SIGTERM, &(act[4]), &(old[4]));  
}  
  
void cpu\_signalsRestore() {  
 sigemptyset(&sigset);  
 sigaction(SIGALRM, &(old[0]), 0);  
 sigaction(SIGUSR1, &(old[1]), 0);  
 sigaction(SIGUSR2, &(old[2]), 0);  
 sigaction(SIGWINCH, &(old[3]), 0);  
 sigaction(SIGTERM, &(old[4]), 0);  
}  
  
void cpu\_setIgnoreAlarm() {  
 if (alarmIgn == 0) {  
 new\_alarm.sa\_handler = SIG\_IGN;  
 sigaction(SIGALRM, &(new\_alarm), &(old\_alarm));  
 alarmIgn = 1;  
 } else return;  
}  
  
void cpu\_restoreIgnoreAlarm() {  
 if (alarmIgn == 1) {  
 sigaction(SIGALRM, &(old\_alarm), 0);  
 alarmIgn = 0;  
 } else return;  
}  
  
void cpu\_setEchoMode() {  
 if (echoIgn == 0) {  
 while (tcgetattr(STDIN\_FILENO, &basic\_term) != 0);  
 rk\_termMode(0, 0, 1, 1, 1);  
 echoIgn = 1;  
 } else return;  
}  
  
void cpu\_restoreEchoMode() {  
 if (echoIgn == 1) {  
 while (tcsetattr(STDIN\_FILENO, TCSANOW, &basic\_term) != 0);  
 echoIgn = 0;  
 } else return;  
}  
  
int cpu\_changeMemoryUnit(int position) {  
 cpu\_setEchoMode();  
 if (readUse == 0) {  
 cpu\_setIgnoreAlarm();  
 cpu\_refreshGui(position);  
 mt\_gotoXY(1, 23);  
 cpu\_printLine(2);  
 mt\_gotoXY(1, 23);  
 ast\_writeChar(1, "Enter value: ");  
 }  
 int flag = 0, value = 0;  
 if (scanNum(&flag, &value) != 0) {  
 ast\_writeChar(2, "Not a number!");  
 if (readUse == 0) cpu\_restoreIgnoreAlarm();  
 cpu\_restoreEchoMode();  
 return -1;  
 }  
 if ((value >= 0) & (value < 0x8000)) {  
 int memory = 0;  
 if (flag) {  
 int command = (value >> 8) & 0x7F;  
 if (value & 0x80) {  
 ast\_writeChar(2, "Operand is 7 bit wide ( <= 7F). ");  
 if (readUse == 0) cpu\_restoreIgnoreAlarm();  
 cpu\_restoreEchoMode();  
 return -1;  
 }  
 int operand = value & 0x7F;  
 if (sc\_commandEncode(command, operand, &memory) != 0) {  
 ast\_writeChar(2, "Wrong command. Aborted. ");  
 if (readUse == 0) cpu\_restoreIgnoreAlarm();  
 cpu\_restoreEchoMode();  
 return -1;  
 }  
 } else {  
 if (value >= 0x4000) {  
 ast\_writeChar(1, "This number must be < 0x4000");  
 if (readUse == 0) cpu\_restoreIgnoreAlarm();  
 cpu\_restoreEchoMode();  
 return -1;  
 }  
 memory = value | 0x4000;  
 }  
 if (sc\_memorySet(position, memory) != 0) {  
 if (readUse == 0) cpu\_restoreIgnoreAlarm();  
 cpu\_restoreEchoMode();  
 return -1;  
 }  
 } else {  
 ast\_writeChar(2, "Memory cell is 15 bit wide");  
 if (readUse == 0) cpu\_restoreIgnoreAlarm();  
 cpu\_restoreEchoMode();  
 return -1;  
 }  
 if (readUse == 0) cpu\_restoreIgnoreAlarm();  
 cpu\_restoreEchoMode();  
 return 0;  
}  
  
int cpu\_changeAccumulator(int position) {  
 cpu\_setIgnoreAlarm();  
 cpu\_setEchoMode();  
 cpu\_refreshGui(position);  
 mt\_gotoXY(1, 23);  
 cpu\_printLine(2);  
 mt\_gotoXY(1, 23);  
 ast\_writeChar(1, "Change Accum: ");  
 int flag = 0, value = 0;  
 if (scanNum(&flag, &value) != 0) {  
 ast\_writeChar(2, "Not a number!");  
 cpu\_restoreEchoMode();  
 cpu\_restoreIgnoreAlarm();  
 return -1;  
 }  
 if ((value >= 0) & (value < 0x4000))  
 accumulator = value;  
 else {  
 ast\_writeChar(2, "Accumutalor is 15 bit wide");  
 cpu\_restoreEchoMode();  
 cpu\_restoreIgnoreAlarm();  
 return -1;  
 }  
 cpu\_restoreEchoMode();  
 cpu\_restoreIgnoreAlarm();  
 return 0;  
}  
  
int cpu\_changeInstructionPointer(int position) {  
 cpu\_setIgnoreAlarm();  
 cpu\_setEchoMode();  
 cpu\_refreshGui(position);  
 mt\_gotoXY(1, 23);  
 cpu\_printLine(2);  
 mt\_gotoXY(1, 23);  
 ast\_writeChar(1, "Change Pointer: ");  
 int flag = 0, value = 0;  
 if (scanNum(&flag, &value) != 0) {  
 ast\_writeChar(2, "Not a number!");  
 cpu\_restoreEchoMode();  
 cpu\_restoreIgnoreAlarm();  
 return -1;  
 }  
 if ((value >= 0) & (value < ram\_size))  
 instruction\_pointer = value;  
 else {  
 ast\_writeChar(2, "Instruction Pointer range: from 0 to 99 (0x63)");  
 cpu\_restoreEchoMode();  
 cpu\_restoreIgnoreAlarm();  
 return -1;  
 }  
 cpu\_restoreEchoMode();  
 cpu\_restoreIgnoreAlarm();  
 return 0;  
}  
  
