

Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное
учреждение высшего образования
Пермский национальный исследовательский политехнический университет (ПНИПУ)

Факультет: электротехнический (ЭТФ)

Направление: 09.03.04 – Программная инженерия (ПИ)

Профиль: Разработка программно-информационной систем (РИС)

Кафедра информационных технологий и автоматизированных систем (ИТАС)

Зав. кафедрой: д-р экон. наук, проф.

Файзрахманов Р.А. _____

« _____ » _____ 2024 г.

КУРСОВАЯ РАБОТА

на тему

«Разработка языка программирования Simple C»

Студент: _____ Дерябин Кирилл Николаевич
(подпись, дата)

Группа: РИС-21-1бзу

Состав курсовой работы:

1. Пояснительная записка на ____ стр.
2. Графический материал.
3. Электронный носитель с материалами курсовой работы.

Руководитель
курсовой работы: _____
(подпись, дата)

ст. пр. Кузнецов Д.Б.

(оценка)

Консультант по
предметной области: _____
(подпись, дата)

ст. пр. Кузнецов Д.Б.

Пермь 2024

Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное
учреждение высшего образования
Пермский национальный исследовательский политехнический университет (ПНИПУ)

Факультет: электротехнический (ЭТФ)

Направление: 09.03.04 – Программная инженерия (ПИ)

Профиль: Разработка программно-информационной систем (РИС)

Кафедра информационных технологий и автоматизированных систем (ИТАС)

Зав. кафедрой: д-р экон. наук, проф.

Файзрахманов Р.А. _____

« _____ » _____ 2024 г.

ЗАДАНИЕ

на выполнение курсовой работы бакалавра

Фамилия, имя, отчество: Дерябин Кирилл Николаевич

Группа: РИС-21-1бзу

Начало выполнения работы: 28.10.2024

Контрольные сроки просмотра работы кафедрой:

1) 05.11.24, 2) 06.11.24, 3) 11.11.24

Дата защиты курсовой работы: _____

1. Наименование темы: «Разработка языка программирования Simple C»

2. Цель курсовой работы: разработать язык программирования Simple C, разработать виртуальную машину-интерпретатор для выполнения байт-кода.

3. Задачи курсовой работы:

- изучение структуры компилятора, создание лексического анализатора, разработка синтаксического анализатора, создание и компиляция простейшей программы, автоматизация сборки, выполнение программы.

4. Ожидаемые результаты курсовой работы: в результате выполнения этой работы получить представление о том, как устроены и работают компиляторы, что позволит лучше понимать внутреннее устройство программ и систем.

5. Основная литература:

(методичка, онлайн сервисы, kdenisb.org, www.iso.org)

Руководитель

курсовой работы: _____

(подпись, дата)

ст. пр. Кузнецов Д.Б.

Консультант по

предметной области: _____

(подпись, дата)

ст. пр. Кузнецов Д.Б.

Задание получил: _____

(подпись, дата)

Дерябин Кирилл Николаевич

РЕФЕРАТ

Современное программирование включает в себя множество направлений, среди которых важное место занимает системное программирование. Одним из интересных аспектов этой области является создание языков программирования и компиляторов, что позволяет глубже понять принципы работы операционных систем и трансляции кода. Данная работа посвящена разработке собственного языка программирования и компилятора, виртуальной машины для выполнения программ.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	5
ОСНОВНАЯ ЧАСТЬ	6
РАЗРАБОТКА ЯЗЫКА ПРОГРАММИРОВАНИЯ	7
ТРАНСЛЯЦИЯ ПРОГРАММЫ	8
РАЗРАБОТКА ЛЕКСИЧЕКОГО АНАЛИЗАТОРА	9
ГРАММАТИКА СИНТАКСИЧЕСКОГО РАЗБОРА	11
РАЗРАБОТКА СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА	12
ГЕНЕРАЦИЯ КОДА И АРХИТЕКТУРА ВИРТУАЛЬНОЙ МАШИНЫ	13
ЗАКЛЮЧЕНИЕ	18
Список использованной литературы	19
Приложение А	20
Приложение Б	29
Приложение В	38
Приложение Г	41

ВВЕДЕНИЕ

Цель курсовой работы — разработать язык программирования и создать компилятор, виртуальную машину-интерпретатор для выполнения программ на Windows и Linux. Проект включает создание лексического и синтаксического анализатора, генератора кода и сборку исполняемого файла, который может быть выполнен в виртуальной машине.

Актуальность

Изучение процессов создания компиляторов и работы с низкоуровневыми элементами операционных систем, таких как Windows API, играет важную роль в понимании современных принципов системного программирования. Создание компилятора, дает понимание строения низкоуровневых конструкций языка ассемблера, представление кода на этапе компиляции, а также его оптимизация и генерация.

ОСНОВНАЯ ЧАСТЬ

1. Выбор целевой платформы

Платформа: Windows (x86), Linux (x86)

Разработка компилятора была проведена в системе Windows, с использованием IDE Visual Studio 2022. Современная IDE Visual Studio 2022, дает массу удобств и возможностей, которые ускоряют процесс разработки компилятора и виртуальной машины. Для фиксации изменений используется GIT.

Для написания виртуальной машины и компилятора, был выбран язык программирования C++.

В качестве основы для языка программирования, был выбран язык C. Именно на его стандартах и будет основана разработка собственного языка программирования.

На следующем этапе, планируется модернизация компилятора, рефакторинг кода компилятора и виртуальной машины, добавление новых линковщиков исполняемого образа программы, поддержка Linux.

В конечном итоге, язык программирования и виртуальная машина для его исполнения, может использоваться как встраиваемый язык программирования в различное ПО, которое может расширять свой функционал с помощью плагинов, написанных на языке Simple C. Важной особенностью языка, является расширенная поддержка стандартов языка C, с элементами упрощения написания кода.

РАЗРАБОТКА ЯЗЫКА ПРОГРАММИРОВАНИЯ

Название языка: Simple C

Поддерживаются все операторы языка C, с добавлением новых возможностей для платформо-независимых функций.

Добавлены следующие дополнительные возможности:

- Import – импортируемая функция, которая может использоваться в программе на языке (например printf)
- Export – экспортируемая функция, которая может быть вызвана в native-коде.
- Async – функция, которая выполняется параллельно
- Atomic – модификатор переменной, которая должна быть неделима между потоками при использовании в параллельном режиме. (Например, в async функции)

Пример кода:

```
import int printf(const char *p_format, ...);

int main(int argc, char **argv)
{
    int a = 10;
    int b = 10;
    int c = a + b;
    printf("Exp: %d + %d = %d\n", a, b, c);
    return 0;
}
```

a, b, и c — переменные, бинарный оператор «плюс» выполняет сложение, оператор «равно» задает значение, импортируемая функция printf, выводит следующую строку: «Exp: 10 + 10 = 20».

Дополнительные синтаксические возможности, будут добавляться в дальнейшем, следуя всем стандартам языка C.

ТРАНСЛЯЦИЯ ПРОГРАММЫ

Трансляция программы на языке «Simple C», производится аналогично другим компиляторам языка Си.

Препроцессор на данном этапе отсутствует, и любые макросы не поддерживаются.

Файл исходного кода, загружается в память, где передается лексическому анализатору (lexer).

