

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 .....	2
Блок-схемы использованных алгоритмов .....	5
Программная реализация .....	8
Результаты проведённого исследования .....	13
Выводы .....	16
Список использованной литературы .....	16
Листинг .....	17

## 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.

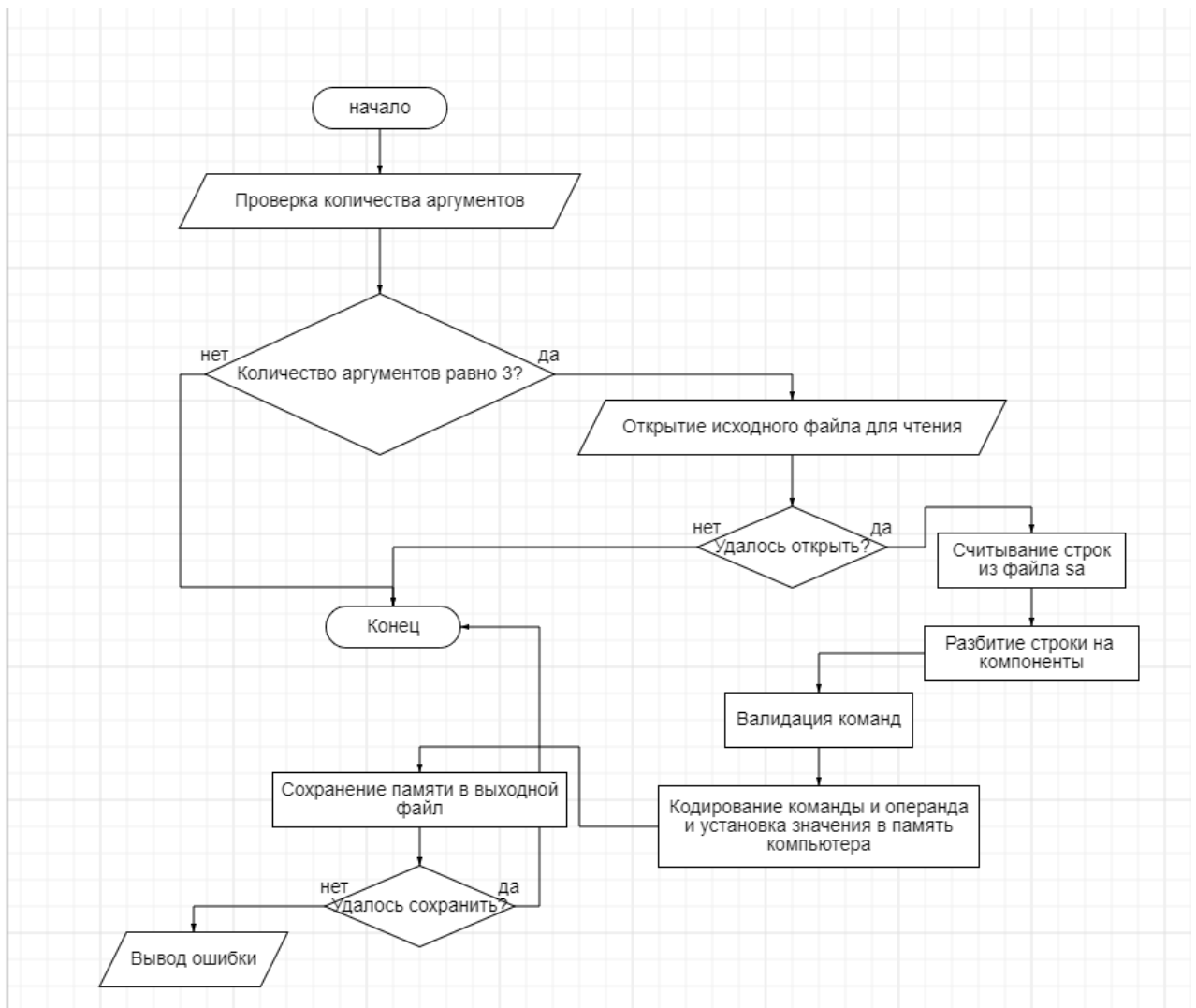
Номера строк должны следовать в возрастающем порядке. Все команды за исключением команды конца программы могут встречаться в программе многократно. Simple Basic должен оперировать с целыми выражениями, включающими операции +, -, \*, и /. Приоритет операций аналогичен C. Для того чтобы изменить порядок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