int scanNum(int \*flag, int \*n) {  
 cpu\_setIgnoreAlarm();  
 cpu\_setEchoMode();  
 char buffer[SIZE\_BUFFER] = {0};  
 int position = 0, i = 0;  
 do {  
 read(0, &buffer[i++], 1);  
 } while (buffer[i - 1] != '\n');  
 buffer[i - 1] = '\0';  
 if (buffer[0] == '+') {  
 position = 1;  
 \*flag = 1;  
 } else {  
 position = 0;  
 \*flag = 0;  
 }  
 if (ast\_strReadInt(buffer + position, n, scan\_print\_radix) != 1) {  
 cpu\_restoreEchoMode();  
 cpu\_restoreIgnoreAlarm();  
 return -1;  
 }  
 cpu\_restoreEchoMode();  
 cpu\_restoreIgnoreAlarm();  
 return 0;  
}  
  
int cpu\_memorySave(int position) {  
 cpu\_setIgnoreAlarm();  
 cpu\_setEchoMode();  
 cpu\_refreshGui(position);  
 mt\_gotoXY(1, 23);  
 cpu\_printLine(2);  
 mt\_gotoXY(1, 23);  
 ast\_writeChar(1, "Enter save file name: ");  
 char filename[SIZE\_BUFFER] = {0};  
 int i = 0;  
 do {  
 read(0, &filename[i++], 1);  
 } while (filename[i - 1] != '\n');  
 filename[i - 1] = '\0';  
 if (sc\_memorySave(filename) == 0) {  
 cpu\_refreshGui(position);  
 ast\_writeChar(1, "File successfully saved.\n");  
 cpu\_restoreEchoMode();  
 cpu\_restoreIgnoreAlarm();  
 return 0;  
 } else {  
 ast\_writeChar(1, "Cannot save file: ");  
 ast\_writeChar(1, filename);  
 ast\_writeChar(1, "\n");  
 cpu\_restoreEchoMode();  
 cpu\_restoreIgnoreAlarm();  
 return -1;  
 }  
}  
  
int cpu\_memoryLoad(int position) {  
 cpu\_setIgnoreAlarm();  
 cpu\_setEchoMode();  
 cpu\_refreshGui(position);  
 mt\_gotoXY(1, 23);  
 cpu\_printLine(2);  
 mt\_gotoXY(1, 23);  
 ast\_writeChar(1, "Enter load file name: ");  
 char filename[SIZE\_BUFFER] = {0};  
 int i = 0;  
 do {  
 read(0, &filename[i++], 1);  
 } while (filename[i - 1] != '\n');  
 filename[i - 1] = '\0';  
 char \*strrchr\_ = strrchr(filename, '.');  
 int exception = 0;  
 if (strrchr\_ != NULL) {  
 if (strcmp(strrchr\_, ".asm") == 0) {  
 char \*string = NULL;  
 int size = strlen(filename);  
 string = malloc(sizeof(char) \* size);  
 for (i = 0; i < size; i++) string[i] = filename[i];  
 string[size] = '\0';  
 strrchr\_[1] = 'o';  
 strrchr\_[2] = '\0';  
 char \*args[3] = {NULL, filename, string};  
 if (asm\_trans(3, args) == 1) exception = 1; /\* 0, file.o file.asm \*/  
 free(string);  
 } else if (strcmp(strrchr\_, ".bas") == 0) {  
 char src[SIZE\_BUFFER];  
 char dest[SIZE\_BUFFER];  
 strcpy(src, filename); /\* file.bas \*/  
 strcpy(dest, filename);  
 char \*strchr\_ = strchr(dest, '.');  
 strchr\_[1] = 'a';  
 strchr\_[2] = 's';  
 strchr\_[3] = 'm';  
 strchr\_[4] = '\0';  
 char \*args[3] = {NULL, dest, src};  
 if (bas\_transAsm(src, dest) == 1) { /\* 0, file.asm file.bas \*/  
 exception = 1;  
 } else {  
 strchr\_ = strchr(src, '.');  
 strchr\_[1] = 'a';  
 strchr\_[2] = 's';  
 strchr\_[3] = 'm';  
 strchr\_[4] = '\0';  
 strchr\_ = strchr(dest, '.');  
 strchr\_[1] = 'o';  
 strchr\_[2] = '\0';  
 args[1] = dest;  
 args[2] = src;  
 if (asm\_trans(3, args) == 1) {  
 exception = 1; /\* 0, file.o file.asm \*/  
 } else {  
 strchr\_ = strchr(filename, '.');  
 strchr\_[1] = 'o';  
 strchr\_[2] = '\0';  
 }  
 }  
 }  
 }  
 if (!exception) {  
 if (sc\_memoryLoad(filename) == 0) {  
 cpu\_refreshGui(position);  
 ast\_writeChar(1, "File successfully loaded\n");  
 cpu\_restoreEchoMode();  
 cpu\_restoreIgnoreAlarm();  
 return 0;  
 } else {  
 ast\_writeChar(1, "Cannot load file: ");  
 ast\_writeChar(1, filename);  
 ast\_writeChar(1, "\n");  
 cpu\_restoreEchoMode();  
 cpu\_restoreIgnoreAlarm();  
 return -1;  
 }  
 } else if (exception) {  
 ast\_writeChar(1, "Cannot load file: ");  
 ast\_writeChar(1, filename);  
 ast\_writeChar(1, "\n");  
 cpu\_restoreEchoMode();  
 cpu\_restoreIgnoreAlarm();  
 return -1;  
 }  
}  
**command.c**  
#include "library.h"  
  
