Федеральное государственное бюджетное образовательное учреждение высшего образования

«Сибирский государственный университет телекоммуникаций и информатики»

(ФГОБУ ВО «СибГУТИ)

Кафедра ВС

Отчет по расчетно-графической работе по дисциплине «Архитектура ЭВМ»

Выполнили:

Студенты 2 курса группы ИА-331

Иванов Иван Алексеевич

Гмыря Ярослав Александрович

Проверил:

Майданов Юрий Сергеевич

Новосибирск 2025

Оглавление

Цель курсовой работы

В рамках курсовой работы необходимо:

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

Введение

В рамках выполнения курсовой работы требуется разработать программное обеспечение для трансляции программ, написанных на языках **Simple Basic** и **Simple Assembler**, в исполняемый формат, пригодный для загрузки в модель **Simple Computer**. Дополнительно необходимо доработать архитектуру **Simple Computer**, реализовав **кэш-память первого уровня (L1)** для команд и данных, а также модифицировать контроллер оперативной памяти и обработчик прерываний для учета времени простоя процессора при прямом доступе к памяти.

Транслятор с языка Simple Assembler

Транслятор преобразует текстовый код на Simple Assembler в бинарный формат, пригодный для исполнения на Simple Computer. Данный транслятор реализован на языке **Python**

Блок схема транслятора с языка Simple Assembler

Принцип работы транслятора с языка Simple Assembler

Чтение исходного кода:

- Файл .sa читается построчно.

Парсинг:

- Каждая строка парсится на:
  - Адрес (например, 00, 01);
  - Команду (READ, WRITE, LOAD, SUB и др.);
  - Операнд (число или метка);
  - Комментарий (игнорируется);

Валидация:

- Проверка корректности команд, операндов и адресов.

Генерация машинного кода:

- Каждая команда конвертируется в числовой код (например, READ → 0x10).
- К команде справа дописывается операнд.

Формирование бинарного образа:

- Создается файл .o, содержащий образ памяти (массив), записанный в файл в бинарном режиме

Программная реализация

Файл translater_from_sa_to_o.py

```
import sys
from array import array

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,
    "NOT": 0x33,
    "AND": 0x34,
    "OR": 0x35,
    "XOR": 0x36,
    "JNS": 0x37,
    "JC": 0x38,
    "JNC": 0x39,
    "JP": 0x3A,
    "JNP": 0x3B,
    "CHL": 0x3C,
    "SHR": 0x3D,
    "RCL": 0x3E,
    "RCR": 0x3F,
    "NEG": 0x40,
    "ADDC": 0x41,
    "SUBC": 0x42,
    "LOGLC": 0x43,
    "LOGRC": 0x44,
    "RCCL": 0x45,
    "RCCR": 0x46,
    "MOVA": 0x47,
    "MOVR": 0x48,
    "MOVCA": 0x49,
    "MOVCR": 0x4A,
    "ADDC2": 0x4B,
    "SUBC2": 0x4C
}

def delete_extra_space(line):
    return ' '.join(line.strip().split())

def main():
    if len(sys.argv) != 3:
        print("Использование: sat файл.sa файл.o")
        sys.exit(1)

    input_file = sys.argv[1]
```

```
    output_file = sys.argv[2]

    result = array('i', [0] * 128)


    with open(input_file, "r") as file:
        for line_num, line in enumerate(file, 1):
            clean_line = delete_extra_space(line)

            if len(clean_line.split(" ")) != 3:
                print(f"Ошибка в строке {line_num}: {clean_line}")
                sys.exit(1)

            address, command, operand = clean_line.split(" ")

            address = int(address)
            operand = int(operand)

            if not (0 <= address < 128):
                print(f"Недопустимый адрес в строке {line_num}:
{clean_line}")
                sys.exit(1)

            if command not in commands:
                print(f"Неизвестная команда в строке {line_num}:
{clean_line}")
                sys.exit(1)

            if not (0 <= operand < 128):
                print(f"Недопустимый операнд в строке {line_num}:
{clean_line}")
                sys.exit(1)

            command_code = commands[command]

            full_command = (command_code << 7) | operand

            result[address] = full_command

    with open(output_file, "wb") as bin_file:
        result.tofile(bin_file)

    print(f"Трансляция завершена. Результат сохранён в '{output_file}'.")

if __name__ == '__main__':
    main()
```

