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

Кафедра вычислительных
систем

КУРСОВАЯ РАБОТА

по дисциплине «Архитектура ЭВМ»

Выполнил:
студент группы ИВ-222
Рудцких Владислав
Евгеньевич

Проверил:
доцент кафедры вычислительных
систем:
Соловьев Александр Сергеевич

Новосибирск, 2024

Оглавление

Постановка задачи

1) Разработать транслятор с языка Simple Basic. Итог работы транслятора – бинарный файл с образом оперативной памяти Simple Computer, который можно загрузить в модель и выполнить;

2) Доработать модель Simple Computer – реализовать алгоритм работы блока «L1-кэш команд и данных» и модифицировать работу контроллера оперативной памяти и обработчика прерываний таким образом, чтобы учитывался простой процессора при прямом доступе к оперативной памяти;

3) Разработать транслятор с языка Simple Basic. Итог работы транслятора – текстовый файл с программой на языке Simple Basic.

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 A)
01 READ 10 ; (Ввод B)
02 LOAD 09 ; (Загрузка A в аккумулятор)
03 SUB 10 ; (Отнять B)
04 JNEG 07 ; (Переход на 07, если отрицательное)
05 WRITE 09 ; (Вывод A)
06 HALT 00 ; (Останов)
07 WRITE 10 ; (Вывод B)
08 HALT 00 ; (Останов)
09 = +0000 ; (Переменная A)
10 = +9999 ; (Переменная B)
```

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.

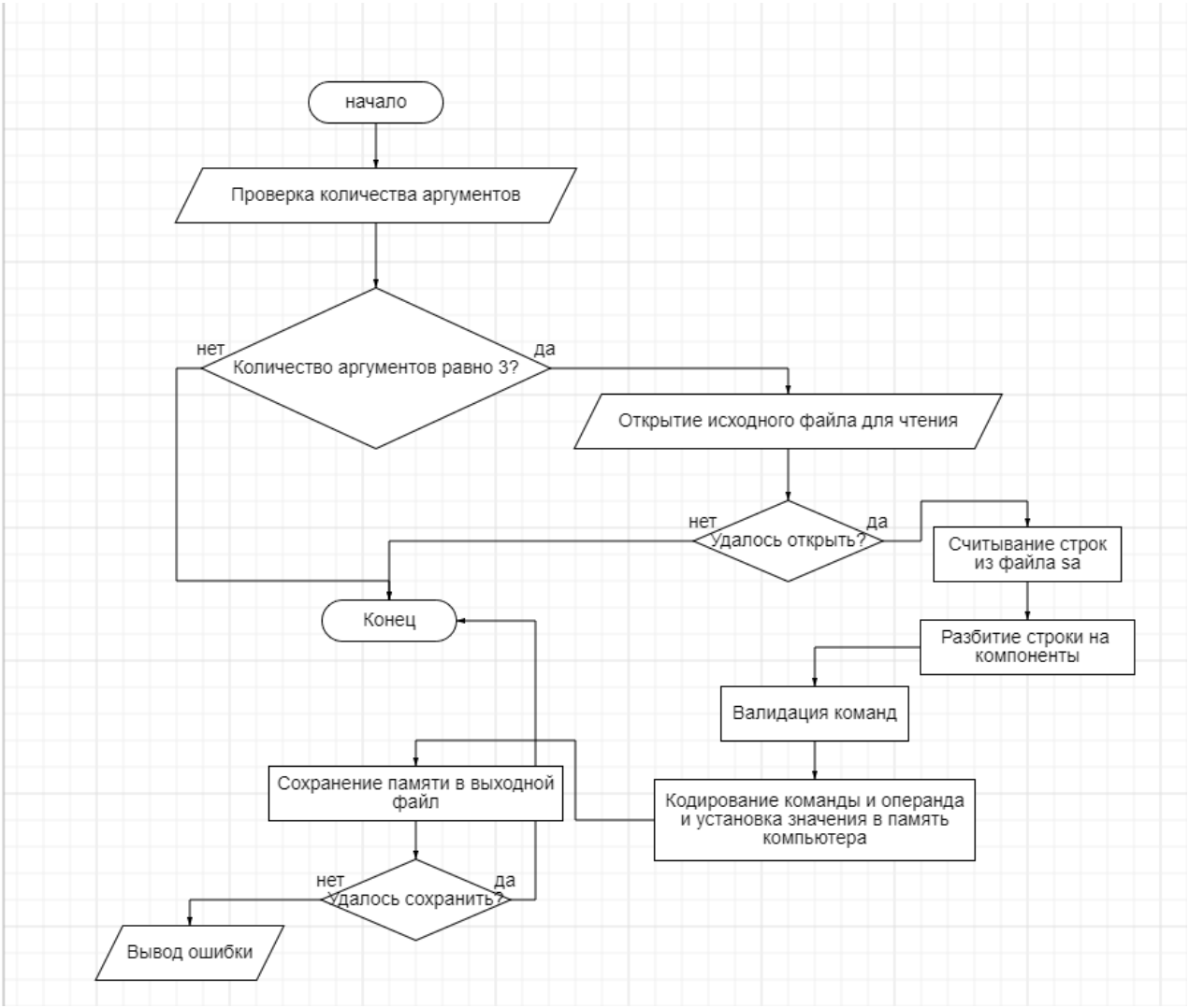
Номера строк должны следовать в возрастающем порядке. Все команды за исключением команды конца программы могут встречаться в программе многократно. Simple Basic должен оперировать с целыми выражениями, включающими операции +, -, *, и /. Приоритет операций аналогичен C. Для того чтобы изменить порядок 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

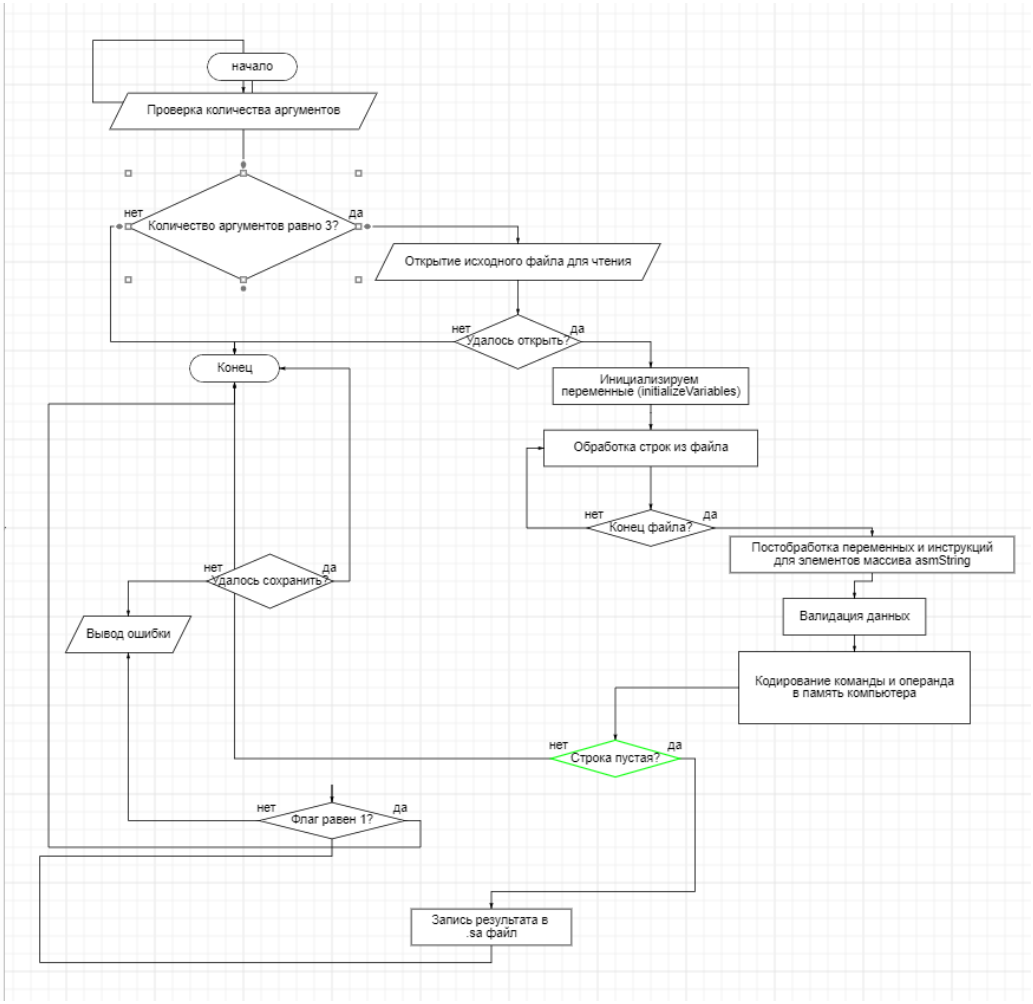
только с целыми значениями переменных, в нем отсутствует объявление переменных, а упоминание переменной автоматически вызывает её объявление и присваивает ей нулевое значение. Синтаксис языка не позволяет выполнять операций со строками.

Блок-схемы использованных алгоритмов

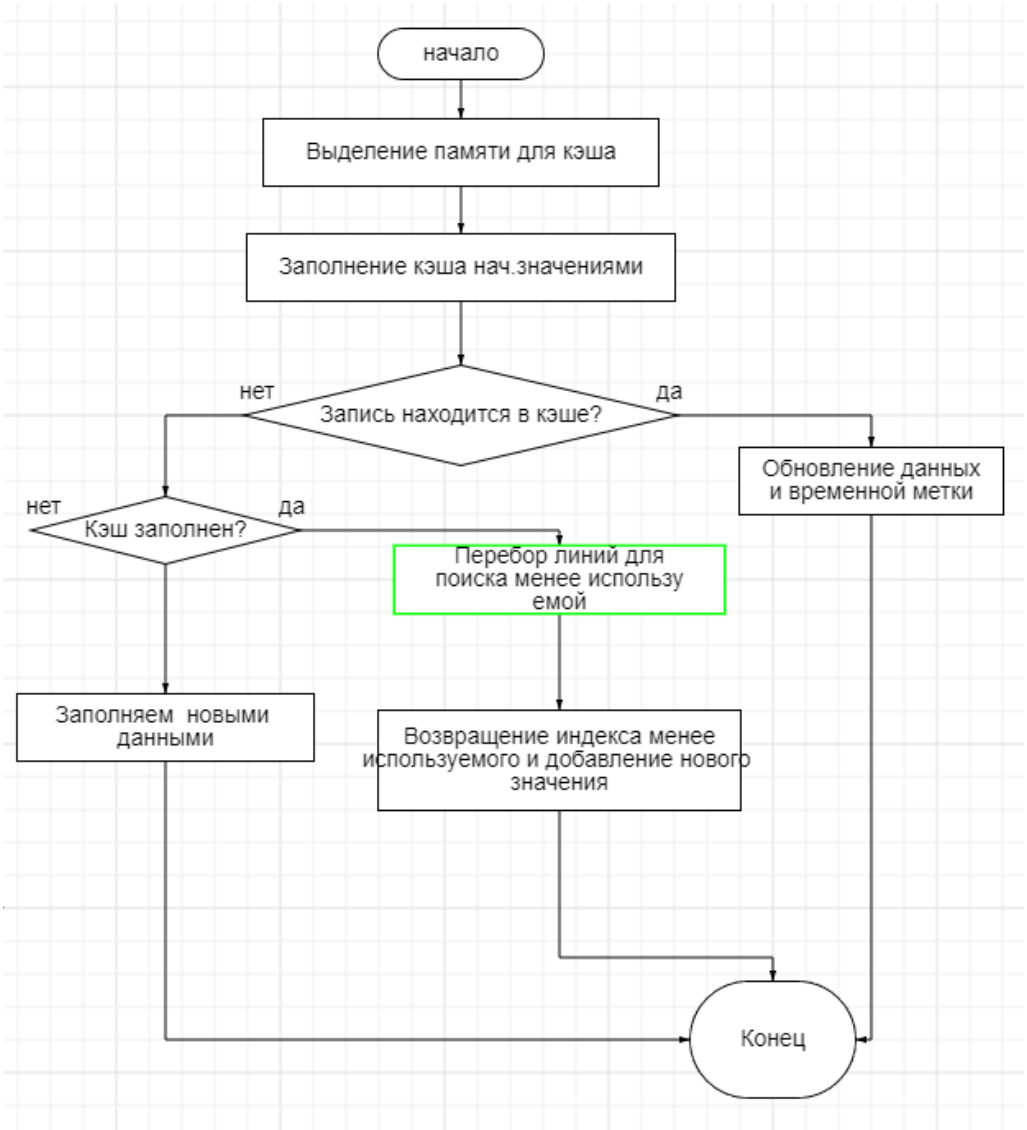
SimpleAssembler:



SimpleBasic:



Кеш процессора:



Программная реализация

Кеш процессора:

sc_initCache ():

Функция инициализирует кэш, выделяя память под массив структуру и устанавливает начальные значения для каждой структуры, такие как адрес и время последнего доступа

sc_findLeastRecentlyUsedCacheEntry ():

Функция проходит через все записи в кэше и ищет индекс наименее недавно использованной записи, основываясь на времени их последнего доступа

sc_updateCache():

updateCacheLine:

Функция обновляет содержимое одной строки кэша. Она копирует данные из памяти, начиная с указанного адреса address, в соответствующую строку кэша, и обновляет адрес этой строки кэша и время её последнего доступа.

sc_updateCacheAfterSave:

Функция обновляет кэш, учитывая адреса в памяти и их соответствие строкам в кэше, основываясь на том, есть ли уже запись в кэше для данного адреса, и либо добавляя новую запись, либо обновляя существующую.

Simple Assembler:

Main.c:

isValidRange():

Функция проверяет, находятся ли переданные адрес и значение в допустимом диапазоне, возвращая true, если оба соответствуют установленным ограничениям, иначе возвращает false.

memorySet():

Функция устанавливает значение по указанному адресу в массиве памяти. Одновременно она проверяет, находятся ли переданный адрес и значение в допустимом диапазоне при помощи функции isValidRange(), и если нет, устанавливает соответствующий флаг ошибки и выводит сообщение о выходе за пределы допустимого диапазона.

commandValidate():

Функция проверяет, является ли переданная строка командой, и если да, возвращает её соответствующий код. Для этого она сравнивает переданную команду с массивом известных команд и их кодов, возвращая соответствующий код, если команда найдена. Если переданная строка начинается с символа '=', она интерпретируется как команда для установки значения, возвращая код 1. Если переданная команда не совпадает ни с одной известной командой, функция возвращает -1.

main():

Функция обрабатывает исходный файл с программой на ассемблере, выполняет ассемблирование инструкций и сохраняет их в память. Он открывает и проверяет наличие входного файла, читает каждую строку из файла, разбирает её на адрес, команду и операнд, затем проверяет корректность команды и сохраняет её в память. По завершении чтения файла он сохраняет результат в выходной файл и выводит сообщение об успешном завершении.

Simple Basic:

sb_clearBackspace:

Функция удаляет все пробелы из строки str. Она использует два индекса - read_index для чтения символов из исходной строки и write_index для записи символов в строку без пробелов. Пока не достигнут конец строки, она проверяет каждый символ: если символ не является пробелом, он копируется в выходную строку, иначе он пропускается. По завершении процесса она добавляет нулевой символ в конец выходной строки для обозначения её конца.