extern int accumulator;  
extern int instruction\_pointer;  
extern int writeUse;  
extern int writeValue;  
extern int position;  
extern int cursorX;  
extern int cursorY;  
  
int cpu\_commandHandler(int command, int operand) {  
 int temp;  
 int temp\_memory;  
 switch (command) {  
 case 0x10: /\* READ \*/  
 while (cpu\_readMemoryUnit(operand) != 0);  
 break;  
 case 0x11: /\* WRITE \*/  
 writeUse = 1;  
 if (sc\_memoryGet(operand, &writeValue) != 0) return -1;  
 break;  
 case 0x20: /\* LOAD \*/  
 if (sc\_memoryGet(operand, &accumulator) != 0) return -1;  
 accumulator &= 0x3FFF;  
 break;  
 case 0x21: /\* STORE \*/  
 temp = accumulator | (1 << 14);  
 if (sc\_memorySet(operand, temp) != 0) return -1;  
 break;  
 case 0x30: /\* ADD \*/  
 if (sc\_memoryGet(operand, &temp\_memory) != 0) return -1;  
 temp\_memory &= 0x3FFF;  
 if (accumulator + temp\_memory > 0x3FFF) {  
 sc\_regSet(FLAG\_OVERFLOW, 1);  
 accumulator += temp\_memory;  
 accumulator &= 0x3FFF;  
 } else accumulator += temp\_memory;  
 break;  
 case 0x31: /\* SUB \*/  
 if (sc\_memoryGet(operand, &temp\_memory) != 0) return -1;  
 temp\_memory &= 0x3FFF;  
 accumulator -= temp\_memory;  
 break;  
 case 0x32: /\* DIVIDE \*/  
 if (sc\_memoryGet(operand, &temp\_memory) != 0) return -1;  
 temp\_memory &= 0x3FFF;  
 if (!(temp\_memory == 0)) accumulator = accumulator / temp\_memory;  
 else {  
 sc\_regSet(FLAG\_DIVISION, 1);  
 return -1;  
 }  
 break;  
 case 0x33: /\* MUL \*/  
 if (sc\_memoryGet(operand, &temp\_memory) != 0) return -1;  
 accumulator = accumulator \* (temp\_memory & 0x3FFF);  
 break;  
 case 0x40: /\* JUMP \*/  
 instruction\_pointer = operand - 1;  
 break;  
 case 0x41: /\* JNEG \*/  
 if (((accumulator >> 14) & 1) == 1) instruction\_pointer = operand - 1;  
 break;  
 case 0x42: /\* JZ \*/  
 if (accumulator == 0) instruction\_pointer = operand - 1;  
 break;  
 case 0x43: /\* HALT \*/  
 sc\_regSet(FLAG\_INTERRUPT, 1);  
 instruction\_pointer--;  
 position = instruction\_pointer;  
 cursorX = instruction\_pointer / 10;  
 cursorY = instruction\_pointer % 10;  
 cpu\_refreshGui(instruction\_pointer);  
 break;  
 case 0x69: /\* RCCL \*/  
 if (sc\_memoryGet(operand, &temp\_memory) != 0)  
 return -1;  
 if (temp\_memory) {  
 if (sc\_memoryGet(accumulator, &temp) != 0)  
 return -1;  
 for (size\_t i = 0; i < temp; i++)  
 temp\_memory = (temp\_memory << 1) & ((temp\_memory >> 14) & 1);  
 accumulator = temp\_memory;  
 accumulator &= 0x7FFF;  
 } else {  
 accumulator = temp\_memory;  
 accumulator &= 0x7FFF;  
 }  
 break;  
 case 0x70: /\* RCCR \*/  
 if (sc\_memoryGet(operand, &temp\_memory) != 0)  
 return -1;  
 if (temp\_memory) {  
 if (sc\_memoryGet(accumulator, &temp) != 0)  
 return -1;  
 for (size\_t i = 0; i < temp; i++)  
 temp\_memory = (temp\_memory >> 1) & ((temp\_memory & 1) << 14);  
 accumulator = temp\_memory;  
 accumulator &= 0x3FFF;  
 } else {  
 accumulator = temp\_memory;  
 accumulator &= 0x3FFF;  
 }  
 break;  
 }  
 return 0;  
}

**cpu.c**  
#include "library.h"  
  
extern int accumulator;  
extern int instruction\_pointer;  
  
void cpu\_CU() {  
 if (instruction\_pointer >= ram\_size) {  
 sc\_regSet(FLAG\_OUTMEM, 1);  
 sc\_regSet(FLAG\_INTERRUPT, 1);  
 cpu\_refreshGui(instruction\_pointer);  
 instruction\_pointer = 0;  
 return;  
 }  
 int command, operand, memory;  
 sc\_memoryGet(instruction\_pointer, &memory);  
 if (sc\_commandDecode(memory, &command, &operand) != 0) {  
 sc\_regSet(FLAG\_COMMAND, 1);  
 sc\_regSet(FLAG\_INTERRUPT, 1);  
 return;  
 }  
 if ((operand < 0) | (operand >= ram\_size)) {  
 sc\_regSet(FLAG\_COMMAND, 1);  
 sc\_regSet(FLAG\_INTERRUPT, 1);  
 return;  
 }  
 if ((command >= 0x30) & (command <= 0x33) |  
 (command >= 0x69) & (command <= 0x70)) {  
 if (cpu\_ALU(command, operand) != 0)  
 sc\_regSet(FLAG\_INTERRUPT, 1);  
 } else cpu\_commandHandler(command, operand);  
 instruction\_pointer++;  
}  
  