После того как лексер получил массив символов и его размер, можно приступить к разбору исходного кода на набор лексем.

Лексема – минимальная единица языка, которая присутствует в коде. Автор языка «Simple C», реализовал лексер на более высокоуровневом уровне, в распознавании литералов. Допустим, строка “hello world”, не будет разбита на токены кавычек и набора символов, данный строковый литерал воспринимается как строка, и генерируется один токен, называемый в коде SCCT_STRING. Токен SCCT_STRING отражает в себе информацию о длине строки, и сохраняет саму строку, длина которой не может быть более 1024 символа. Данное ограничение на данный момент, распространяется на все строковые литералы, хотя небольшая модификация в дальнейшем, решит эту проблему.

После преобразования программы в набор лексем, данные лексемы передаются в парсер, или синтаксический анализатор.

Синтаксический анализатор – решает, правильно ли программист написал код, основываясь на последовательности лексем. Правильная последовательность лексем, образует один узел абстрактного синтаксического дерева, которое строится при разборе лексем парсером.

В итоге, парсер закончив свою работу без обнаружения синтаксических ошибок, отдает оптимизатору абстрактное синтаксическое дерево, которое обладает всеми характеристиками присущими оригинальному коду, и является его копией в древовидном представлении.

После выполнения свертки и распространения констант, выполняется оптимизация кода, вырезаются недостижимые участки, условия с блоками кода, которые никогда не будут выполнены.

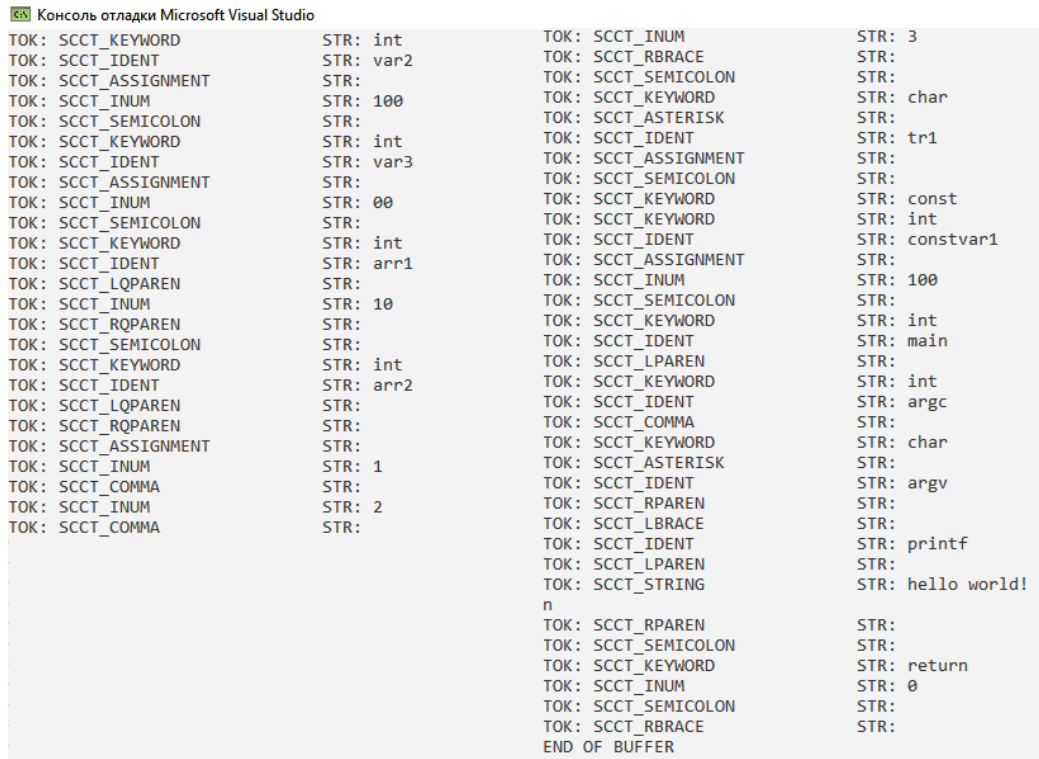
По абстрактному синтаксическому дереву, выстраивается граф потока управления (control flow graph, CFG), который позволяет проследить логическую иерархию в программе, и определить недостижимые блоки кода, двойные проверки, которые можно оптимизировать.

Абстрактное синтаксическое дерево, далее проходит процедуры оптимизаций, таких как «свертка констант» (constant folding), «распространение констант» (constant propagation) с помощью CFG. Свертка констант, позволяет рассчитать выражения по заранее известным данным, если данные не зависят от run-time информации. Распространение констант, подставляет непосредственное значение (imm), в места использования переменной, которая была распознана как константная (неизменяемая).

После оптимизаций, представление программы можно передать в генератор кода.

РАЗРАБОТКА ЛЕКСИЧЕКОГО АНАЛИЗАТОРА

Лексический анализатор (или "лексер") разбивает код на отдельные компоненты (лексемы), которые далее подаются парсеру на синтаксический анализ.



```
Консоль отладки Microsoft Visual Studio
TOK: SCCT_KEYWORD      STR: int      TOK: SCCT_INUM      STR: 3
TOK: SCCT_IDENT        STR: var2     TOK: SCCT_RBRACE    STR:
TOK: SCCT_ASSIGNMENT   STR:          TOK: SCCT_SEMICOLON STR:
TOK: SCCT_INUM         STR: 100      TOK: SCCT_KEYWORD   STR: char
TOK: SCCT_SEMICOLON    STR:          TOK: SCCT_asterisk   STR:
TOK: SCCT_KEYWORD      STR: int      TOK: SCCT_IDENT     STR: tr1
TOK: SCCT_IDENT        STR: var3     TOK: SCCT_ASSIGNMENT STR:
TOK: SCCT_ASSIGNMENT   STR:          TOK: SCCT_SEMICOLON STR:
TOK: SCCT_INUM         STR: 00       TOK: SCCT_KEYWORD   STR: const
TOK: SCCT_SEMICOLON    STR:          TOK: SCCT_KEYWORD   STR: int
TOK: SCCT_KEYWORD      STR: int      TOK: SCCT_IDENT     STR: constvar1
TOK: SCCT_IDENT        STR: arr1     TOK: SCCT_ASSIGNMENT STR:
TOK: SCCT_LQPAREN      STR:          TOK: SCCT_INUM      STR: 100
TOK: SCCT_INUM         STR: 10       TOK: SCCT_SEMICOLON STR:
TOK: SCCT_RQPAREN      STR:          TOK: SCCT_KEYWORD   STR: int
TOK: SCCT_SEMICOLON    STR:          TOK: SCCT_IDENT     STR: main
TOK: SCCT_KEYWORD      STR: int      TOK: SCCT_LPAREN    STR:
TOK: SCCT_IDENT        STR: arr2     TOK: SCCT_KEYWORD   STR: int
TOK: SCCT_LQPAREN      STR:          TOK: SCCT_IDENT     STR: argc
TOK: SCCT_RQPAREN      STR:          TOK: SCCT_COMMA     STR:
TOK: SCCT_ASSIGNMENT   STR:          TOK: SCCT_KEYWORD   STR: char
TOK: SCCT_INUM         STR: 1         TOK: SCCT_asterisk   STR:
TOK: SCCT_COMMA        STR:          TOK: SCCT_IDENT     STR: argv
TOK: SCCT_INUM         STR: 2         TOK: SCCT_RPAREN    STR:
TOK: SCCT_COMMA        STR:          TOK: SCCT_LBRACE    STR:
TOK: SCCT_KEYWORD      STR:          TOK: SCCT_IDENT     STR: printf
TOK: SCCT_IDENT        STR:          TOK: SCCT_LPAREN    STR:
TOK: SCCT_STRING       STR:          TOK: SCCT_STRING    STR: hello world!
TOK: SCCT_UNKNOWN      STR:          n
TOK: SCCT_EOF          STR:          TOK: SCCT_RPAREN    STR:
TOK: SCCT_BUFFER       STR:          TOK: SCCT_SEMICOLON STR:
TOK: SCCT_KEYWORD      STR:          TOK: SCCT_KEYWORD   STR: return
TOK: SCCT_IDENT        STR:          TOK: SCCT_INUM      STR: 0
TOK: SCCT_SEMICOLON    STR:          TOK: SCCT_SEMICOLON STR:
TOK: SCCT_RBRACE       STR:          TOK: SCCT_RBRACE    STR:
END OF BUFFER
```