Запуск программы:

python3 translater_from_sa_to_o.py *.sa *.o

Объяснение работы программы:

1. Создается словарь, где ключ – команда (строка), а значение – ее машинный код

2. Начало выполнение функции main

3. Проверка количества аргументов, переданных в командной строке (их должно быть 3: один аргумент под название программы и 2 под файлы)

4. Создается массив (образ памяти), куда будут записываться команды

5. Извлекаем из аргументов командой строки названия файлов

6. Открываем файл с программой для чтения. Проходимся циклом по файлу, извлекая на каждой итерации строку

7. Очищаем строку от лишних пробелов с помощью функции delete_extra_space

8. Далее делаем split по пробелу (разбиваем строку на элемент, разделенные пробелом и помещаем элементы в список), если длина списка не равна трем (т.е помимо адреса, команды и операнда есть что-то лишнее в строке), то заканчиваем работу программы

9. Далее в переменные записываем адрес, команду, операнд

10. Приводим адрес и операнд к int

11. Далее идут проверки. Если адрес и операнд не в отрезке [0, 128] или команды нет в словаре, то программа заканчивает работу

12. Если все проверки пройдены, то дописываем к команде справа операнд и записываем в массив result по индексу равному адресу

13. Открываем файл на запись в бинарном режиме. Записываем в него массив

Пример программы assembler (вычисление факториала)

```
00 READ 20 ; записываем n
01 READ 21 ; записываем 1 (константа для декремента)
02 READ 22 ; записываем 1 (здесь будет значение факториала)
03 LOAD 20 ; загрузить в аккумулятор текущий n
04 JZ 11 ; конец программы, если n == 0 (переход к ячейке, где записана команда
HALT)
05 LOAD 22 ; загрузить в аккумулятор текущий факториал
06 MUL 20 ; умножаем текущий факториал на n
07 STORE 22 ; выгружаем новое значение факториала из аккумулятора
08 LOAD 20 ; записываем в аккумулятор n
09 SUB 21 ; декрементируем n
10 STORE 20 ; записываем из аккумулятора в память новое n
11 JUMP 03 ; безусловный переход на строку 3 (цикл)
12 HALT ; конец программы
```

Результат работы программы

После завершения программы в директории, где находится транслятор, появится файл с бинарным образом памяти, в которой хранится программа

Транслятор с языка Simple Basic

Транслятор с высокоуровневого языка Simple Basic преобразует файл, содержащий программу на языке Simple Basic в файл с кодом Simple Assembler, затем Simple Assembler-файл транслируется в бинарный формат. Данный транслятор реализован на языке **Python**

Блок схема транслятора с языка Simple Basic

Принцип работы транслятора с языка Simple Basic

Этапы трансляции:

1) Чтение .sb

2) Парсинг:

- **Разбиение на составные части**

3) Валидация:

- **Проверка корректности составных частей**

4) Генерация кода Simple Assembler:

- **Перевод команд bs в команды sa**

5) Формирование .sa файла

- **Запись команд в .sa в текстовом формате**

Программная реализация

Файл translater_to_basic.py

```
import random
import sys

commands_basic = {
    "REM" : "NOP",
    "INPUT" : "READ",
    "LET" : "",
    "IF" : ["JNS", "JNEG", "JZ"],
    "GOTO" : "JUMP",
    "PRINT" : "WRITE",
    "END" : "HALT",
}

variables = {}

def delete_extra_space(line):
    return ' '.join(line.strip().split())

def main():
    random.seed(10)
    if len(sys.argv) != 3:
        print("Использование: sat файл.sa файл.o")
        sys.exit(1)

    input_file = sys.argv[1]
    output_file = sys.argv[2]

    current_line_number = 0
```

```
    num_line = 0

    with open(input_file, "r") as input, open(output_file, "w") as output:
        for line in input:
            line = delete_extra_space(line)
            split_line = line.split(" ")
            #num_line = int(split_line[0])

            if num_line % 10 != 0 or num_line < current_line_number or
num_line >= 980:
                print(f"Неверный номер в строке: {line}")
                exit(1)

            current_line_number = num_line

            command = split_line[1]

            if command == "INPUT":
                if len(split_line) != 3:
                    print(f"Неверное число аргументов в строке : {line}")
                    exit(1)

                operand = split_line[2]

                if operand not in variables:
                    print(len(variables.values()))
                    if len(variables.values()) == 0:
                        new_address = 98

                    else:
                        new_address = max(variables.values())

                    if new_address == 127:
                        print(f"Нет свободной памяти под переменную
{operand}")
                        exit(1)

                    variables[operand] = new_address + 1

                output.write(f"{num_line // 10} {commands_basic[command]}
{variables[operand]}\n")
                num_line += 10

            if command == "END":
                if len(split_line) != 2:
                    print(f"Неверное число аргументов в строке : {line}")
                    exit(1)

                output.write(f"{num_line // 10} {commands_basic[command]}
{0}\n")
                num_line += 10

            if command == "PRINT":
                if len(split_line) != 3:
                    print(f"Неверное число аргументов в строке : {line}")
                    exit(1)

                operand = split_line[2]
```

```
                if operand not in variables:
                    print(f"Попытка вывести несуществующую переменную:
{line}")
                    exit(1)

                output.write(f"{num_line // 10} {commands_basic[command]}
{variables[operand]}\n")
                num_line += 10

            if command == "GOTO":
                if len(split_line) != 3:
                    print(f"Неверное кол-во аргументов в строке: {line}")
                    exit(1)

                next_hop_address = int(split_line[2])

                if next_hop_address > current_line_number or next_hop_address
>= 98:
                    print(f"Попытка перейти на несуществующий адрес: {line}")
                    exit(1)

                output.write(f"{num_line // 10} {commands_basic[command]}
{next_hop_address // 10}\n")
                num_line += 10

            if command == "IF":
                operand = split_line[2]

                if operand not in variables:
                    print(f"Попытка обратиться к несуществующей переменной:
{line}")
                    exit(1)

                sign = split_line[3]

                if sign != "<" and sign != ">" and sign != "==":
                    print(f"Неверная операция в условном операторе: {line}")
                    exit(1)

                if split_line[4] != "0":
                    print(f"Попытка выполнить сравнение не с нулем: {line}")
                    exit(1)

                if split_line[5] != "GOTO":
                    print(f"Не указан адрес перехода: {line}")
                    exit(1)

                next_hop_address = int(split_line[6])

                if sign == "==":
                    output.write(f"{num_line // 10} LOAD
{variables[operand]}\n")
                    num_line += 10
                    output.write(f"{num_line // 10}
{commands_basic[command][2]} {next_hop_address // 10}\n")
                    num_line += 10

                if sign == ">":
```

```
                    output.write(f"{num_line // 10} LOAD
{variables[operand]}\n")
                    num_line += 10
                    output.write(f"{num_line // 10}
{commands_basic[command][0]} {next_hop_address // 10}\n")
                    num_line += 10

                if sign == "<":
                    output.write(f"{num_line // 10} LOAD
{variables[operand]}\n")
                    num_line += 10
                    output.write(f"{num_line // 10}
{commands_basic[command][1]} {next_hop_address // 10}\n")
                    num_line += 10

                #output.write(f"{num_line // 10} {commands_basic[command]}
{variables[operand]}\n")

            if command == "LET":
                if len(split_line) != 7:
                    print(f"Неверный формат выражения: {line}")
                    exit(1)

                store, first_operand, sign, second_operand = split_line[2],
split_line[4], split_line[5], split_line[6]

                if store not in variables:

                    if len(variables.values()) == 0:
                        new_address = 98

                    else:
                        new_address = max(variables.values())

                    if new_address == 127:
                        print(f"Нет свободной памяти под переменную
{operand}")
                        exit(1)

                    variables[store] = new_address + 1

                if split_line[4] not in variables or split_line[6] not in
variables:
                    print(f"Попытка выполнить операцию с необъявленными
переменными: {line}")
                    exit(1)

                output.write(f"{num_line // 10} LOAD
{variables[first_operand]}\n")
                num_line += 10

                if sign == "+":
                    output.write(f"{num_line // 10} ADD
{variables[second_operand]}\n")
                    num_line += 10

                if sign == "-":
                    output.write(f"{num_line // 10} SUB
{variables[second_operand]}\n")
```

```
                    num_line += 10

                if sign == "/":
                    output.write(f"{num_line // 10} DIVIDE
{variables[second_operand]}\n")
                    num_line += 10

                if sign == "*":
                    output.write(f"{num_line // 10} MUL
{variables[second_operand]}\n")
                    num_line += 10

                output.write(f"{num_line // 10} STORE {variables[store]}\n")
                num_line += 10

    print(f"Трансляция завершена. Результат сохранён в '{output_file}'.")

if __name__ == '__main__':
    main()
```