int cpu\_ALU(int command, int operand) {  
 if ((operand < 0) | (operand >= ram\_size)) {  
 sc\_regSet(FLAG\_OUTMEM, 1);  
 return ERR\_WRONG\_ADDR;  
 } else {  
 if (cpu\_commandHandler(command, operand) != 0) return -1;  
 if ((accumulator > 0x3FFF)) {  
 accumulator &= 0x3FFF;  
 sc\_regSet(FLAG\_OVERFLOW, 1);  
 } else sc\_regSet(FLAG\_OVERFLOW, 0);  
 sc\_regSet(FLAG\_ODD, accumulator & 1);  
 return 0;  
 }  
}  
**handlers.c**  
#include "library.h"  
  
extern int accumulator;  
extern int instruction\_pointer;  
extern int writeUse;  
  
void cpu\_timerHand() {  
 int mRegister;  
 sc\_regGet(FLAG\_INTERRUPT, &mRegister);  
 if (mRegister == 0) {  
 cpu\_CU();  
 cpu\_refreshGui(instruction\_pointer);  
 }  
}  
  
void cpu\_urSignalHand() {  
 sc\_memoryInit();  
 sc\_regInit();  
 sc\_regSet(FLAG\_INTERRUPT, 1);  
 accumulator = 0;  
 writeUse = 0;  
 instruction\_pointer = 0;  
}  
  
void cpu\_urSignalHand2() {  
 setitimer(TIMER, NULL, NULL);  
 sc\_regSet(FLAG\_INTERRUPT, 1);  
}  
  
void cpu\_windHand() {  
 cpu\_refreshGuiState(0);  
}  
  
void cpu\_killHand() {  
 setitimer(TIMER, NULL, NULL);  
 mt\_clrscr();  
 rk\_termRestore();  
 cpu\_signalsRestore();  
 system("rm -f term\_temp");  
 exit(0);  
}

**read.c**  
#include "library.h"  
  
extern int instruction\_pointer;  
extern int readUse;  
  
int cpu\_readMemoryUnit(int position) {  
 cpu\_setEchoMode();  
 cpu\_refreshGui(position);  
 mt\_gotoXY(1, 23);  
 cpu\_printLine(3);  
 mt\_gotoXY(1, 23);  
 ast\_writeChar(1, "READ: ");  
 mt\_gotoXY(1, 24);  
 readUse = 1;  
 int callback = cpu\_changeMemoryUnit(position);  
 readUse = 0;  
 cpu\_restoreEchoMode();  
 return callback;  
}  
**ui.c**  
#include "library.h"  
  
extern int accumulator;  
extern int instruction\_pointer;  
extern int bigChars[];  
extern int writeValue;  
extern int writeUse;  
extern int scan\_print\_radix;  
static int counterRefresh = 0;  
  
static void cpu\_printWriteValue() {  
 mt\_gotoXY(1, 23);  
 cpu\_printLine(3);  
 mt\_gotoXY(1, 23);  
 int command = (writeValue >> 14) & 1;  
 if (writeUse != 0) {  
 ast\_writeChar(1, "Last write: ");  
 if (command == 0) {  
 int operand;  
 int operand\_code;  
 sc\_commandDecode(writeValue, &operand\_code, &operand);  
 ast\_writeChar(1, "+");  
 ast\_writeInt(1, operand\_code, 16, 2);  
 ast\_writeInt(1, operand, 16, 2);  
 } else {  
 ast\_writeChar(1, " ");  
 ast\_writeInt(1, writeValue & 0x3FFF, scan\_print\_radix, 4);  
 }  
 }  
}  
  
void cpu\_printLine(int size) {  
 int x = 0, y = 0;  
 mt\_getScreenSize(&y, &x);  
 for (size\_t j = 0; j < size; j++) {  
 for (size\_t i = 0; i < x; i++)  
 ast\_writeChar(1, " ");  
 ast\_writeChar(1, "\n");  
 }  
}  
  
void cpu\_refreshGuiState(int position) {  
 mt\_clrscr();  
 cpu\_printBoxes();  
 cpu\_printLabels();  
 cpu\_printKeys(48, 14);  
 cpu\_printPointer();  
 cpu\_printAccumulator();  
 cpu\_printOperation(position);  
 cpu\_printMemory(2, 2, position);  
 cpu\_printFlags(66, 11);  
 cpu\_printWriteValue();  
 mt\_gotoXY(1, 23);  
}  
  
void cpu\_refreshGui(int position) {  
 counterRefresh++;  
 if (counterRefresh % REFRESH == 0) {  
 cpu\_refreshGuiState(position);  
 counterRefresh = 0;  
 } else {  
 cpu\_printPointer();  
 cpu\_printAccumulator();  
 cpu\_printOperation(position);  
 cpu\_printMemory(2, 2, position);  
 cpu\_printFlags(66, 11);  
 cpu\_printWriteValue();  
 mt\_gotoXY(1, 23);  
 }  
}  
  
void cpu\_printAccumulator() {  
 mt\_setFgColor(ACCUM\_COLOR\_FG);  
 mt\_gotoXY(68, 2);  
 ast\_writeChar(1, " ");  
 mt\_gotoXY(68, 2);  
 if (scan\_print\_radix == 16)ast\_writeInt(1, accumulator & 0x3FFF, scan\_print\_radix, 4);  
 else if (scan\_print\_radix == 10) ast\_writeInt(1, accumulator & 0x3FFF, scan\_print\_radix, 5);  
 mt\_setFgColor(mt\_default);  
}  
  
void cpu\_printBoxes() {  
 bc\_box(1, 1, 61, 12);  
 bc\_box(62, 1, 79, 3);  
 bc\_box(62, 4, 79, 6);  
 bc\_box(62, 7, 79, 9);  
 bc\_box(62, 10, 79, 12);  
 bc\_box(1, 13, 46, 22);  
 bc\_box(47, 13, 79, 22);  
}  
  