Транслятор должен распознавать 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

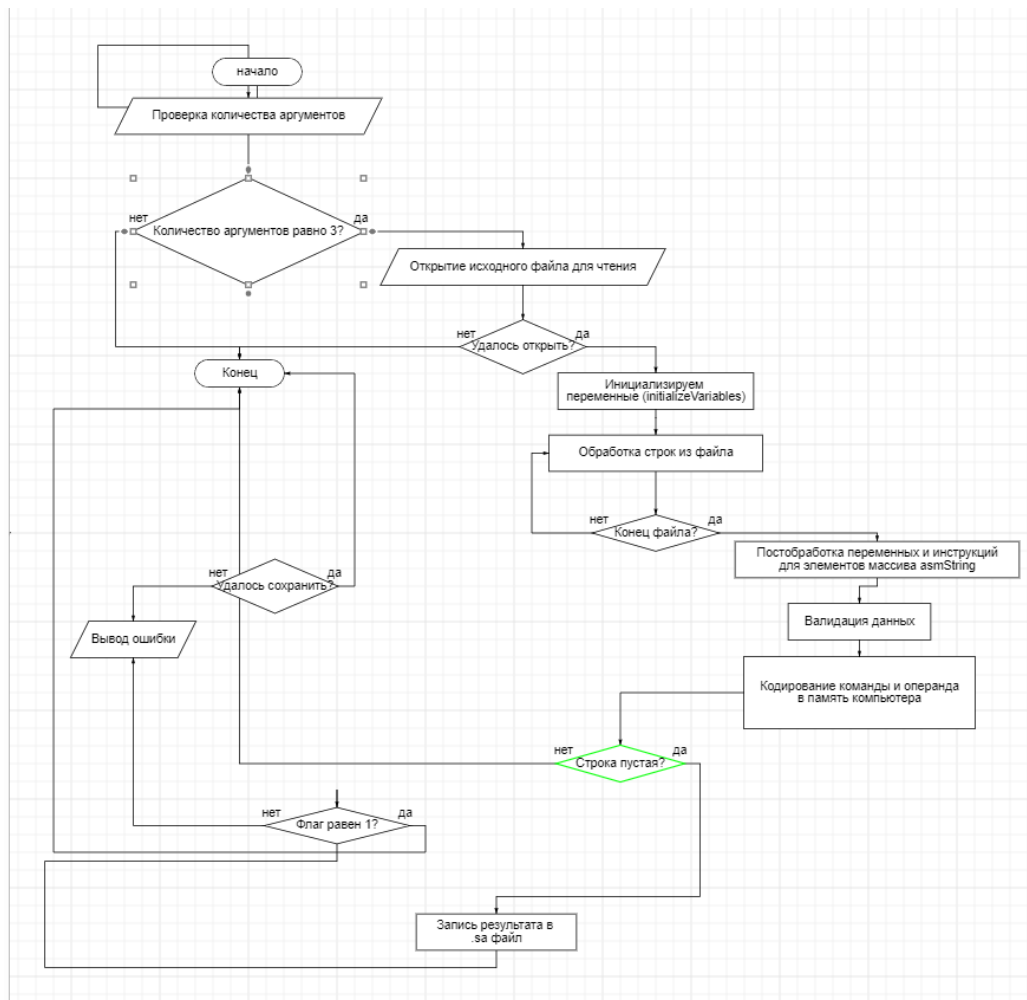
только с целыми значениями переменных, в нем отсутствует объявление переменных, а упоминание переменной автоматически вызывает её объявление и присваивает ей нулевое значение. Синтаксис языка не позволяет выполнять операций со строками.