sb_ commandValidate ():

Функция проверяет, является ли переданная строка command командой Simple BASIC и возвращает её соответствующий код. Для этого она сравнивает переданную команду с массивом известных команд и их кодов, возвращая код команды, если команда найдена. Если переданная строка начинается с символа '=', она интерпретируется как команда для установки значения, возвращая код 1. Если переданная команда не совпадает ни с одной из известных команд, функция возвращает -1.

sb_getGotoAddr () :

Функция ищет адрес строки, на которую должен осуществиться переход после команды "GOTO" с указанным номером gotoLine. Функция проходит по каждой строке в массиве asmString, копирует её для безопасной работы с содержимым, затем использует функцию strtok() для разделения строки на токены по пробелам. Затем она преобразует каждый токен в целое число и сравнивает его с gotoLine. Если токен соответствует gotoLine, функция берет следующий токен, который должен содержать адрес, и возвращает его значение в качестве адреса, на который нужно перейти. Если совпадение не найдено, функция возвращает -1.

sb_findOrCreateVariableAddress ():

Функция ищет переменную с заданным именем varName в массиве variables. Если переменная уже существует, функция возвращает её адрес, начиная с startOffset. Если переменная не найдена, функция создает новую переменную с указанным именем и возвращает её адрес, также начиная с startOffset. Если в массиве variables нет места для новой переменной, функция возвращает -1.

sb_getVariableName ():

Функция возвращает имя переменной, хранящейся по указанному адресу variableAddress, с учетом смещения nameOffset. Если переменная существует и её имя не пустое, функция возвращает первый символ имени переменной. Если переменная не найдена или её имя пустое, функция возвращает нулевой символ.

sb_handleEnd ():

Функция обрабатывает команду "END" в Simple BASIC. Она проверяет синтаксис строки команды, используя функцию sscanf(), чтобы считать номер строки, операцию и дополнительный текст. Если количество успешно считанных элементов не равно 2, функция выводит сообщение об ошибке и завершает свою работу. Если синтаксис верен, функция формирует строку для записи в массив asmString, содержащую номер строки, номер операции, адрес и команду "HALT".

Sb_ handleGoto ():

Функция обрабатывает команду "GOTO" в Simple BASIC. Она считывает номер строки, операцию и номер строки для перехода из строки команды. Если количество успешно считанных элементов не равно 3, функция выводит сообщение об ошибке и завершает работу. Если синтаксис верен, функция формирует строку для записи в массив asmString, содержащую номер текущей строки, номер операции, адрес, команду "JUMP" и номер строки для перехода, завершая строку символом новой строки.

Sb_handleIf ():

Функция обрабатывает команду "IF" в Simple BASIC. Она извлекает условие из строки команды и проверяет его синтаксис. Если синтаксис верен, функция формирует соответствующие инструкции для загрузки значения переменной и выполнения условного перехода, добавляя их в массив asmString. Если условие неверное, функция выводит сообщение об ошибке.

Sb_handleInput ():

Функция обрабатывает команду "INPUT" в Simple BASIC. Она считывает номер строки, операцию и переменную, в которую будет сохранено введенное значение, из строки команды. Если количество успешно считанных элементов не равно 3, функция выводит сообщение об ошибке и завершает работу. Если синтаксис верен, функция формирует строку для записи в массив asmString, содержащую номер текущей строки, номер операции, адрес, команду "READ" и переменную, куда будет сохранено введенное значение, завершая строку символом новой строки.

Sb_handleLet():

Функция обрабатывает команду "LET" в Simple BASIC. Она считывает выражение из строки команды и вычисляет его значение. Затем она генерирует соответствующие инструкции ассемблера для выполнения вычисления и сохранения результата в указанную переменную. Если синтаксис команды неверный, функция выводит сообщение об ошибке

Sb_handlePrint():

Функция обрабатывает команду "PRINT" в Simple BASIC. Она считывает номер строки, операцию и переменную, которую нужно вывести, из строки команды. Если количество успешно считанных элементов не равно 3, функция выводит сообщение об ошибке и завершает работу. Если синтаксис верен, функция формирует строку для записи в массив asmString, содержащую номер текущей строки, номер операции, адрес, команду "WRITE" и переменную, которую нужно вывести, завершая строку символом новой строки.

sb_memorySet ():