void cpu\_printPointer() {  
 mt\_gotoXY(68, 5);  
 mt\_setFgColor(INST\_REG\_COLOR\_FG);  
 ast\_writeInt(1, instruction\_pointer, scan\_print\_radix, 4);  
 mt\_setFgColor(mt\_default);  
}  
  
void cpu\_printKeys(int x, int y) {  
 mt\_setFgColor(KEY\_COLOR\_FG);  
 mt\_gotoXY(x, y);  
 ast\_writeChar(1, " L/l: load");  
 mt\_gotoXY(x, y + 1);  
 ast\_writeChar(1, " S/s: save");  
 mt\_gotoXY(x, y + 2);  
 ast\_writeChar(1, " R/r: run ");  
 mt\_gotoXY(x, y + 3);  
 ast\_writeChar(1, " C/c: stop ");  
 mt\_gotoXY(x, y + 4);  
 ast\_writeChar(1, " T/t: step");  
 mt\_gotoXY(x, y + 5);  
 ast\_writeChar(1, " I/i: reset");  
 mt\_gotoXY(x, y + 6);  
 ast\_writeChar(1, " F5: accumulator");  
 mt\_gotoXY(x, y + 7);  
 ast\_writeChar(1, " F6: instruction pointer");  
 mt\_setFgColor(mt\_default);  
}  
  
void cpu\_printLabels() {  
 mt\_setFgColor(KEY\_COLOR\_FG);  
 mt\_gotoXY(30, 1);  
 ast\_writeChar(1, " Memory ");  
 mt\_gotoXY(64, 1);  
 ast\_writeChar(1, " Accumulator ");  
 mt\_gotoXY(65, 4);  
 ast\_writeChar(1, " InstPointer ");  
 mt\_gotoXY(65, 7);  
 ast\_writeChar(1, " Operation ");  
 mt\_gotoXY(67, 10);  
 ast\_writeChar(1, " Flags ");  
 mt\_gotoXY(48, 13);  
 ast\_writeChar(1, " Keys: ");  
 mt\_gotoXY(2, 22);  
 ast\_writeChar(1, " Input/Output ");  
 mt\_setFgColor(mt\_default);  
}  
  
void cpu\_printOperation(int position) {  
 if ((position >= ram\_size) & (position <= 0)) {  
 sc\_regSet(FLAG\_OUTMEM, 1);  
 return;  
 }  
 int memory;  
 sc\_memoryGet(position, &memory);  
 int isCommand = (memory >> 14) & 1;  
 mt\_gotoXY(68, 8);  
 if ((position >= 0) & (position < ram\_size)) {  
 if (isCommand == 0) {  
 int command, operand;  
 sc\_commandDecode(memory, &command, &operand);  
 ast\_writeChar(1, " ");  
 mt\_gotoXY(68, 8);  
 mt\_setFgColor(COMMAND\_COLOR\_FG);  
 char c = '+';  
 write(1, &c, 1);  
 ast\_writeInt(1, command, 16, 2);  
 ast\_writeChar(1, " : ");  
 mt\_setFgColor(OPERAND\_COLOR\_FG);  
 ast\_writeInt(1, operand, 16, 2);  
 mt\_setFgColor(mt\_default);  
 } else {  
 mt\_setFgColor(COMMAND\_COLOR\_FG);  
 ast\_writeChar(1, " ");  
 mt\_gotoXY(68, 8);  
 ast\_writeInt(1, memory & 0x3FFF, 16, 4);  
 mt\_setFgColor(mt\_default);  
 }  
 }  
}  
  
void cpu\_printFlags(int x, int y) {  
 int mRegister;  
 mt\_gotoXY(x, y);  
 sc\_regGet(FLAG\_OVERFLOW, &mRegister);  
 if (mRegister == 1) {  
 mt\_setFgColor(REG\_COLOR\_FG);  
 ast\_writeChar(1, "O");  
 mt\_setFgColor(mt\_default);  
 } else ast\_writeChar(1, " ");  
 ast\_writeChar(1, " ");  
 sc\_regGet(FLAG\_COMMAND, &mRegister);  
 if (mRegister == 1) {  
 mt\_setFgColor(REG\_COLOR\_FG);  
 ast\_writeChar(1, "E");  
 mt\_setFgColor(mt\_default);  
 } else ast\_writeChar(1, " ");  
 ast\_writeChar(1, " ");  
 sc\_regGet(FLAG\_INTERRUPT, &mRegister);  
 if (mRegister == 1) {  
 mt\_setFgColor(REG\_COLOR\_FG);  
 ast\_writeChar(1, "T");  
 mt\_setFgColor(mt\_default);  
 } else ast\_writeChar(1, " ");  
 ast\_writeChar(1, " ");  
 sc\_regGet(FLAG\_OUTMEM, &mRegister);  
 if (mRegister == 1) {  
 mt\_setFgColor(REG\_COLOR\_FG);  
 ast\_writeChar(1, "M");  
 mt\_setFgColor(mt\_default);  
 } else ast\_writeChar(1, " ");  
 ast\_writeChar(1, " ");  
 sc\_regGet(FLAG\_DIVISION, &mRegister);  
 if (mRegister == 1) {  
 mt\_setFgColor(REG\_COLOR\_FG);  
 ast\_writeChar(1, "Z");  
 mt\_setFgColor(mt\_default);  
 } else ast\_writeChar(1, " ");  
}  
  