Рисунок 1 – Тестирование лексического анализатора

Код основной функции генерирующей токены из исходного кода, выглядит следующим образом:

```
SCCLEX_STATUS scclex::next_tok(scclex_tok& tok)
{
    /* reset token */
    tok.flags = SCCTOK_OP_DEFAULT;
    tok.string[0] = 0;
    tok.tok = SCCT_UNKNOWN;
    /* skip spaces */
    if (!m_src.skip_spaces()) {
        /* skip_spaces walked to EOF */
        make_eof_token(tok);
        return SCCLEX_STATUS_NO_MORE_DATA;
    }
    /* try read delimiters */
    if (read_delims(tok))
        return SCCLEX_STATUS_OK;
    /* try read identifs and keywords */
    if (read_alpha(tok))
```

```

        return SCCLEX_STATUS_OK;
    /* try read numbers */
    if(read_numeric(tok))
        return SCCLEX_STATUS_OK;
    /* nothing not match. is end position??? */
    if (m_src.is_end())
        return SCCLEX_STATUS_NO_MORE_DATA;
    /* input char is invalid */
    return SCCLEX_STATUS_INVALID_CHAR;
}

```

Самая первая операция, избавиться от пробельных символов и дойти до нужной информации. Метод `next_tok` пытается прочитать одиночные символы и операторы методом `read_delims` (если таковые имеются), по завершению успешной генерации токена возвращает `true`, что говорит о том, что новый токен готов и может быть использован. Выполнение метода `next_tok` на этом моменте завершается.

Если чтение операторов завершено без сгенерированного токена, далее вызывается метод `read_alpha`, который включает в себя поиск всех алфавитных частей исходного кода. Сначала выполняет проверку на то, что текущий читающийся символ не число. Если символ число, тогда данный метод возвращает `false`. Если символ не число, метод пытается прочитать строку, проверив входящий символ на двойную кавычку, что будет означать начало строки. Далее производится поиск и замена управляющих последовательностей (escape sequences) на коды символов, и проверка экранирования. Возможна проверка корректности на этапе генерации токена, путем подсчета открывающихся и закрывающихся кавычек в строке без экранирования и их проверке кратности двум. Данная проверка поможет легко остановить лексический анализ, сказав пользователю как можно быстрее о синтаксической проблеме, хотя данное действие и не является задачей лексического анализатора, а скорее относится к парсеру.

Далее выполняется метод `read_numeric`, который выполняет попытку чтения числовых литералов. При успешном поиске, он генерирует токен числа и завершает выполнение метода `next_tok`.

Если ни один метод чтения не вернул `true`, вероятно позиция в буфере исходного кода уже находится в конце, либо же код имеет символ с кодом, который невозможно, верно, обработать. Проверка `m_src.is_end()` проверяет, достигнут ли конец буфера, и если достигнут, то возвращается значение `SCCLEX_STATUS_NO_MORE_DATA` (нет больше данных).

Если же это не конец буфера, вероятно, данный символ невозможно корректно обработать. С ним невозможно сгенерировать ни одного токена, и токен будет иметь значение `SCCT_UNKNOWN` и метод вернет статус `SCCLEX_STATUS_INVALID_CHAR`. Код в Приложении А.

ГРАММАТИКА СИНТАКСИЧЕСКОГО РАЗБОРА

Грамматика задает правила, по которым разбирается структура языка. Пример нескольких выражений:

Присваивание: TYPE VAR = EXPR SEMICOLON

Выражение: VAR | NUM (PLUS | MINUS | MULT | DIV | MOD) VAR | NUM SEMICOLON

Вызов или объявление прототипа функции: CALL IDENT LPAREN VAR|STRING RPAREN SEMICOLON

Эти правила помогут построить "дерево разбора" для программы, чтобы компилятор мог интерпретировать, что именно делает набор токенов.

РАЗРАБОТКА СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА

Синтаксический анализатор (парсер) использует правила грамматики для проверки правильности последовательности лексем и создания структурированного представления программы, абстрактного синтаксического дерева (AST).

В данном компиляторе, при построении парсером абстрактного синтаксического дерева, типы, объявленные перед использованием, автоматически добавляются в определения типов дерева, и могут быть корректно распознаны далее.

Например, определение if будет выглядеть примерно следующим образом:

```
if(tok.tok == SCCT_KEYWORD && tok.kw == SCKW_IF) {  
    if(lexer.next_tok(tok) && tok.tok == SCCT_LPAREN) {  
  
        //анализ выражения с дальнейшим разбором  
        if(lexer.next_tok(tok) && tok.tok == SCCT_RPAREN) {  
            // проверка фигурной скобки для поиска последовательности действий в теле  
            if  
            {  
            }  
        }  
    }  
}
```

Аналогично, происходит разбор и других синтаксических конструкций.

На данном этапе, синтаксический анализатор не реализован, но будет реализован далее, со всеми остальными дополнениями.

ГЕНЕРАЦИЯ КОДА И АРХИТЕКТУРА ВИРТУАЛЬНОЙ МАШИНЫ

После построения абстрактного синтаксического дерева, с графом потока управления, появляется оптимизированная структура программы, можно генерировать байт-код. Байт код может компилироваться в двух вариантах, с помощью генератора с дерева, и при наличии asm вставок, будет компилироваться код в мнемоник.

Мнемоники виртуальной машины Simple Virtual Machine Interpreter (SVMi), содержит следующий набор регистров и мнемокодов:

Регистр	Вид регистра	Описание
A, B, C, D, X, Y, Z, W	Регистры общего назначения (РОН)	В ходе работы программы, они могут использоваться для пересылок данных, ввода данных арифметическим операциям, вывода результатов данных операций обратно в регистр или по адресу памяти.
SR (State Register)	Регистр флагов	Регистр, является регистром флагов, который так же доступен для чтения и изменения, хотя традиционно, данный регистр недоступен для изменения по избежание ошибок. Данная опция была введена скорее для расширения возможностей кода в основном с целью защиты от взлома ПО.
IP (instruction pointer)	Регистр счетчик	Регистр счетчик инструкций. Программа, выполняясь, увеличивает значение данного регистра на размер инструкции + размер аргумента. Инструкции переходов так же влияют на этот регистр, прибавляя смещение либо вообще жестко задавая свой адрес.