Функция предназначена для установки значения в ячейку памяти Simple Computer. Если адрес или значение находятся вне допустимого диапазона, то функция устанавливает соответствующий флаг регистра SC и выводит сообщение об ошибке. В противном случае значение устанавливается в указанную ячейку памяти, и функция возвращает 0.

Sb_postprocessJumps ():

Функция выполняет постобработку инструкций перехода (например, JUMP, JNEG, JNS, JZ) в ассемблерном коде. Она извлекает информацию о метке перехода из строки ассемблера и находит соответствующий адрес этой метки. Если синтаксис строки ассемблера неверен или адрес метки не найден, функция выводит сообщение об ошибке. В противном случае она формирует строку для вывода в выходной файл, содержащую адрес перехода и операцию перехода, а затем сохраняет эту строку в массиве.

sb_ postprocessCommands ():

Функция выполняет постобработку инструкций команд (например, READ, WRITE, LOAD, STORE, ADD, SUB, DIVIDE, MUL) в ассемблерном коде. Она извлекает информацию из строки ассемблера, включая номер строки, номер операции, адрес и операцию. Затем она находит или создает адрес переменной в памяти для операнда команды и формирует строку для вывода в выходной файл, содержащую адрес переменной в памяти и операцию команды. Если команда является операцией загрузки (LOAD) и имеет значение, она также сохраняет это значение в массиве numArray, используя адрес переменной в памяти в качестве индекса.

sb_ postprocessHalt ():

Функция выполняет постобработку инструкции HALT в ассемблерном коде. Она извлекает информацию из строки ассемблера, включая номер строки, номер операции и адрес. Затем она формирует строку для вывода в выходной файл, содержащую адрес и операцию HALT, с последующим добавлением нулевого операнда.

sb_basicToAssembler():

initializeVariables ():

Функция инициализирует массивы строк asmString, asmStringOut и variables, обнуляя их содержимое. Для этого она проходит по каждому элементу каждого массива и устанавливает каждой строке нулевой символ.

dispatchInstruction ():

Эта функция выполняет обработку инструкций, идентифицируя команды и вызывая соответствующие функции для их выполнения. Например, если операнд opr соответствует

команде "REM", функция просто увеличивает номер строки linenum и возвращает ноль. Если операнд соответствует команде "END", вызывается функция handleEnd(). Если операнд не соответствует ни одной из известных команд, функция выводит сообщение об ошибке и возвращает -1.

Postprocess ():

Функция выполняет последующую обработку строк ассемблерного кода после их разбора и преобразования в соответствующие значения. Она анализирует операнд каждой строки и, в зависимости от команды, вызывает соответствующие функции для дополнительной обработки

processLineFromFile ():

Функция обрабатывает каждую строку из файла. Она считывает строку из файла, проверяет, достигнут ли конец файла, и, если да, возвращает 0. Если строка не является пустой строкой, она разбирает строку, считывая номер строки и операнд, и передает их функции dispatchInstruction() для дальнейшей обработки. После успешной обработки строки функция инкрементирует значение asm_addr, представляющее адрес текущей инструкции, и обновляет значение last_strnum для проверки порядка строк. Если строка имеет неверный синтаксис, функция выводит сообщение об ошибке и возвращает -1.

compileBasic ():

Функция компилирует базовый код из файла, применяя последовательность шагов. Сначала инициализируются переменные с помощью функции initializeVariables(). Затем происходит обработка строк файла в цикле с вызовом функции processLineFromFile(), увеличивая linenum на каждой итерации. После завершения обработки файла закрывается, сохраняется максимальный номер строки и определяется смещение переменных. Далее происходит постобработка инструкций и переменных в цикле с вызовом функции postprocess() или postprocessVariables(). Результат записывается в файл file_sa, а затем происходит его чтение и проверка. В случае успешной компиляции функция возвращает 0, иначе возвращает -1. Если установлен флаг, процесс завершается.

Main ():

Функция, которая компилирует файлы, написанные на языке SimpleBasic. Входные файлы именуются в соответствии с их расширениями: file.sb для исходного кода SimpleBasic и file.sa для сгенерированного ассемблерного кода. Если передано неправильное количество аргументов командной строки, программа выводит сообщение об использовании. Если не удается открыть входной или создать выходной файл, программа сообщает об ошибке. После успешного завершения компиляции файлы закрываются, и программа завершается с кодом возврата 0.

Результаты проведённого исследования

После написания проекта необходимо его протестировать, в качестве теста было предложено написать программу для нахождения факториала. Код программы на языке SimpleBasic:

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

Результат трансляции с SimpleBasic на SimpleAssembler:

```
❯ ./sb factorial.sb factorial.sa
Basic file 'factorial.sb' is successfully opened.
10 INPUT A
20 LET B = 1
30 LET C = 1
40 LET B = A * B
50 LET A = A - C
60 IF A > 0 GOTO 40
70 PRINT B
80 END
```

factorial.sa

simplebasic > factorial.sa

```
1   00 READ 19
2   01 LOAD 20
3   02 STORE 21
4   04 LOAD 20
5   05 STORE 22
6   07 LOAD 19
7   08 MUL 21
8   09 STORE 21
9   11 LOAD 19
10  12 SUB 22
11  13 STORE 19
12  15 LOAD 19
13  16 JNS 07
14  17 WRITE 21
15  18 HALT 00
16  19 = +0000
17  20 = +0001
18  21 = +0000
19  22 = +0000
20
```

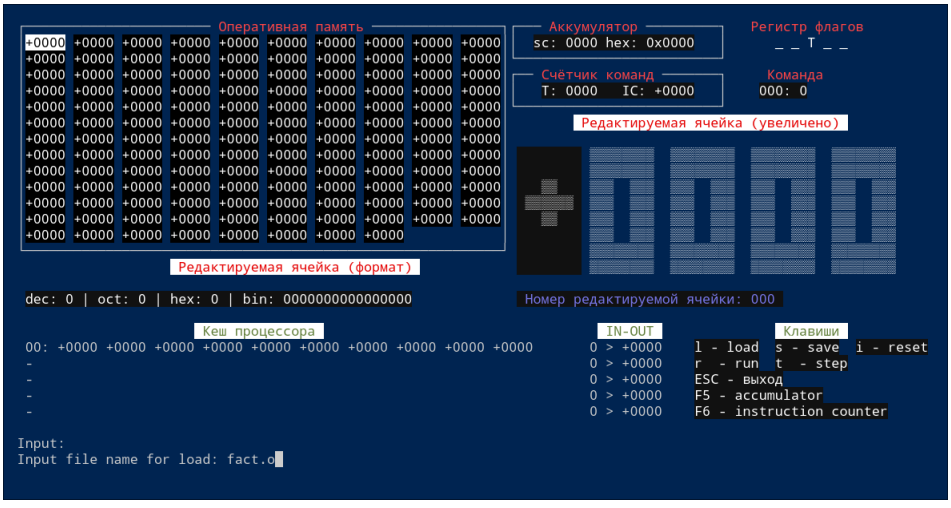
```
❯ ./sb factorial.sb factorial.sa
Basic file 'factorial.sb' is successfully opened.
10 INPUT A
20 LET B = 1
30 LET C = 1
error hehehe
Invalid syntax
40 LET B = A * B
50 LET A = A - C
60 IF A > 0 GOTO 40
70 PRINT B
80 END
```

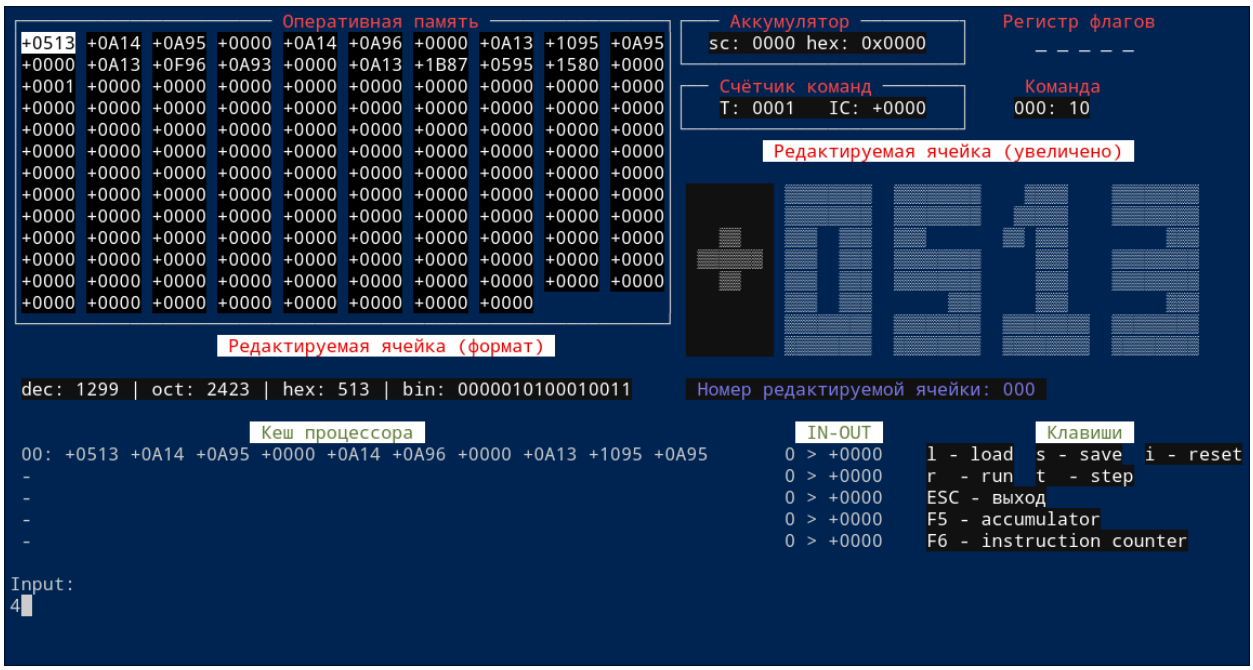
Ошибка трансляции

Результат трансляции с SimpleAssembler в образ оперативной памяти для SimpleComputer:

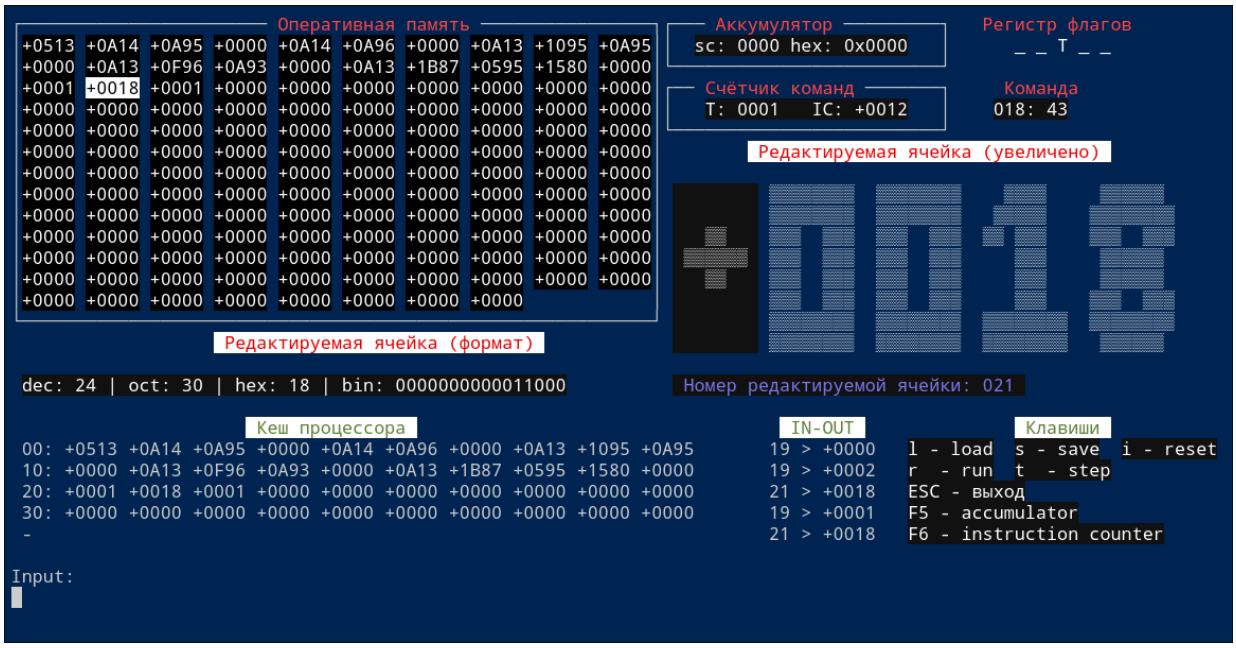
```
❯ ./sa factorial.sa fact.o
Source file is open
0 10 19
1 20 20
2 21 21
4 20 20
5 21 22
7 20 19
8 33 21
9 21 21
11 20 19
12 31 22
13 21 19
15 20 19
16 55 07
17 11 21
18 43 00
19 1 +0000
20 1 +0001
21 1 +0000
22 1 +0000
Output file 'fact.o' created successfully.
```

Результат загрузки файла в SimpleComputer:





Вводим значение A = 4



Выводы

В рамках курсовой работы была реализована интеграция кэш-процессора, а также разработаны трансляторы с языка Simple Basic и Simple Assembler.

Список использованной литературы

1. ЭВМ и ПЕРИФЕРИЙНЫЕ УСТРОЙСТВА [Учебное пособие]. С.Н. Мамойленко О.В. Молдованова

2. c-cpp.ru [сайт]. URL: https://www.c-cpp.ru/

3. Язык программирования Си - R-5.

https://www.r-5.org/files/books/computers/languages/c/kr/Brian_Kernighan_Dennis_Ritchie-The_C_Programming_Language-RU.pdf [URL]

Листинг