Запуск программы:

python3 translater_from_sb_to_sa.py *.bs *.sa

Объяснение работы программы

Программа работает по принципу транслятора с языка sa в бинарный образ памяти, но только более сложно. Сложность заключается в большом количестве проверок, описывать которые нет смысла, поэтому выделим ключевые отличия:

В языке sb появились полноценные переменные. Каждая переменная соответствует номеру ячейки памяти (допустим переменной A соответствует адрес 95, а переменной B – 110). При анализе кода вместо переменных подставляются соответствующие им адреса памяти. Реализовано с помощью словаря, где переменная – ключ, адрес – значение

Ячейки памяти для переменных присваиваются автоматически. Последние 30 ячеек памяти отведены под переменные. Остальные ячейки используются для хранения команд

У одной команды sb может быть несколько представлений в sa (допустим команда IF, она будет отличаться в sa представлении в зависимости от знака <, >, ==)

Одна команда может соответствовать нескольким командам sa. Допустим команда A + B. В языке sa это будет выглядеть так: LOAD A, ADD B (загружаем A в аккумулятор, добавляем к аккумулятору B)

Формируется не бинарный файл, а текстовый

Пример программы Simple Basic (вычисление факториала)

```
00 INPUT N
10 INPUT A
20 INPUT FACT
30 IF N == 0 GOTO 120
40 LET FACT = N * FACT
50 LET N = N - A
60 GOTO 30
70 END
```