int cpu\_printMemoryUnit(int \*bigChars, int position) {  
 int memory;  
 if ((position >= ram\_size) & (position < 0)) {  
 sc\_regSet(FLAG\_OUTMEM, 1);  
 return 1;  
 }  
 sc\_memoryGet(position, &memory);  
 char str[10] = {0};  
 int command = (memory >> 14) & 1;  
 memory &= 0x3FFF;  
 if (command == 0) {  
 int operand\_code, operand;  
 sc\_commandDecode(memory, &operand\_code, &operand);  
 str[0] = '+';  
 ast\_strWriteInt(str + 1, operand\_code, 16, 2);  
 ast\_strWriteInt(str + 3, operand, 16, 2);  
 } else {  
 str[0] = ' ';  
 ast\_strWriteInt(str + 1, memory, 16, 4);  
 }  
 int indexBigChar;  
 for (size\_t i = 0; i < 5; i++) {  
 mt\_gotoXY(2 + 9 \* i, 14);  
 indexBigChar = (str[i] == '+')  
 ? 0 : (((str[i] >= '0') && (str[i] <= '9'))  
 ? (str[i] - '0' + 1) : ((str[i] >= 'A') && (str[i] <= 'F'))  
 ? (str[i] - 'A' + 11) : -1);  
 bc\_printBigChar(bigChars + (indexBigChar \* 2), 2 + i \* 9, 14, BIGCHARS\_COLOR\_FG, BIGCHARS\_COLOR\_BG);  
 }  
 return 0;  
}  
  
void cpu\_printMemory(int x, int y, int position) {  
 if ((position >= ram\_size) & (position < 0)) {  
 sc\_regSet(FLAG\_OUTMEM, 1);  
 return;  
 }  
 int memory, command, operand\_code, operand;  
 for (size\_t i = 0; i < 10; i++) {  
 mt\_gotoXY(x, y + i);  
 for (size\_t j = 0; j < 10; j++) {  
 sc\_memoryGet(i + j \* 10, &memory);  
 command = (memory >> 14) & 1;  
 memory &= 0x3FFF;  
 if ((i + j \* 10) == position) {  
 mt\_setFgColor(TEXT\_COLOR\_FG);  
 mt\_setBgColor(TEXT\_COLOR\_BG);  
 }  
 if ((i + j \* 10) == instruction\_pointer) mt\_setFgColor(INST\_REG\_COLOR\_FG);  
 if (command == 0) {  
 sc\_commandDecode(memory, &operand\_code, &operand);  
 ast\_writeChar(1, "+");  
 ast\_writeInt(1, operand\_code, 16, 2);  
 ast\_writeInt(1, operand, 16, 2);  
 } else if (command == 1) {  
 ast\_writeChar(1, " ");  
 ast\_writeInt(1, memory, 16, 4);  
 }  
 if ((i + j \* 10) == position) {  
 mt\_setFgColor(mt\_default);  
 mt\_setBgColor(mt\_default);  
 }  
 if ((i + j \* 10) == instruction\_pointer) mt\_setFgColor(mt\_default);  
 if (j != 9) ast\_writeChar(1, " ");  
 }  
 }  
 cpu\_printMemoryUnit(bigChars, position);  
}

**libTranslateAsm.a  
library.h**  
#ifndef TRANSLATEASM\_LIBRARY\_H  
#define TRANSLATEASM\_LIBRARY\_H  
  
#include "./../../CPU/include/library.h"  
  
int asm\_trans(int argc, char \*\*argv);  
int asm\_commands(char \*string);  
int asm\_parsingLine(char \*string, int \*address, int \*value);  
int asm\_strToCommand(char \*string, int \*value);  
int asm\_testArgv(char \*\*argv);  
  
#endif //TRANSLATEASM\_LIBRARY\_H

**asm.c**  
#include "library.h"  
  
int asm\_trans(int argc, char \*\*argv) {  
 if (asm\_testArgv(argv) == 1) return 1;  
 int input = 0, output = 0, counter\_tokens = 0, flag\_ign = 0, i = 0;  
 if ((input = open(argv[2], O\_RDONLY)) == -1) return 1;  
 if ((output = open(argv[1], O\_WRONLY | O\_CREAT | O\_TRUNC, 0666)) == -1) return 1;  
 int ram[ram\_size] = {0};  
 for (i = 0; i < ram\_size; i++) ram[i] = 0;  
 char buffer[SIZE\_BUFFER] = {0};  
 int address = 0, value = 0, callback = 0;  
 i = 0;  
 do {  
 value = 0;  
 address = 0;  
 int read\_count = read(input, &buffer[i], sizeof(char));  
 if (read\_count == 0) break;  
 if (buffer[i] == TOKENSYMB)  
 if (buffer[i - 1] == TOKENSYMB) continue;  
 if (buffer[i] == ' ')  
 if (buffer[i - 1] == ' ') continue;  
 if (buffer[i] == EOF & flag\_ign == 0) {  
 if (counter\_tokens == 2) {  
 buffer[i] = '\0';  
 callback = asm\_parsingLine(buffer, &address, &value);  
 if (!callback) ram[address] = value;  
 }  
 break;  
 }  
 if (buffer[i] == '\n' & flag\_ign == 1) {  
 flag\_ign = 0;  
 i = 0;  
 counter\_tokens = 0;  
 continue;  
 }  
 if (flag\_ign == 0) {  
 if (buffer[i] == TOKENSYMB & counter\_tokens < 2) {  
 counter\_tokens++;  
 if (counter\_tokens == 2) i++;  
 }  
 if (counter\_tokens == 2 &  
 (buffer[i] == ';' | buffer[i] == '\n' | buffer[i] == ' ' | buffer[i] == TOKENSYMB)) {  
 if (buffer[i] == ';' | buffer[i] == ' ' | buffer[i] == TOKENSYMB) flag\_ign = 1;  
 buffer[i] = '\0';  
 callback = asm\_parsingLine(buffer, &address, &value);  
 if (callback == 0 | callback == 2) ram[address] = value;  
 counter\_tokens = 0;  
 i = 0;  
 }  
 i++;  
 }  
 } while (1);  
 write(output, ram, sizeof(int) \* ram\_size);  
 close(input);  
 close(output);  
 return 0;  
}  
  