```
#include <mySimpleComputer.h>
#include <sc.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_LINE_LENGTH 100
#define MAX_COMMENT_LENGTH 10
#define MEMORY_SIZE 128
typedef struct
{
const char *name;
int code;
} Command;
bool
isValidRange (int addr, int value)
{
return (addr >= 0 && addr < MEMORY_SIZE && value >= 0 && value <= 65535);
}
int
memorySet (int addr, int value)
{
if (!isValidRange (addr, value))
{
sc_regSet (OUT_OF_MEMORY_BOUNDS, 1);
printf ("OUT_OF_MEMORY_BOUNDS, address: %x\n", addr);
return -1;
}
memory[addr] = value;
return 0;
}
int
commandValidate (char *command)
{
Command commands[]
= { { "NOP", 0x00 }, { "CPUINFO", 0x01 }, { "READ", 0x0A },
{ "WRITE", 0x0B }, { "LOAD", 0x14 }, { "STORE", 0x15 },
{ "ADD", 0x1E }, { "SUB", 0x1F }, { "DIVIDE", 0x20 },
{ "MUL", 0x21 }, { "JUMP", 0x28 }, { "JNEG", 0x29 },
{ "JZ", 0x2A }, { "HALT", 0x2B }, { "JNS", 0x37 },
{ "JP", 0x3A }, { "SUBC", 0x42 } };
int numCommands = sizeof (commands) / sizeof (commands[0]);
for (int i = 0; i < numCommands; ++i)
{
if (strcmp (command, commands[i].name) == 0)
{
18
return commands[i].code;
```