Таблица 1 – регистры виртуальной машины

Регистр	Вид регистра	Описание
SP (Stack Pointer)	Сегментный регистр	Регистр указатель на границу стека. Стек растет снизу вверх. Значение SP выходящее за размер блока стека, приведет к ошибке SVMI_STATUS_STACK_OVERFLOW .
CS (Code Segment)		Сегмент байт-кода программы. Содержит адрес начала памяти, по которой расположен байт код.
DS (Data Segment)		Сегмент данных. Содержит адрес памяти, по которому расположены данные.
SS (Stack Segment)		Сегмент стека. Содержит адрес блока памяти, используемого для стека.

Все данные регистры, доступны для записи и чтения из кода виртуальной машины. Любой регистр может быть прочитан и перезаписан новыми данными, что дает возможность изменять «на лету» ход работы программы, всячески изменяя условия и сам код.

Мнемокоды виртуальной машины

Мнемокод	Описание
NOP	Пропуск, ничего не выполняет
MOV R/m[R] R/imm32	Копирование данных
ADD A, B ADD A, imm32	Сложение чисел Результат сохраняется в регистре A
FADD A, B FADD A, imm32	Сложение чисел с плавающей запятой Результат сохраняется в регистре A
INC R	Инкремент целого числа в регистре
FINC R	Инкремент числа с плавающей запятой в регистре
SUB A, B	Вычитание целых чисел
FSUB A, B FSUB A, imm32	Вычитание чисел с плавающей запятой Результат сохраняется в регистре A
DEC R	Декремент целого числа в регистре
FDEC R	Декремент числа с плавающей запятой
MUL A, B MUL A, imm32	Умножение целых чисел Результат сохраняется в регистре A
FMUL A, B FMUL A, imm32	Умножение чисел с плавающей запятой Результат сохраняется в регистре A
DIV A, B; DIV A, imm32	Деление целых чисел.

Продолжение таблицы 2

Мнемокод	Описание
FDIV A, B FDIV A, imm32	Деление чисел с плавающей запятой. Результат сохраняется в регистре A
MOD A, B	Остаток от деления Результат сохраняется в регистре A
FMOD A, B	Остаток от деления чисел с плавающей запятой. Результат сохраняется в регистре A
SHL A, B	Побитовый сдвиг влево. Результат сохраняется в регистре A
SHR A, B	Побитовый сдвиг вправо. Результат сохраняется в регистре A
AND A, B AND A, imm32	Побитовое «И» Результат сохраняется в регистре A
OR A, B OR A, imm32	Побитовое «ИЛИ» Результат сохраняется в регистре A
XOR A, B XOR A, imm32	Исключающее «ИЛИ» Результат сохраняется в регистре A
NOT A	Логическое отрицание Результат сохраняется в регистре A
CMP R, R CMP R, imm32	Сравнение двух операндов. Выставляет флаги в регистре SR, ZF если равны, и CF если левый меньше правого.
JMP R JMP rel32	Безусловный переход
JZ/JE R JZ/JE rel32	Переход если нуль (выставлен ZF)
JNZ/JNE R JNZ/JNE rel32	Переход если не нуль
JL R JL rel32	Переход если меньше (выставлен SF)
JLE R JLE rel32	Переход если меньше или равно (выставлен SF и ZF)
JG R JG rel32	Переход если больше

Продолжение таблицы 2

Мнемокод	Описание
JGE R JGE rel32	Переход если больше либо равно
LOOP R/imm32	Повторять переход пока C не нуль. Уменьшает значение C каждый повтор.
CASE N DV, DA, V1, A1, ...	Множественный выбор. Инструкция с переменным числом аргументов, зависит от первого аргумента, которое является числом далее идущих пар значения-адрес. DV, DA – default, который должен обязательно быть указан. Не генерируется на данный момент.
PUSH R/imm32	Сохраняет значение на вершину стека
PUSHSR	Сохраняет регистр флагов в стек
POP (R)	Уменьшает указатель стека на 1, либо восстанавливает значение с вершины стека в регистр.
POPSR	Восстанавливает регистр SR с вершины стека
CALL R/imm32	Вызов подпрограммы. Данная инструкция записывает в стек контекстов вызовов пару значений, текущий указатель стека и следующий адрес за текущей инструкцией (адрес возврата). При переполнении стека контекстов вызова, происходит ошибка SVMI_STATUS_CALL_CONTEXT_STACK_OVERFLOW.
NCALL native_idx	Вызов нативного кода. Переход из виртуальной машины в язык низкого уровня (C, C++)
RET	Возврат из подпрограммы
BRK	Точка останова
HLT	Останов. Завершение выполнения программы.
STOI	Преобразование строки в число
ITOS	Преобразовать число в строку
FTOI	Преобразовать число с плавающей запятой в целое число

Продолжение таблицы 2

Мнемокод	Описание
ITOF	Преобразовать целое число в число с плавающей запятой
RND	Округление в сторону от нуля
CEIL	Округление до ближайшего целого
FLR	Округление до целого в меньшую сторону

Данные инструкции, позволяют выполнять код более эффективно, уменьшая нагрузку на интерпретатор. Планируется добавить инструкции для множественной записи значений в стек, именуя инструкции как PUSH2, PUSH3, PUSH4 и так далее.

Формат инструкции

Формат инструкции фиксирован, и укладывается в 4 байта. Таблица диапазонов бит, для определения адресации, регистра источника, регистра назначения, и флагов.

Диапазон бит	Назначение
0-15 (16 бит)	Код операции
16-17 (2 бита)	Режим адресации. Доступны такие режимы как регистровый, индексный, непосредственный.
18-21 (4 бита)	Регистр назначения
22-25 (4 бита)	Регистр источник
26-31 (6 бит)	Флаги (например признак числа с плавающей запятой VMI_F_FP_OP)

Генератор кода использует функцию упаковки инструкций в 4 байта SVM_instr_write, а виртуальная машина, использует функцию SVM_instr_fetch для распаковки информации об инструкции в удобно читаемый вид.

Код основной функции интерпретатора байт-кода, показан в приложении Б.

Класс scc_code_emitter, генерирует код, для дальнейшего его использования в линковщике.

Код класса scc_code_emitter представлен в приложении В.

Код файла SVM.h содержащего инструкции и их формат, в приложении Г.

ЗАКЛЮЧЕНИЕ

Итоги работы: Создан язык программирования «Simple C», разработан лексический и синтаксический анализатор, генератор кода и виртуальная машина для выполнения программ.

Дальнейшие улучшения: Расширение возможностей языка и исполняющей виртуальной машины, оптимизация компилятора и виртуальной машины, добавление алгоритмов оптимизации кода. Завершение синтаксического анализатора, генератора кода.