int asm\_testArgv(char \*\*argv) {  
 if ((strcmp(strrchr(argv[1], '.'), ".o") != 0) ||  
 (strcmp(strrchr(argv[2], '.'), ".asm") != 0))  
 return 1;  
 else return 0;  
}  
  
int asm\_commands(char \*string) {  
 if (strcmp(string, "READ") == 0)  
 return 0x10;  
 else if (strcmp(string, "WRITE") == 0)  
 return 0x11;  
 else if (strcmp(string, "LOAD") == 0)  
 return 0x20;  
 else if (strcmp(string, "STORE") == 0)  
 return 0x21;  
 else if (strcmp(string, "ADD") == 0)  
 return 0x30;  
 else if (strcmp(string, "SUB") == 0)  
 return 0x31;  
 else if (strcmp(string, "DIVIDE") == 0)  
 return 0x32;  
 else if (strcmp(string, "MUL") == 0)  
 return 0x33;  
 else if (strcmp(string, "JUMP") == 0)  
 return 0x40;  
 else if (strcmp(string, "JNEG") == 0)  
 return 0x41;  
 else if (strcmp(string, "JZ") == 0)  
 return 0x42;  
 else if (strcmp(string, "HALT") == 0)  
 return 0x43;  
 else if (strcmp(string, "RCCL") == 0)  
 return 0x69;  
 else if (strcmp(string, "RCCR") == 0)  
 return 0x70;  
 else  
 return -1;  
}  
  
int asm\_parsingLine(char \*string, int \*address, int \*value) {  
 char \*string1 = string;  
 char \*string2 = string1;  
 int i = 0;  
 while (1) {  
 if (string2[i] == TOKENSYMB) {  
 string1 = string2 + i + 1;  
 string2[i] = '\0';  
 break;  
 }  
 i++;  
 }  
 ast\_strReadInt(string2, address, 10);  
 if (\*address < 0 | \*address > ram\_size) return -1;  
 int command = 0, operand = 0, flag\_command = 0;  
 i = 0;  
 string2 = string1;  
 if (\*string1 == '=') flag\_command = 1;  
 if (flag\_command) {  
 while (1) {  
 if (string2[i] == TOKENSYMB) {  
 string1 = string2 + i + 1;  
 string2[i] = '\0';  
 break;  
 }  
 i++;  
 }  
 string2 = string1 + 1;  
 if (\*string2 == '+') {  
 char tmp = string2[3];  
 string2[3] = '\0';  
 ast\_strReadInt(string2 + 1, &command, 16);  
 string2[3] = tmp;  
 string2 = &string2[3];  
 tmp = string2[2];  
 string2[2] = '\0';  
 ast\_strReadInt(string2, &operand, 16);  
 string2[2] = tmp;  
 sc\_commandEncode(command, operand, value);  
 \*value &= 0x7FFF;  
 } else {  
 ast\_strReadInt(string2, value, 16);  
 \*value |= (1 << 14);  
 \*value &= 0x7FFF;  
 }  
 } else {  
 while (1) {  
 if (string2[i] == TOKENSYMB) {  
 string1 = string2 + i + 1;  
 string2[i] = '\0';  
 break;  
 }  
 i++;  
 }  
 command = asm\_commands(string2);  
 if (command == -1) return -1;  
 string2 = string1 + 1;  
 if (ast\_strReadInt(string2, &operand, 10) == -1) return -1;  
 sc\_commandEncode(command, operand, value);  
 }  
 if (command == 0x43) return 2;  
 return 0;  
}  
**libTranslateBasic.a**  
**library.h**  
#ifndef TRANSLATEBASIC\_LIBRARY\_H  
#define TRANSLATEBASIC\_LIBRARY\_H  
  
#include "./../../Assets/include/library.h"  
#include "./../../CPU/include/library.h"  
#include <malloc.h>  
  
struct stack\_t {  
 char c;  
 struct stack\_t \*next;  
};  
  
struct stack\_t \*bas\_stackPush(struct stack\_t \*head, char a);  
char bas\_stackPop(struct stack\_t \*\*head);  
int bas\_priority(char c);  
void bas\_translateToRpn(char \*outstr, char \*a);  
  
int bas\_transAsm(const char \*filename\_bas, const char \*filename\_asm);  
  
#endif //TRANSLATEBASIC\_LIBRARY\_H  
**basic.c**  
  
#include "library.h"  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <ctype.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;  
};  
  
struct stack\_t \*bas\_stackPush(struct stack\_t \*head, char a) {  
 struct stack\_t \*pStack;  
 if ((pStack = malloc(sizeof(struct stack\_t))) == NULL) {  
 perror("Out of memory\n");  
 exit(1);  
 }  
 pStack->c = a;  
 pStack->next = head;  
 return pStack;  
}  
  
char bas\_stackPop(struct stack\_t \*\*head) {  
 if (\*head == NULL) return '\0';  
 struct stack\_t \*ptr = \*head;  
 char a = ptr->c;  
 \*head = ptr->next;  
 free(ptr);  
 return a;  
}  
  
int bas\_priority(char c) {  
 switch (c) {  
 case '\*':  
 case '/':  
 return 3;  
 case '-':  
 case '+':  
 return 2;  
 case '(':  
 return 1;  
 default:  
 break;  
 }  
}  
  