```
}
}
if (command[0] == '=')
{
return 1;
}
return -1;
}
int
main (int argc, char **argv)
{
FILE *source_file;
if (argc != 3)
{
printf ("Usage: %s source_file.sa output_file.o\n", argv[0]);
return -1;
}
if ((source_file = fopen (argv[1], "rb")) == NULL)
{
printf ("Can't open '%s' file.\n", argv[1]);
return -1;
}
printf ("Source file is open\n");
sc_memoryInit ();
char line[MAX_LINE_LENGTH];
char comment[MAX_COMMENT_LENGTH];
char operand_str[MAX_COMMENT_LENGTH];
char comment_line[MAX_LINE_LENGTH] = "\0";
int address, command, operand, encoded_value;
do
{
fgets (line, sizeof (line), source_file);
if (feof (source_file))
{
break;
}
if (sscanf (line, "%d %s %s ;%s", &address, comment, operand_str,
comment_line)
< 3)
{
printf ("Invalid syntax in line: %s\n", line);
fclose (source_file);
19
return -1;
}
command = commandValidate (comment);
printf ("%d %d %s\n", address, command, operand_str);
if (command != -1)
{
switch (command)
```

```
{
case 0:
case 1:
case 0xA:
case 0xB:
case 0x14:
case 0x15:
case 0x1E:
case 0x1F:
case 0x20:
case 0x21:
case 0x28:
case 0x29:
case 0x2A:
case 0x2B:
case 0x37:
case 0x3A:
case 0x42:
if (sscanf (operand_str, "%d", &operand) != 1
|| sc_commandEncode (command, operand, &encoded_value) == -1
|| memorySet (address, encoded_value) == -1)
{
printf ("Invalid syntax in line: %s\n", line);
fclose (source_file);
return -1;
}
break;
default:
printf ("Invalid command in line: %s\n", line);
fclose (source_file);
return -1;
}
}
else
{
printf ("Invalid syntax in line: %s\n", line);
fclose (source_file);
return -1;
}
}
while (!feof (source_file));
20
fclose (source_file);
if (sc_memorySave (argv[2]) == -1)
{
printf ("Can't create '%s' file.\n", argv[2]);
return -1;
}
printf ("Output file '%s' created successfully.\n", argv[2]);
return 0;
```

```
}
#include "mySimpleComputer.h"
#include "sb_variables.h"
#include "sc.h"
#include "simplebasic.h"
int
main (int argc, char **argv)
{
FILE *file_sb, *file_sa;
if (argc != 3)
{
fprintf (stderr, "Usage: %s file.sb file.sa\n", argv[0]);
return -1;
}
if ((file_sb = fopen (argv[1], "rb")) == NULL)
{
fprintf (stderr, "Error: Unable to open '%s' file for reading.\n",
argv[1]);
return -1;
}
if ((file_sa = fopen (argv[2], "wb")) == NULL)
{
fprintf (stderr, "Error: Unable to create '%s' file for writing.\n",
argv[2]);
fclose (file_sb);
return -1;
}
fprintf (stdout, "Basic file '%s' is successfully opened.\n", argv[1]);
compileBasic (file_sb, file_sa, argv, 1);
fclose (file_sb);
fclose (file_sa);
21
return 0;
}
#include "simplebasic.h"
int var_offset, strnum, asm_addr = 0, last_strnum = 0;
int max_strnum, goto_addr, goto_line;
char opr[6], str[100], ch;
char line[100];
int linenum = 1;
void
initializeVariables ()
{
for (size_t i = 0; i < sizeof (asmString) / sizeof (asmString[0]); i++)
{
sprintf (asmString[i], "%s", "");
}
for (size_t i = 0; i < sizeof (asmStringOut) / sizeof (asmStringOut[0]); i++)
{
sprintf (asmStringOut[i], "%s", "");
```

```
}
for (size_t i = 0; i < sizeof (variables); i++)
{
strcpy (&variables[i], "\0");
}
}
int
dispatchInstruction (int strnum, char *opr)
{
if (strcmp (opr, "REM") == 0)
{
linenum++;
return 0;
}
else if (strcmp (opr, "END") == 0)
{
handleEnd ();
}
else if (strcmp (opr, "INPUT") == 0)
{
handleInput ();
}
else if (strcmp (opr, "LET") == 0)
{
handleLet ();
}
else if (strcmp (opr, "IF") == 0)
22
{
handleIf ();
}
else if (strcmp (opr, "GOTO") == 0)
{
handleGoto ();
}
else if (strcmp (opr, "PRINT") == 0)
{
handlePrint ();
}
else
{
printf ("Invalid syntax\n");
return -1;
}
}
void
postprocess (int i)
{
sscanf (asmString[i], "%d %d %d %s", &linenum, &strnum, &asm_addr, opr);
if ((strcmp (opr, "READ") == 0) || (strcmp (opr, "WRITE") == 0)
```

```
|| (strcmp (opr, "LOAD") == 0) || (strcmp (opr, "STORE") == 0)
|| (strcmp (opr, "ADD") == 0) || (strcmp (opr, "SUB") == 0)
|| (strcmp (opr, "DIVIDE") == 0) || (strcmp (opr, "MUL") == 0))
{
postprocessCommands (i);
}
else if ((strcmp (opr, "JUMP") == 0) || (strcmp (opr, "JNEG") == 0)
|| (strcmp (opr, "JNS") == 0) || (strcmp (opr, "JZ") == 0))
{
postprocessJumps (i);
}
else if (strcmp (opr, "HALT") == 0)
{
postprocessHalt (i);
}
}
int
processLineFromFile (FILE *file_sb)
{
fgets (line, sizeof (line), file_sb);
if (feof (file_sb))
{
return 0;
}
printf ("%s", line);
23
if (strcmp (line, "\n") != 0)
{
if (sscanf (line, "%d %s", &strnum, opr) >= 2)
{
if (strnum <= last_strnum)
{
printf ("Invalid syntax\n");
return -1;
}
dispatchInstruction (strnum, opr);
asm_addr++;
last_strnum = strnum;
}
else
{
printf ("Invalid syntax\n");
return -1;
}
}
}
int
compileBasic (FILE *file_sb, FILE *file_sa, char **argv, int flag)
{
initializeVariables ();
```

```
do
{
processLineFromFile (file_sb);
linenum++;
}
while (!feof (file_sb));
fclose (file_sb);
max_strnum = strnum;
var_offset = asm_addr;
for (int i = 0; i < sizeof (asmString) / sizeof (asmString[0]); i++)
{
if (strcmp (asmString[i], "") != 0)
{
postprocess (i);
}
else
{
if (i >= var_offset)
{
postprocessVariables (i);
}
}
24
}
// Запись результата в файл
for (int i = 0; i < sizeof (asmString) / sizeof (asmString[0]); i++)
{
fwrite (asmStringOut[i], 1, strlen (asmStringOut[i]), file_sa);
} #include "sb_variables.h"
#include "simplebasic.h"
void
sb_clearBackspace (char *str)
{
int read_index = 0;
int write_index = 0;
while (str[read_index] != '\0')
{
if (str[read_index] != ' ')
{
str[write_index] = str[read_index];
write_index++;
}
read_index++;
}
str[write_index] = '\0';
}
fclose (file_sa);
if (flag == 1)
{
exit (0);
```