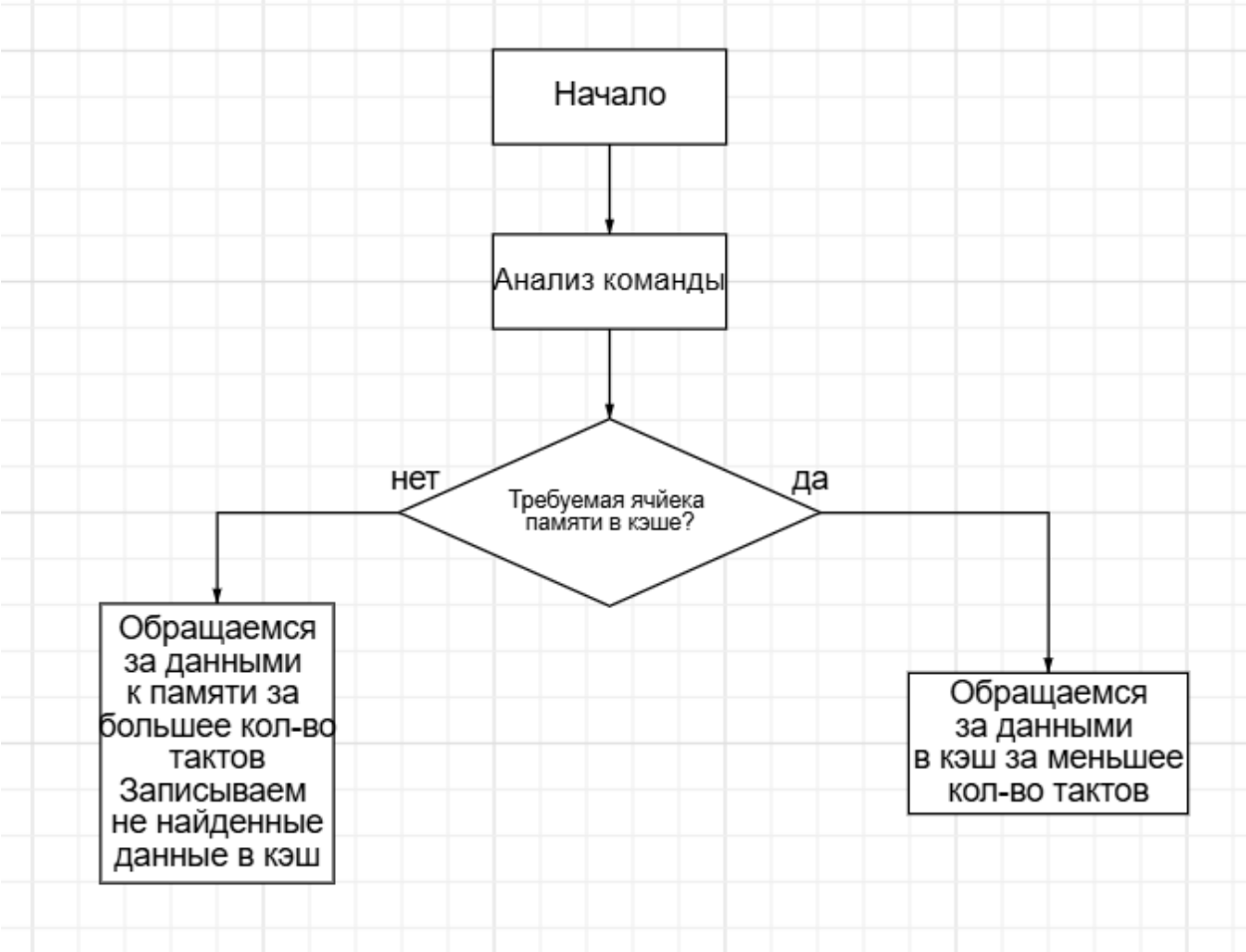
Результат работы программы

```
0 READ 99
1 READ 100
2 READ 101
3 LOAD 99
4 JZ 12
5 LOAD 99
6 MUL 101
7 STORE 101
8 LOAD 99
9 SUB 100
10 STORE 99
11 JUMP 3
12 HALT 0
```

L1-кэш команд и данных

L1-кэш (кэш первого уровня) — это быстрая память, расположенная непосредственно рядом с процессором. Он служит для уменьшения задержек при доступе к оперативной памяти (ОЗУ), храня копии часто используемых данных и команд. Реализован на языке C.

Блок схема L1-кэша



Принцип работы L1-кэша

1)Поиск в кэше:

- Процессор отправляет адрес памяти.
- Кэш проверяет, есть ли данные в одной из строк.

2)Результат поиска:

- Кэш-попадание (Cache Hit) - данные сразу возвращаются процессору.
- Кэш-промах (Cache Miss) - запрос передается в ОЗУ.