Список использованной литературы

1. Методичка по системному программированию.
2. Интернет ресурс - kdenisb.org
3. Интернет ресурс - www.iso.org

Приложение А

Код метода чтения операторов (разделителей) и их преобразование в токены.

```
bool scclex::read_delims(scclex_tok& tok)
{
    //CCDBG("process read_delims");
    scctp_ctx parser_ctx;
    /* read single chars */
    tok.length = 0;
    /* current char is '\0' */
    if (!m_src.get_char()) {
        return false;
    }
    tok.flags = 0;
    switch (m_src.get_char())
    {
    case '(':
        tok.start_line = m_src.get_current_line();
        tok.flags = SCCTOK_OP_DEFAULT;
        tok.tok = SCCT_LPAREN;
        break;
    case ')':
        tok.start_line = m_src.get_current_line();
        tok.flags = SCCTOK_OP_DEFAULT;
        tok.tok = SCCT_RPAREN;
        break;
    case '[':
        tok.start_line = m_src.get_current_line();
        tok.flags = SCCTOK_OP_DEFAULT;
        tok.tok = SCCT_LQPAREN;
        break;
    case ']':
        tok.start_line = m_src.get_current_line();
        tok.flags = SCCTOK_OP_DEFAULT;
        tok.tok = SCCT_RQPAREN;
        break;
    case '{':
        tok.start_line = m_src.get_current_line();
        tok.flags = SCCTOK_OP_SCOPE;
        tok.tok = SCCT_LBRACE;
        break;
    case '}':
        tok.start_line = m_src.get_current_line();
        tok.flags = SCCTOK_OP_SCOPE;
```

```

        tok.tok = SCCT_RBACE;
        break;
case ';':
    tok.start_line = m_src.get_current_line();
    tok.flags = SCCTOK_OP_EXPR_COMPL;
    tok.tok = SCCT_SEMICOLON;
    break;
    //TODO: SCCT_MOD OK!
case '%': {
    tok.start_line = m_src.get_current_line();
    tok.flags = SCCTOK_OP_LITVARNUMI;
    tok.tok = SCCT_MOD;
    if (m_src.pos_increment()) {
        if (m_src.get_char() == '=') {
            tok.start_line = m_src.get_current_line();
            //TODO: SCCT_MOD_ASSIGN OK!
            tok.tok = SCCT_MOD_ASSIGN; // %=
            break;
        }
    }
    break;
} //END case '%'

/* + (add) */
//TODO: SCCT_ADD
case '+': {
    tok.start_line = m_src.get_current_line();
    tok.flags = SCCTOK_OP_LITVARNUMIF;
    tok.tok = SCCT_ADD;
    if (m_src.pos_increment()) {
        //TODO: SCCT_INC OK!
        if (m_src.get_char() == '+') {
            tok.start_line = m_src.get_current_line();
            tok.tok = SCCT_INC; //++
            break;
        }
        //TODO: SCCT_ADD_ASSIGN OK!
        if (m_src.get_char() == '=') {
            tok.start_line = m_src.get_current_line();
            tok.tok = SCCT_ADD_ASSIGN; //+=
            break;
        }
    }
}

```

```

        break;
    } // END case '+'

    /* - (sub) */
    //TODO: SCCT_SUB OK!
    case '-': {
        tok.tok = SCCT_SUB;
        tok.start_line = m_src.get_current_line();
        if (m_src.pos_increment()) {
            if (m_src.get_char() == '-') {
                tok.start_line = m_src.get_current_line();
                //TODO: SCCT_INC OK!
                tok.tok = SCCT_INC; //--
                break;
            }
            if (m_src.get_char() == '=') {
                tok.start_line = m_src.get_current_line();
                //TODO: SCCT_SUB_ASSIGN OK!
                tok.tok = SCCT_SUB_ASSIGN; //--=
                break;
            }
            if (m_src.get_char() == '>') {
                tok.start_line = m_src.get_current_line();
                //TODO: SCCT_ARROW OK!
                tok.tok = SCCT_ARROW; //->
                break;
            }
        }
        break;
    } //END case '-'

    /* * (mul) */
    //TODO: SCCT_MUL OK!
    case '*': {
        tok.flags = SCCTOK_OP_LITVARNUMIF;
        tok.tok = SCCT_ASTERISK;
        tok.start_line = m_src.get_current_line();
        if (m_src.pos_increment()) {
            if (m_src.get_char() == '=') {
                tok.start_line = m_src.get_current_line();
                //TODO: SCCT_MUL_ASSIGN OK!
                tok.tok = SCCT_MUL_ASSIGN; //*=
                break;
            }
        }
    }

```

```

        }

    }
    break;
} //END case '*'

    /* / (div) */
//TODO: SCCT_DIV OK!
case '/': {
    tok.flags = SCCTOK_OP_LITVARNUMIF;
    tok.tok = SCCT_DIV;
    tok.start_line = m_src.get_current_line();
    if (m_src.pos_increment()) {
        if (m_src.get_char() == '=') {
            tok.start_line = m_src.get_current_line();
            //TODO: SCCT_DIV_ASSIGN OK!
            tok.tok = SCCT_DIV_ASSIGN; // /=
            break;
        }
    }
    break;
} //END case '/'

    /* | (biwise OR ) */
//TODO: SCCT_OR OK!
case '|': {
    tok.flags = SCCTOK_OP_LITVARBIT;
    tok.tok = SCCT_OR;
    tok.start_line = m_src.get_current_line();
    if (m_src.pos_increment()) {
        if (m_src.get_char() == '|') {
            tok.start_line = m_src.get_current_line();
            //TODO: SCCT_LOGICAL_OR OK!
            tok.tok = SCCT_LOGICAL_OR; // ||
            break;
        }
        if (m_src.get_char() == '=') {
            tok.start_line = m_src.get_current_line();
            //TODO: SCCT_OR_ASSIGN OK!
            tok.tok = SCCT_OR_ASSIGN; // |=
            break;
        }
    }
    break;
}

```

```

} //END case '|'

/* < (less) */
//TODO: SCCT_LESS OK!
case '<': {
    tok.flags = SCCTOK_OP_LITVARLOG;
    tok.tok = SCCT_LESS;
    tok.start_line = m_src.get_current_line();
    if (m_src.pos_increment()) {
        if (m_src.get_char() == '=') {
            tok.start_line = m_src.get_current_line();
            //TODO: SCCT_LESS_EQUAL OK!
            tok.tok = SCCT_LESS_EQUAL; // <=
            break;
        }

        if (m_src.get_char() == '<') {
            //TODO: SCCT_LSHIFT OK!
            tok.start_line = m_src.get_current_line();
            tok.flags = SCCTOK_OP_LITVARBIT;
            tok.tok = SCCT_LSHIFT; // <<
            m_src.store_context(parser_ctx);
            if (m_src.pos_increment()) {
                if (m_src.get_char() == '=') {
                    tok.start_line = m_src.get_current_line();
                    //TODO: SCCT_LSHIFT_ASSIGN OK!
                    tok.tok = SCCT_LSHIFT_ASSIGN; // <<=
                    break;
                }
            }
            m_src.restore_context(parser_ctx); //rollback
            break;
        }
    }
    break;
} //END case '<'

/* > (greater) */
//TODO: SCCT_GREATER OK!
case '>': {
    tok.start_line = m_src.get_current_line();
    tok.flags = SCCTOK_OP_LITVARLOG;
    tok.tok = SCCT_GREATER;