void bas\_translateToRpn(char \*outstr, char \*a) {  
 struct stack\_t \*pStack = NULL;  
 int k = 0, point = 0;  
 while (a[k] != '\0' && a[k] != '\n') {  
 if (a[k] == ')') {  
 while ((pStack->c) != '(')  
 outstr[point++] = bas\_stackPop(&pStack);  
 bas\_stackPop(&pStack);  
 }  
 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] == '(')  
 pStack = bas\_stackPush(pStack, '(');  
 else if (a[k] == '+' || a[k] == '-' || a[k] == '/' || a[k] == '\*') {  
 if (pStack == NULL)  
 pStack = bas\_stackPush(pStack, a[k]);  
 else if (bas\_priority(pStack->c) < bas\_priority(a[k]))  
 pStack = bas\_stackPush(pStack, a[k]);  
 else {  
 while ((pStack != NULL) && (bas\_priority(pStack->c) >= bas\_priority(a[k])))  
 outstr[point++] = bas\_stackPop(&pStack);  
 pStack = bas\_stackPush(pStack, a[k]);  
 }  
 }  
 k++;  
 }  
 while (pStack != NULL)  
 outstr[point++] = bas\_stackPop(&pStack);  
 outstr[point] = '\0';  
}  
  
int bas\_commandType(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 bas\_variableId(const struct stVariables \*vars, unsigned max, char var) {  
 for (unsigned id = 0; 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 = bas\_variableId(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 bas\_transAsm(const char \*filename\_bas, const char \*filename\_asm) {  
 FILE \*pIoFile = fopen(filename\_bas, "r");  
 if (!pIoFile) 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';  
 while (end == 0 && fscanf(pIoFile, "%u %[A-Z] ", &line, buffer) != 0) {  
 if (line\_id != 0 && line <= lines[line\_id].line\_number) {  
 end = fail = 1;  
 break;  
 }  
 int type = bas\_commandType(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(pIoFile, "%[A-Z]", buffer);  
 if (buffer[1] != 0 || !(buffer[0] >= 'A' && buffer[0] <= 'Z')) {  
 end = fail = 1;  
 break;  
 }  
 unsigned id = bas\_variableId(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(pIoFile, "%[A-Z]", buffer);  
 if (buffer[1] != 0 || !(buffer[0] >= 'A' && buffer[0] <= 'Z')) {  
 end = fail = 1;  
 break;  
 }  
 unsigned id = bas\_variableId(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(pIoFile, "%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(pIoFile, "%[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(pIoFile, "%[A-Z]", buffer);  
 type = bas\_commandType(buffer);  
 goto type\_check;  
 }  
 int id\_ = var\_id, id\_\_ = var\_id;  
 if (isdigit(op1[0])) {  
 var[id\_].name = tmp\_var;  
 var[id\_].address = 99 - var\_id;  
 var[id\_].init\_value = atoi(op1);  
 ++var\_id;  
 ++tmp\_var;  
 id\_\_ = bas\_variableId(var, var\_id, op2[0]);  
 if (id\_\_ == var\_id) {  
 var[id\_\_].name = op2[0];  
 var[id\_\_].address = 99 - var\_id;  
 var[id\_\_].init\_value = 0;  
 ++var\_id;  
 }  
 } else if (isdigit(op2[0])) {  
 var[id\_\_].name = tmp\_var;  
 var[id\_\_].address = 99 - var\_id;  
 var[id\_\_].init\_value = atoi(op2);  
 ++var\_id;  
 ++tmp\_var;  
 id\_ = bas\_variableId(var, var\_id, op1[0]);  
 if (id\_ == var\_id) {  
 var[id\_].name = op1[0];  
 var[id\_].address = 99 - var\_id;  
 var[id\_].init\_value = 0;  
 ++var\_id;  
 }  
 } else {  
 id\_ = bas\_variableId(var, var\_id, op1[0]);  
 if (id\_ == var\_id) {  
 var[id\_].name = op1[0];  
 var[id\_].address = 99 - var\_id;  
 var[id\_].init\_value = 0;  
 ++var\_id;  
 }  
 id\_\_ = bas\_variableId(var, var\_id, op2[0]);  
 if (id\_\_ == var\_id) {  
 var[id\_\_].name = op2[0];  
 var[id\_\_].address = 99 - var\_id;  
 var[id\_\_].init\_value = 0;  
 ++var\_id;  
 }  
 }  
 switch (buffer[0]) {  
 case '<':  
 sprintf(buffer, "%02u LOAD %02u\n%02u SUB %02u\n%02u JNEG 00\n", address,  
 var[id\_\_].address, address + 1, var[id\_].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[id\_].address, address + 1, var[id\_\_].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:  
 sprintf(buffer, "%02u LOAD %02u\n%02u SUB %02u\n%02u JZ %02u\n%02u JUMP 00\n", address,  
 var[id\_\_].address, address + 1, var[id\_].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(pIoFile, " %[A-Z] ", buffer);  
 type = bas\_commandType(buffer);  
 goto type\_check;  
 }  
 case 6: {  
 fscanf(pIoFile, "%[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, pIoFile);  
 bas\_translateToRpn(rpn, exp);  
 if (strlen(rpn) > 1) {  
 char stack[100] = "\0";  
 int position = 0;  
 for (size\_t i = 0; rpn[i]; i++) {  
 if (position > 1 && isalnum(rpn[i]) == 0) {  
 unsigned id;  
 char var\_ = stack[position - 2];  
 getVarID(id, var\_)  
 sprintf(buffer, "%02u LOAD %02u\n", address, var[id].address);  
 strcat(asm\_code, buffer);  
 ++address;  
 char var\_\_ = stack[position - 1];  
 getVarID(id, var\_\_)  
 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;  
 position--;  
 stack[position - 1] = tmp\_var - 1;  
 } else {  
 stack[position] = rpn[i];  
 position++;  
 }  
 }  
 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;  
 if (isdigit(rpn[0])) {  
 id = var\_id;  
 ++var\_id;  
 if (bas\_variableId(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 = bas\_variableId(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 = bas\_variableId(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(pIoFile); }  
 while (ignore != '\n' && ignore != EOF);  
 if (ignore == EOF) break;  
 }  
 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;  
}