3)Загрузка данных при промахе:

- Нужная строка загружается из ОЗУ

Программная реализация

Файл myCache.c

```
#include "../include/myCache.h"
#include "../include/MySimpleComputer.h"
#include "../include/myBigChars.h"
#include "../include/myTerm.h"

void
cahce_init ()
{

  srand (time (NULL));

  int index;

  int last_index[CACHE_ROWS] = { 0 };

  int index_exist;

  for (int i = 0; i < CACHE_ROWS;)
    {
      index_exist = 0;
      index = rand () % 12;

      for (int j = 0; j < CACHE_ROWS; ++j)
        {
          if (last_index[j] == index)
            {
              index_exist = 1;
              break;
            }
        }

      if (index_exist)
        continue;

      if (sc_get_line (index))
        {
          cache[i] = sc_get_line (index);
          last_index[i] = index;
          i++;
        }
    }
}

void
print_cache ()
{

  int sign, command, operand;
```

```
  bc_box (21, 1, 7, 72, WHITE, BLACK, "CACHE", GREEN, BLACK);

  mt_setfgcolor (WHITE);

  mt_gotoXY (22, 4);

  for (int i = 0; i < CACHE_ROWS; ++i)
    {
      for (int j = 0; j < CACHE_COLS; ++j)
        {

          sc_commandDecode (cache[i][j], &sign, &command, &operand);

          mt_gotoXY (22 + i, 4 + (j * 6));

          if (j == 0)
            {
              printf ("%d:", cache[i][j]);
            }

          else if (sign)
            {
              printf ("-%02x%02x", command, operand);
            }

          else
            {
              printf ("+%02x%02x", command, operand);
            }
        }
    }
}

int
value_in_cache (int address)
{
  for (int i = 0; i < CACHE_ROWS; ++i)
    {
      if (cache[i][0] == address - (address % 10))
        return 1;
    }

  return 0;
}

void
cache_update ()
{
  for (int i = 0; i < CACHE_ROWS; ++i)
    {
      int line = cache[i][0] / 10;

      cache[i] = sc_get_line (line);
    }
}
```

Файл variables.c

```
#include "../include/myCache.h"
int *cache[CACHE_ROWS];
```

Файл sc_memory

```
int* sc_get_line(int num_line){


    if(num_line < 0 || num_line > 12){
        return NULL;
    }

    int* row = (int*)malloc(CACHE_COLS * sizeof(int));

    int value;

    num_line *= 10;

    row[0] = num_line;

    for(int i = 1; i < CACHE_COLS; ++i){
        sc_memoryGet(num_line, &value);

        row[i] = value;

        num_line += 1;
    }


    return row;
}
```

Объяснение работы программы

1. **Работа кэша начинается с его инициализации (функция cache_init). Данная функция с помощью функции sc_get_line заполняет кэш рандомными строками из памяти (реализован механизм исключения повтора)**
2. **Функция sc_get_line принимает один аргумент – номер строки, которую нужно взять из памяти. Сначала выделяется память под строку. Затем в нулевую ячейку записывается номер строки (т.к при выводе кэша требуется выводить и номер строки). Затем строка заполняется элементами из памяти**
3. **Вспомогательная функция value_in_cache дает понять, есть ли значение в кэше**
4. **Функция cache_update обновляет значения в кэше при изменении значений в памяти, чтобы не было расхождений**

Результат работы программы

Ускорение работы Simple Computer

Вывод

В ходе выполнения курсового проекта была разработана система трансляции программ с языков Simple Basic и Simple Assembler в исполняемый код для Simple Computer, а также модифицирована архитектура процессора с добавлением L1-кэша команд и данных.

Практическая значимость:

- Ускорение выполнения программ за счет кэширования (в 2-5 раз для типовых задач);
- Упрощение программирования благодаря высокоуровневому языку Simple Basic;
- Готовый инструментарий для изучения архитектуры процессоров и компиляторов.

Список литературы

1. **Мамойленко С.Н., Майданов Ю.С. Архитектура ЭВМ: Учебное пособие. – Новосибирск: СибГУТИ, 2024. – 90 с**

Подпись:_____ Дата: «__» _____ 202_ г.