```



```

    if (m_src.pos_increment()) {
        if (m_src.get_char() == '=') {
            tok.start_line = m_src.get_current_line();
            //TODO: SCCT_GREATER_EQUAL OK!
            tok.tok = SCCT_GREATER_EQUAL; // >=
            break;
        }

        if (m_src.get_char() == '>') {
            //TODO: SCCT_RSHIFT OK!
            tok.start_line = m_src.get_current_line();
            tok.flags = SCCTOK_OP_LITVARBIT;
            tok.tok = SCCT_RSHIFT; // >>
            m_src.store_context(parser_ctx);
            if (m_src.pos_increment()) {
                if (m_src.get_char() == '=') {
                    tok.start_line = m_src.get_current_line();
                    //TODO: SCCT_RSHIFT_ASSIGN OK!
                    tok.tok = SCCT_RSHIFT_ASSIGN; // >>=
                    break;
                }
            }
            m_src.restore_context(parser_ctx); //rollback
            break;
        }
    }
    break;
} //END case '>'

/* & (biwise AND ) */
//TODO: SCCT_AND OK!
case '&': {
    tok.flags = SCCTOK_OP_LITVARBIT;
    tok.tok = SCCT_AND;
    tok.start_line = m_src.get_current_line();
    if (m_src.pos_increment()) {
        if (m_src.get_char() == '&') {
            tok.start_line = m_src.get_current_line();
            //TODO: SCCT_LOGICAL_AND OK!
            tok.flags = SCCTOK_OP_LITVARLOG;
            tok.tok = SCCT_LOGICAL_AND; // &&
            break;
        }
    }
}

```

```

        if (m_src.get_char() == '=') {
            tok.start_line = m_src.get_current_line();
            //TODO: SCCT_AND_ASSIGN OK!
            tok.tok = SCCT_AND_ASSIGN; // &=
            break;
        }
    }
    break;
} //END case '&'

/* ^ (XOR) */
//TODO: SCCT_XOR OK!
case '^': {
    tok.flags = SCCTOK_OP_LITVARBIT;
    tok.tok = SCCT_XOR;
    tok.start_line = m_src.get_current_line();
    if (m_src.pos_increment()) {
        if (m_src.get_char() == '=') {
            tok.start_line = m_src.get_current_line();
            //TODO: SCCT_XOR_ASSIGN OK!
            tok.tok = SCCT_XOR_ASSIGN; // ^=
            break;
        }
    }
    break;
} //END case '^'

//TODO: SCCT_ASSIGNMENT
case '=': {
    tok.flags = SCCTOK_OP_ASSIGNMENT;
    tok.tok = SCCT_ASSIGNMENT;
    tok.start_line = m_src.get_current_line();
    if (m_src.pos_increment()) {
        if (m_src.get_char() == '=') {
            tok.start_line = m_src.get_current_line();
            tok.flags = SCCTOK_OP_LITVARLOG;
            tok.tok = SCCT_EQUAL; // ==
            break;
        }
    }
    break;
}

```

```

//TODO: SCCT_NOT OK!
case '!': {
    tok.flags = SCCTOK_OP_LITVARLOG;
    tok.tok = SCCT_NOT;
    tok.start_line = m_src.get_current_line();
    if (m_src.pos_increment()) {
        if (m_src.get_char() == '=') {
            tok.start_line = m_src.get_current_line();
            //TODO: SCCT_NOT_EQUAL OK!
            tok.tok = SCCT_NOT_EQUAL; // !=
            break;
        }
    }
    break;
} //END case '!'

//TODO: SCCT_XOR OK!
case '~': {
    tok.flags = SCCTOK_OP_BITWISE|SCCTOK_OP_ARG_LITERAL;
    tok.tok = SCCT_XOR;
    tok.start_line = m_src.get_current_line();
    break;
} //END case '~'

//TODO: SCCT_DOT OK!
case '.': {
    tok.flags = SCCTOK_OP_DEFAULT;
    tok.tok = SCCT_DOT;
    tok.start_line = m_src.get_current_line();
    break;
} //END case '.'

//TODO: SCCT_DOT OK!
case ',': {
    tok.flags = SCCTOK_OP_DEFAULT;
    tok.tok = SCCT_COMMA;
    tok.start_line = m_src.get_current_line();
    break;
} //END case ','

//TODO: SCCT_QUESTION OK!
case '?': {
    tok.flags = SCCTOK_OP_DEFAULT;

```

```

        tok.tok = SCCT_QUESTION;
        tok.start_line = m_src.get_current_line();
        break;
    } //END case '?'

default:
    /* unrecognized sequence */
    return false;
}
m_src.pos_increment();
return true;
}

```

Приложение Б

Метод exec класса интерпретатора виртуальной машины

```
/**
 * executing byte-code here
 */
SVMI_STATUS SVMI::exec(SVMI_context* p_ctx)
{
    assert(m_pimage_info && "image ptr was nullptr!");
    SVM_instruction instr;
    SVMI_call_context call_ctx;
    uint8_t *p_pcode = m_pimage_info->get_code();
    uint8_t *p_data = m_pimage_info->get_data();
    cell_t *p_stack = p_ctx->get_stack();
    SVMI_VCPU_registers* p_regs = p_ctx->get_regs();
    const SVMI_native_decl* p_imp;
    union {
        struct { float fa, fb, fc; };
        struct { cell_t ia, ib, ic; };
    };
    /* parse bytecode */
    while (1) {
        /* decode instruction
         structure: [opcode][mode][rdst][rsrc][reserved] */
        SVM_instr_fetch(instr, *((int *)&p_pcode[p_regs->IP]));
        /* handle opcodes */
        p_regs->SR_reset(); //reset state register
        p_regs->IP_add(VM_I_SIZE); //skip instruction size, move next to args
        switch (instr.opcode) {
            /* NOP */
            case SVM_OP_NOP:
                break;

            /* MOVE */
            case SVM_OP_MOV:
                /* register to register */
                if (instr.mode == SVMI_ARG_REG) {
                    p_regs->regs[instr.rdst] = p_regs->regs[instr.rsrc];
                    break; /* no args */
                }
                // move value from register to memory
                // s: SVM_OP_MOV SVMI_ARG_REG rdst rsrc
                if (instr.mode == SVMI_ARG_ADDR) {
                    p_data[p_regs->regs[instr.rdst]] = p_regs->regs[instr.rsrc]
& 0xff;
                    break; /* no args */
                }
                // move imm32 value to register
                // s: SVM_OP_MOV SVMI_ARG_IMM rdst rsrc imm32
                if (instr.mode == SVMI_ARG_IMM) {
                    p_regs->regs[instr.rdst] = *((cell_t*)&p_pcode[p_regs->IP]);
                    p_regs->IP_add(SVM_CELL_SIZE); //move next from imm32 instr
arg
                    break; /* 1 arg - 4 bytes */
                }
                /* instr.mode is not correct */
                return SVMI_STATUS_INVALID_INSTRUCTION;

            /* ADD/SUBTRACT */
            // A = A + B
            // A = A - B
            case SVM_OP_ADD:
            case SVM_OP_SUB:
                /* register with register */
```

```

        ia = (instr.opcode == SVM_OP_ADD) ? 1 : -1; /* is ADD? */
        if (instr.mode == SVMI_ARG_REG) {
            /* floating point addition */
            if (instr.flags & VMI_F_FP_OP) {
                fc = SVM_ctof(p_regs->A) + SVM_ctof(p_regs->B) *
(float)ia;

                p_regs->SR_set_flags(fc);
                p_regs->A = SVM_ftoc(fc);
            }
            else {
                /* integer */
                p_regs->A = p_regs->A + p_regs->B * ia;
                p_regs->SR_set_flags(p_regs->A);
            }
            break; /* no args */
        }
        // add imm32 to register
        // s: SVM_OP_ADD|SVM_OP_SUB SVMI_ARG_IMM rdst rsrc imm32
        if (instr.mode == SVMI_ARG_IMM) {
            if (instr.flags & VMI_F_FP_OP) {
                fc = SVM_ctof(p_regs->A) + SVM_ctof(p_pcode[p_regs-
>IP]) * (float)ia;

                p_regs->SR_set_flags(fc);
                p_regs->A = SVM_ftoc(fc);
            }
            else {
                /* integer add/sub operation*/
                p_regs->A = p_regs->A + *((cell_t*)&p_pcode[p_regs-
>IP]) * ia;

                p_regs->SR_set_flags(p_regs->A);
            }
            p_regs->IP_add(SVM_CELL_SIZE); //move next from imm32 instr
arg
            break; /* 1 arg - 4 bytes */
        }
        break;

        /* INCREMENT/DECREMENT */
        // R++;R--
        case SVM_OP_INC:
        case SVM_OP_DEC:
            /* increment only register value */
            ia = (instr.opcode == SVM_OP_INC) ? 1 : -1;
            if (instr.mode == SVMI_ARG_REG) {
                if (instr.flags & VMI_F_FP_OP) {
                    SVM_ctof(p_regs->regs[instr.rdst]) += (float)ia;
                    p_regs->SR_set_flags(SVM_ctof(p_regs-
>regs[instr.rdst]));
                }
                else {
                    p_regs->regs[instr.rdst] += ia;
                    p_regs->SR_set_flags(p_regs->regs[instr.rdst]);
                }
                break;
            }
            break;

        /* MUL/DIV */
        case SVM_OP_MUL:
        case SVM_OP_DIV:
            ia = (int)(instr.opcode == SVM_OP_MUL);
            /* register with register */
            if (instr.mode == SVMI_ARG_REG) {
                /* floating point op */