```
}
if ((file_sa = fopen (argv[2], "r")) == NULL)
{
printf ("Unable to open file '%s'\n", argv[2]);
return -1;
}
if (fseek (file_sa, 0, SEEK_SET) != 0)
{
printf ("Error moving file pointer\n");
return -1;
}
return 0;
}
25
#include "simplebasic.h"
typedef struct
{
const char *name;
int code;
} Command;
int
sb_commandValidate (char *command)
{
Command commands[]
= { { "NOP", 0x00 }, { "CPUINFO", 0x01 }, { "READ", 0x0A },
{ "WRITE", 0x0B }, { "LOAD", 0x14 }, { "STORE", 0x15 },
{ "ADD", 0x1E }, { "SUB", 0x1F }, { "DIVIDE", 0x20 },
{ "MUL", 0x21 }, { "JUMP", 0x28 }, { "JNEG", 0x29 },
{ "JZ", 0x2A }, { "HALT", 0x2B }, { "JNS", 0x37 },
{ "JP", 0x3A }, { "SUBC", 0x42 } };
int numCommands = sizeof (commands) / sizeof (commands[0]);
for (int i = 0; i < numCommands; ++i)
{
if (strcmp (command, commands[i].name) == 0)
{
return commands[i].code;
}
}
if (command[0] == '=')
{
return 1;
}
return -1;
}
#include "sb_variables.h"
#include "simplebasic.h"
int
sb_findOrCreateVariableAddress (char varName, int startOffset)
{
size_t i;
```

```
for (i = 0; i < sizeof (variables); i++)
{
if (variables[i] == varName)
return startOffset + i;
else if (variables[i] == '\0')
{
variables[i] = varName;
return startOffset + i;
26
}
}
return -1; // Возврат -1, если нет свободного места для новой переменной
}
#include "sb_variables.h"
#include "simplebasic.h"
#include <string.h>
int
sb_getGotoAddr (int gotoLine)
{
int ln, num, addr;
char *token;
for (size_t k = 0; k < sizeof (asmString) / sizeof (asmString[0]); k++)
{
char line[strlen (asmString[k]) + 1];
strcpy (line, asmString[k]);
token = strtok (line, " ");// разделение строки по пробелами
while (token != NULL)
{
sscanf (token, "%d", &ln);
if (ln == gotoLine)
{
// Берем следующий токен (это должен быть адрес)
token = strtok (NULL, " ");
sscanf (token, "%d", &addr);
return addr;
}
token = strtok (NULL, " ");
}
}
return -1;
}
#include "sb_variables.h"
#include "simplebasic.h"
char
sb_getVariableName (int variableAddress, int nameOffset)
{
if (variableAddress - nameOffset >= 0
&& strcmp ((variables + variableAddress - nameOffset), "\0") != 0)
27
{
```

```
return variables[variableAddress - nameOffset];
}
return '\0';
}
#include "simplebasic.h"
void
handleEnd ()
{
int result = sscanf (line, "%d %s %s", &strnum, opr, str);
if (result != 2)
{
printf ("Invalid syntax\n");
return;
}
sprintf (asmString[asm_addr], "%d %d %d %s", linenum, strnum, asm_addr,
"HALT\n");
}
#include "simplebasic.h"
void
handleGoto ()
{
int result = sscanf (line, "%d %s %d %s", &strnum, opr, &goto_line, str);
if (result != 3)
{
printf ("Invalid syntax\n");
return;
}
sprintf (asmString[asm_addr], "%d %d %d %s %d%s", linenum, strnum, asm_addr,
"JUMP", goto_line, "\n");
}
#include "simplebasic.h"
void
handleIf ()
{
char *s = strstr (line, "IF") + 2;
sb_clearBackspace (s);
char ch1, ch2, ch3;
if (sscanf (s, "%c%c%cGOTO%d%s", &ch1, &ch2, &ch3, &goto_line, str) != 4)
{
printf ("Invalid syntax\n");
return;
}
sprintf (asmString[asm_addr], "%d %d %d %s %c%s", linenum, strnum, asm_addr,
"LOAD", ch1, "\n");
28
asm_addr++;
char *jumpInstr = "";
if ((ch2 == '<' && ch3 == '0') || (ch2 == '>' && ch3 == '0')
|| (ch2 == '=' && ch3 == '0'))
{
```

```
if (ch2 == '<')
jumpInstr = "JNEG";
else if (ch2 == '>')
jumpInstr = "JNS";
else if (ch2 == '=')
jumpInstr = "JZ";
sprintf (asmString[asm_addr], "%d %d %d %s %d%s", -1, -1, asm_addr,
jumpInstr, goto_line, "\n");
}
else
{
printf ("Invalid syntax\n");
return;
}
}
#include "simplebasic.h"
void
handleInput ()
{
int result = sscanf (line, "%d %s %c %s", &strnum, opr, &ch, str);
if (result != 3)
{
printf ("Invalid syntax\n");
return;
}
sprintf (asmString[asm_addr], "%d %d %d %s %c%s", linenum, strnum, asm_addr,
"READ", ch, "\n");
}
#include "simplebasic.h"
void generateAssembly(int linenum, int strnum, int asm_addr, char *operation,
char var, char *newline) {
sprintf(asmString[asm_addr], "%d %d %d %s %c%s", linenum, strnum, asm_addr,
operation, var, newline);
}
void handleLet() {
char *s = strstr(line, "LET") + 4;
29
sb_clearBackspace(s);
char targetVar, sourceVar1, sourceVar2, operator;
int value;
// Проверка на выражение в скобках
if (sscanf(s, "%c=(%c%c%c)", &targetVar, &sourceVar1, &operator,
&sourceVar2) == 4) {
// Вычисление выражения в скобках
generateAssembly(linenum, strnum, asm_addr, "LOAD", sourceVar1, "");
asm_addr++;
switch (operator) {
case '+':
generateAssembly(-1, -1, asm_addr, "ADD", sourceVar2, "\n");
break;
```

```
case '-':
generateAssembly(-1, -1, asm_addr, "SUB", sourceVar2, "\n");
break;
case '*':
generateAssembly(-1, -1, asm_addr, "MUL", sourceVar2, "\n");
break;
case '/':
generateAssembly(-1, -1, asm_addr, "DIVIDE", sourceVar2, "\n");
break;
default:
printf("Invalid syntax\n");
return;
}
asm_addr++;
generateAssembly(-1, -1, asm_addr, "STORE", 'T', "\n"); // Временная
переменная для результата в скобках
asm_addr++;
// Загрузка результата выражения в скобках и выполнение операции с
использованием его
generateAssembly(linenum, strnum, asm_addr, "LOAD", 'T', "");
asm_addr++;
generateAssembly(-1, -1, asm_addr, "MUL", sourceVar1, "\n");
asm_addr++;
generateAssembly(-1, -1, asm_addr, "STORE", targetVar, "\n");
asm_addr++;
last_strnum = strnum;
linenum++;
return;
}
// Обработка обычного выражения без скобок
if (sscanf(s, "%c=%d", &targetVar, &value) == 2) {
generateAssembly(linenum, strnum, asm_addr, "LOAD", value, "");
asm_addr++;
generateAssembly(-1, -1, asm_addr, "STORE", targetVar, "\n");
30
asm_addr++;
last_strnum = strnum;
linenum++;
return;
}
if (sscanf(s, "%c=%c%c%c", &targetVar, &sourceVar1, &operator, &sourceVar2)
!= 4) {
printf("Invalid syntax\n");
return;
}
// Обработка обычного выражения без скобок
generateAssembly(linenum, strnum, asm_addr, "LOAD", sourceVar1, "");
asm_addr++;
switch (operator) {
case '+':
```