## Блок-схемы использованных алгоритмов

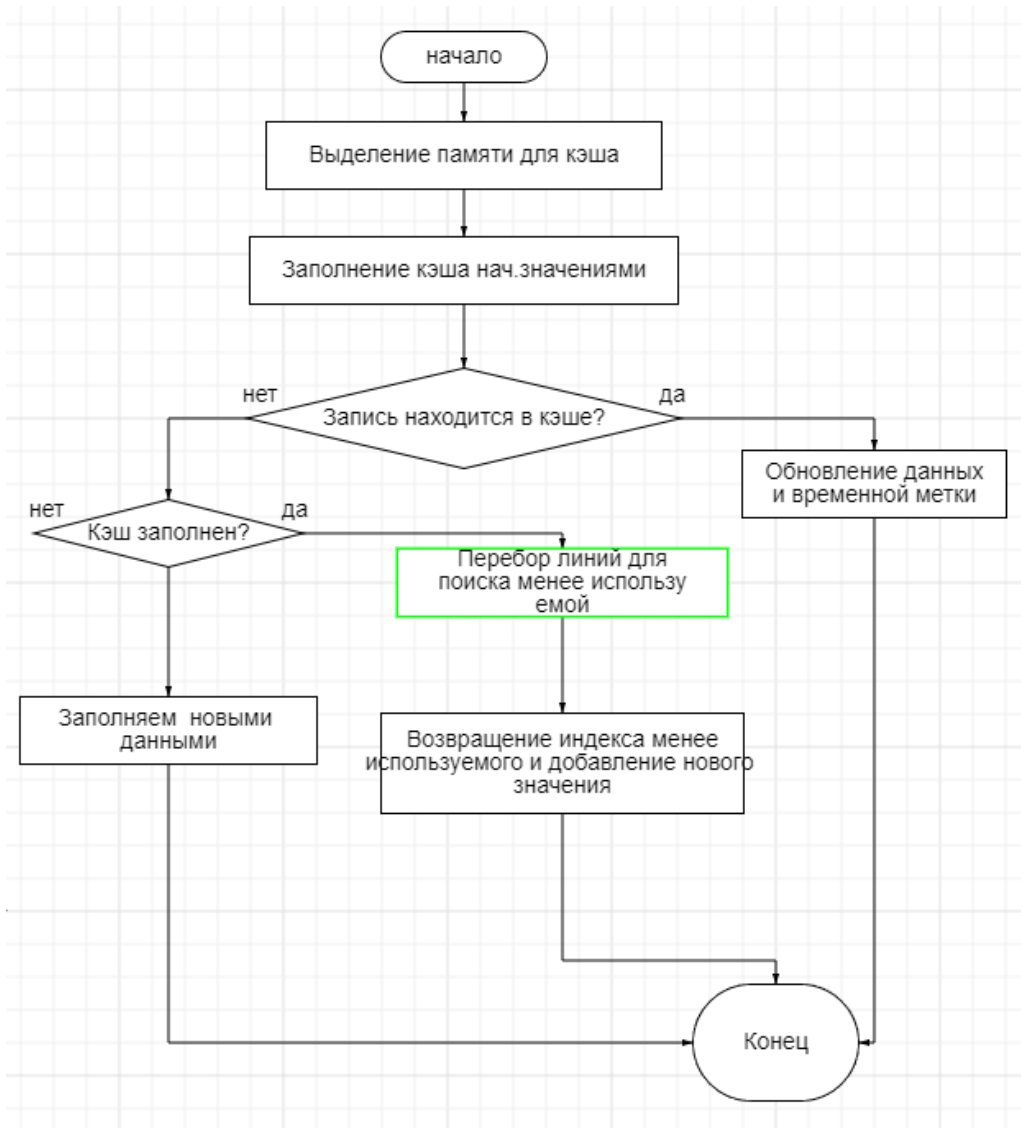
### SimpleAssembler:



## SimpleBasic:



## Кеш процессора:





## Программная реализация

Кеш процессора:

`sc_initCache ():`

Функция инициализирует кэш, выделяя память под массив структуры 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 орг 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:

The screenshot displays the SimpleComputer interface with the following components:

- Оперативная память (Main Memory):** A grid of 256 memory cells, each containing the value +0000.
- Аккумулятор (Accumulator):** sc: 0000 hex: 0x0000
- Регистр флагов (Flag Register):** -- T --
- Счётчик команд (Instruction Counter):** T: 0000 IC: +0000
- Команда (Instruction):** 000: 0
- Редактируемая ячейка (увеличено) (Editable Cell - enlarged):** A large grid for editing memory cells.
- Редактируемая ячейка (формат) (Editable Cell - format):** dec: 0 | oct: 0 | hex: 0 | bin: 0000000000000000
- Кеш процессора (Processor Cache):** 00: +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
- IN-OUT:** 0 > +0000, 0 > +0000, 0 > +0000, 0 > +0000, 0 > +0000
- Клавиши (Keys):** l - load s - save i - reset, r - run t - step, ESC - выход, F5 - accumulator, F6 - instruction counter
- Input:** Input file name for load: fact.o





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