```

```

        if (instr.flags & VMI_F_FP_OP) {
            if (ia) {
                /*SVM_OP_MUL*/
                fc = SVM_ctof(p_regs->A) * SVM_ctof(p_regs->B);
            }
            else {
                /*SVM_OP_DIV*/
                fb = SVM_ctof(p_regs->B);
                if (fabsf(fb) < FLT_EPSILON) {
                    /* register B was contains zero value */
                    return
SVMI_STATUS_FLOATING_POINT_DIVISION_BY_ZERO;
                }
                fc = SVM_ctof(p_regs->A) / fb;
            }
            p_regs->SR_set_flags(fc);
            p_regs->A = SVM_ftoc(fc);
        }
        else {
            /* integer */
            if (ia) {
                /*SVM_OP_MUL*/
                p_regs->A = p_regs->A * p_regs->B;
                p_regs->SR_set_flags(p_regs->A);
            }
            else {
                /*SVM_OP_DIV*/
                if (!p_regs->B) {
                    /* register B was contains zero value */
                    return
SVMI_STATUS_FLOATING_POINT_DIVISION_BY_ZERO;
                }
                p_regs->A = p_regs->A / p_regs->B;
                p_regs->SR_set_flags(p_regs->A);
            }
        }
        break; /* no args */
    }
    // mul/div register by imm32 value
    // s: SVM_OP_MOV SVMI_ARG_IMM rdst rsrc imm32
    if (instr.mode == SVMI_ARG_IMM) {
        /* is floatign point? */
        if (instr.flags & VMI_F_FP_OP) {
            if (ia) {
                /*SVM_OP_MUL*/
                fc = SVM_ctof(p_regs->A) *
SVM_ctof(p_pcode[p_regs->IP]);
            }
            else {
                /*SVM_OP_DIV*/
                fb = SVM_ctof(p_pcode[p_regs->IP]);
                if (fabsf(fb) < FLT_EPSILON) {
                    /* prevent division by zero */
                    return
SVMI_STATUS_FLOATING_POINT_DIVISION_BY_ZERO;
                }
                fc = SVM_ctof(p_regs->A) / fb;
            }
            p_regs->SR_set_flags(fc);
            p_regs->A = SVM_ftoc(fc);
        }
        else {
            /* integer */
            if (ia) {

```

```

        /*SVM_OP_MUL*/
        p_regs->A = p_regs->A *
*((cell_t*)&p_pcode[p_regs->IP]);
    }
    else {
        /*SVM_OP_DIV*/
        ib = *((cell_t*)&p_pcode[p_regs->IP]);
        if (!ib) {
            /* imm32 is zero for division */
            return SVMI_STATUS_INT_DIVISION_BY_ZERO;
        }
        p_regs->A = p_regs->A / ib;
    }
    p_regs->SR_set_flags(p_regs->A); //update SR
}
p_regs->IP_add(SVM_CELL_SIZE); //move next from imm32 instr
arg
break; /* 1 arg - 4 bytes */
}
break;

case SVM_OP_MOD:
    /* register with register */
    if (instr.mode == SVMI_ARG_REG) {
        if (!p_regs->B) {
            /* imm32 is zero for division */
            return SVMI_STATUS_INT_DIVISION_BY_ZERO;
        }
        p_regs->A = p_regs->A % p_regs->B;
        p_regs->SR_set_flags(p_regs->A);
        break; /* no args */
    }
    // imm32
    // s: SVM_OP_MOD SVMI_ARG_IMM rdst rsrc imm32
    if (instr.mode == SVMI_ARG_IMM) {
        ia = *((cell_t*)&p_pcode[p_regs->IP]);
        if (!ia) {
            /* imm32 is zero for division */
            return SVMI_STATUS_INT_DIVISION_BY_ZERO;
        }
        p_regs->A = p_regs->A % ia;
        p_regs->SR_set_flags(p_regs->A);
        p_regs->IP_add(SVM_CELL_SIZE); //move next from imm32 instr
arg
break; /* 1 arg - 4 bytes */
    }
    break;

    /* SHIFT LEFT */
case SVM_OP_SHL:
    //if (instr.mode == SVMI_ARG_REG) {
        p_regs->regs[instr.rdst] = p_regs->regs[instr.rsrc] <<
p_regs->regs[instr.rdst];
        p_regs->SR_set_flags(p_regs->regs[instr.rdst]);
        break; /* no args */
    //}
    break;

case SVM_OP_SHR:
    //if (instr.mode == SVMI_ARG_REG) {
        p_regs->regs[instr.rdst] = p_regs->regs[instr.rsrc] >> p_regs-
>regs[instr.rdst];
        p_regs->SR_set_flags(p_regs->regs[instr.rdst]);
        break; /* no args */
    //}