```
generateAssembly(-1, -1, asm_addr, "ADD", sourceVar2, "\n");
break;
case '-':
generateAssembly(-1, -1, asm_addr, "SUB", sourceVar2, "\n");
break;
case '*':
generateAssembly(-1, -1, asm_addr, "MUL", sourceVar2, "\n");
break;
case '/':
generateAssembly(-1, -1, asm_addr, "DIVIDE", sourceVar2, "\n");
break;
default:
printf("Invalid syntax\n");
return;
}
asm_addr++;
generateAssembly(-1, -1, asm_addr, "STORE", targetVar, "\n");
asm_addr++;
last_strnum = strnum;
linenum++;
}
#include "simplebasic.h"
void
handlePrint ()
{
int result = sscanf (line, "%d %s %c %s", &strnum, opr, &ch, str);
if (result != 3)
{
31
printf ("Invalid syntax\n");
return;
}
sprintf (asmString[asm_addr], "%d %d %d %s %c%s", linenum, strnum, asm_addr,
"WRITE", ch, "\n");
}
#include "../include/mySimpleComputer.h"
#include "../mySimpleComputer/sc.h"
#include "simplebasic.h"
int
sb_memorySet (int address, int value)
{
if (address < 0 || address > 128)
{
sc_regSet (OUT_OF_MEMORY_BOUNDS, 1);
printf ("OUT_OF_MEMORY_BOUNDS, address: %x\n", address);
return -1;
}
if (value < 0 || value > 65535)
{
sc_regSet (OUT_OF_MEMORY_BOUNDS, 1);
```

```
printf ("OUT_OF_MEMORY_BOUNDS, value: %x\n", value);
return -1;
}
memory[address] = value;
return 0;
}
#include "simplebasic.h"
void
postprocessJumps (int i)
{
int parsed_values = sscanf (asmString[i], "%d %d %d %s %d", &linenum,
&strnum, &asm_addr, opr, &goto_line);
if (parsed_values != 5)
{
printf ("Invalid syntax\n");
return;
}
int goto_addr = sb_getGotoAddr (goto_line);
if (goto_addr == -1)
{
printf ("Invalid syntax\n");
return;
}
sprintf (asmStringOut[i], "%02d %s %02d\n", asm_addr, opr, goto_addr);
32
}
#include "simplebasic.h"
void
postprocessCommands (int i)
{
sscanf (asmString[i], "%d %d %d %s %c", &linenum, &strnum, &asm_addr, opr,
&ch);
int var_addr = sb_findOrCreateVariableAddress (ch, var_offset);
sprintf (asmStringOut[i], "%02d %s %02d%s", asm_addr, opr, var_addr, "\n");
int val;
if (sscanf (asmString[i], "%*d %*d %*d %s %d", opr, &val) == 2
&& strcmp (opr, "LOAD") == 0)
{
numArray[var_addr] = val;
}
}
#include "simplebasic.h"
void
postprocessHalt (int i)
{
sscanf (asmString[i], "%d %d %d %s", &linenum, &strnum, &asm_addr, opr);
sprintf (asmStringOut[i], "%02d %s%s", asm_addr, opr, " 00\n");
}
#include "simplebasic.h"
void
```

```
postprocessVariables (int i)
{
char varName = sb_getVariableName (i, var_offset);
if (varName != '\0')
{
sprintf (asmStringOut[i], "%02d %c %s", i, '=', "+0000\n");
}
if (numArray[i] != 0)
{
sprintf (asmStringOut[i], "%02d %c +%04X%s", i, '=', numArray[i], "\n");
}
}
#pragma once
#include "../include/mySimpleComputer.h"
#include "../mySimpleComputer/sc.h"
#include "sb_variables.h"
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
33
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
int compileBasic (FILE *file_sb, FILE *file_sa, char **argv, int flag);
char sb_getVariableName (int variableAddress, int nameOffset);
int sb_getGotoAddr (int gotoLine);
void sb_clearBackspace (char *str);
int sb_findOrCreateVariableAddress (char varName, int startOffset);
int sb_memorySet (int address, int value);
int sb_commandValidate (char *command);
void handleEnd ();
void handleInput ();
void handleLet ();
void handleIf ();
void handleGoto ();
void handlePrint ();
void postprocessJumps (int i);
void postprocessCommands (int i);
void postprocessHalt (int i);
void postprocessVariables (int i);
extern int var_offset, strnum, asm_addr, last_strnum;
extern int max_strnum, goto_addr, goto_line;
extern char opr[6], str[100], ch;
extern char line[100];
extern int linenum;
#pragma once
#include "simplebasic.h"
extern char variables[256];
```

```
extern char asmString[256][256];
extern char asmStringOut[256][256];
extern int numArray[256];
#include <mySimpleComputer.h>
#include <sc.h>
void sc_initCache() {
cache = calloc(CACHE_SIZE, sizeof(CacheLine));
if (cache == NULL) {
return;
}
for (size_t i = 0; i < CACHE_SIZE; ++i) {
cache[i].address = -1;
cache[i].lastAccessTime = 0;
34
}
}
#include <mySimpleComputer.h>
#include <sc.h>
#include <limits.h>
int sc_findLeastRecentlyUsedCacheEntry() {
size_t leastRecentlyUsedIndex = INT_MAX; // Инициализация максимальным
значением
time_t oldestAccessTime = LONG_MAX; // Инициализация максимальным значением
времени доступа
for (size_t i = 0; i < CACHE_SIZE; ++i) {
if (cache[i].lastAccessTime < oldestAccessTime) {
oldestAccessTime = cache[i].lastAccessTime;
leastRecentlyUsedIndex = i;
}
}
return leastRecentlyUsedIndex;
}
#include <mySimpleComputer.h>
#include <sc.h>
#include <stdbool.h>
#include <time.h>
static void updateCacheLine(int cacheIndex, int address) {
for (int j = 0; j < CACHE_LINE_SIZE; ++j) {
cache[cacheIndex].data[j] = memory[address + j];
}
cache[cacheIndex].address = address;
cache[cacheIndex].lastAccessTime = time(NULL);
}
void sc_updateCacheAfterSave(int memaddress, int cacheLine, int *value) {
static bool initialized = false;
static size_t currentIndex = 0;
if (!initialized) {
initialized = true;
currentIndex = 0;
}
```

```
*value = memory[memaddress];
int address = (cacheLine / 10) * 10;
// Проверяем, есть ли уже запись в кеше для данного адреса
size_t i;
35
bool cacheHit = false;
for (i = 0; i < CACHE_SIZE; ++i) {
if (cache[i].address == address) {
cacheHit = true;
currentIndex = i; // Запоминаем индекс строки кеша, в которую
попали
break;
}
}
if (!cacheHit) {
if (i < CACHE_SIZE) {
// Если кеш не заполнен, добавляем новую запись
currentIndex = i;
} else {
// Если кеш полон, заменяем наименее используемую запись
currentIndex = sc_findLeastRecentlyUsedCacheEntry();
}
updateCacheLine(currentIndex, address);
} else {
// Обновляем существующую запись в кеше
updateCacheLine(currentIndex, address);
}
}
```