```



```

        //}
        break;

case SVM_OP_AND:
    /* register with register */
    if (instr.mode == SVMI_ARG_REG) {
        p_regs->regs[instr.rdst] &= p_regs->regs[instr.rsrc];
        p_regs->SR_set_flags(p_regs->regs[instr.rdst]);
        break; /* no args */
    }
    /* register with imm32 */
    if (instr.mode == SVMI_ARG_IMM) {
        p_regs->regs[instr.rdst] &= *((cell_t*)&p_pcode[p_regs-
>IP]);

        p_regs->SR_set_flags(p_regs->regs[instr.rdst]);
        p_regs->IP_add(SVM_CELL_SIZE); //move next from imm32 instr
arg
        break; /* 1 arg - 4 bytes */
    }
    break;

case SVM_OP_OR:
    /* register with register */
    if (instr.mode == SVMI_ARG_REG) {
        p_regs->regs[instr.rdst] |= p_regs->regs[instr.rsrc];
        p_regs->SR_set_flags(p_regs->regs[instr.rdst]);
        break; /* no args */
    }
    /* register with imm32 */
    if (instr.mode == SVMI_ARG_IMM) {
        p_regs->regs[instr.rdst] |= *((cell_t*)&p_pcode[p_regs-
>IP]);

        p_regs->SR_set_flags(p_regs->regs[instr.rdst]);
        p_regs->IP_add(SVM_CELL_SIZE); //move next from imm32 instr
arg
        break; /* 1 arg - 4 bytes */
    }
    break;

case SVM_OP_XOR:
    /* register with register */
    if (instr.mode == SVMI_ARG_REG) {
        p_regs->regs[instr.rdst] ^= p_regs->regs[instr.rsrc];
        p_regs->SR_set_flags(p_regs->regs[instr.rdst]);
        break; /* no args */
    }
    /* register with imm32 */
    if (instr.mode == SVMI_ARG_IMM) {
        p_regs->regs[instr.rdst] ^= *((cell_t*)&p_pcode[p_regs-
>IP]);

        p_regs->SR_set_flags(p_regs->regs[instr.rdst]);
        p_regs->IP_add(SVM_CELL_SIZE); //move next from imm32 instr
arg
        break; /* 1 arg - 4 bytes */
    }
    break;

case SVM_OP_NOT:
    p_regs->regs[instr.rdst] = (cell_t) (!p_regs->regs[instr.rdst]);
    break;

case SVM_OP_CMP:
    /* register with register */
    if (instr.mode == SVMI_ARG_REG) {

```

```

        /* is floatign pointnt? */
        if (instr.flags & VMI_F_FP_OP) {
            fb = SVM_ctof(p_regs->regs[instr.rsrc]) -
SVM_ctof(p_regs->regs[instr.rdst]);
            p_regs->SR_set_flags(fb);
        } else {
            ib = p_regs->regs[instr.rsrc] - p_regs-
>regs[instr.rdst];
            p_regs->SR_set_flags(ib);
        }
        break; /* no args */
    }
    /* register with imm32 */
    if (instr.mode == SVMI_ARG_IMM) {
        /* is floatign pointnt? */
        if (instr.flags & VMI_F_FP_OP) {
            fc = SVM_ctof(p_regs->regs[instr.rsrc]) -
SVM_ctof(p_pcode[p_regs->IP]);
            p_regs->SR_set_flags(fc);
        }
        else {
            ib = p_regs->regs[instr.rsrc] - p_pcode[p_regs->IP];
            p_regs->SR_set_flags(ib);
        }
        p_regs->IP_add(SVM_CELL_SIZE); //move next from imm32 instr
arg
        break; /* 1 arg - 4 bytes */
    }
    break;

case SVM_OP_JMP:
    /* imm32 */
    if (instr.mode == SVMI_ARG_IMM) {
        p_regs->IP += *((cell_t*)&p_pcode[p_regs->IP]);
        p_regs->IP_add(SVM_CELL_SIZE);
        break;
    }
    /* with register */
    if (instr.mode == SVMI_ARG_REG) {
        p_regs->IP += p_regs->regs[instr.rsrc];
        break;
    }
    break;

case SVM_OP_JE:
case SVM_OP_JZ:
    /* imm32 */
    if (instr.mode == SVMI_ARG_IMM) {
        if (p_regs->SR_is_set(VMSRF_ZF)) {
            p_regs->IP += *((cell_t*)&p_pcode[p_regs->IP]);
        }
        p_regs->IP_add(SVM_CELL_SIZE);
        break;
    }
    /* with register */
    if (p_regs->SR_is_set(VMSRF_ZF)) {
        if (instr.mode == SVMI_ARG_REG) {
            p_regs->IP += p_regs->regs[instr.rsrc];
        }
        break;
    }
    break;

```

```

case SVM_OP_JNE:
case SVM_OP_JNZ:
    /* imm32 */
    if (instr.mode == SVMI_ARG_IMM) {
        if (!p_regs->SR_is_set(VMSRF_ZF)) {
            p_regs->IP += *((cell_t*)&p_pcode[p_regs->IP]);
        }
        p_regs->IP_add(SVM_CELL_SIZE);
        break;
    }
    /* with register */
    if (!p_regs->SR_is_set(VMSRF_ZF)) {
        if (instr.mode == SVMI_ARG_REG) {
            p_regs->IP += p_regs->regs[instr.rsrc];
        }
        break;
    }
    break;

case SVM_OP_JL:
case SVM_OP_JLE:
    ia = VMSRF_SF;
    if (instr.opcode == SVM_OP_JLE)
        ia |= VMSRF_ZF;

    /* imm32 */
    if (instr.mode == SVMI_ARG_IMM) {
        if (!p_regs->SR_is_set(ia)) {
            p_regs->IP += *((cell_t*)&p_pcode[p_regs->IP]);
        }
        p_regs->IP_add(SVM_CELL_SIZE);
        break;
    }

    /* with register */
    if (!p_regs->SR_is_set(ia)) {
        if (instr.mode == SVMI_ARG_REG) {
            p_regs->IP += p_regs->regs[instr.rsrc];
        }
        break;
    }
    break;

case SVM_OP_JG:
case SVM_OP_JGE:
    /* imm32 */
    if (instr.mode == SVMI_ARG_IMM) {
        /* zero flag is set? */
        if (p_regs->SR_is_set(ia))
            p_regs->IP += *((cell_t*)&p_pcode[p_regs->IP]);

        p_regs->IP_add(SVM_CELL_SIZE);
        break;
    }

    /* with register */
    ia = 0; // remove SF
    if (instr.opcode == SVM_OP_JLE)
        ia |= VMSRF_ZF;

    if (p_regs->SR_is_set(ia)) {
        /* src is register? */
        if (instr.mode == SVMI_ARG_REG) {
            p_regs->IP += p_regs->regs[instr.rsrc];

```

```

        break;
    }
}
break;

case SVM_OP_LOOP:
    /* counter register */
    if (p_regs->C > 0) {
        p_regs->IP += (instr.mode == SVMI_ARG_IMM) ?
*((cell_t*)&p_pcode[p_regs->IP]) : p_regs->regs[instr.rsrc];
        p_regs->C--;
    }
    /* zero flag */
    p_regs->SR_is_set(VMSRF_ZF);
    p_regs->IP_add(SVM_CELL_SIZE);
    break;

case SVM_OP_CASE:
    break;

case SVM_OP_PUSH:
    /* detect stack overflow */
    if (p_regs->SP >= p_ctx->get_stack_size())
        return SVMI_STATUS_STACK_OVERFLOW;

    if (instr.mode == SVMI_ARG_IMM) {
        p_stack[p_regs->SP++] = *((cell_t*)&p_pcode[p_regs->IP]);
        p_regs->IP_add(SVM_CELL_SIZE);
        break;
    }
    p_stack[p_regs->SP++] = p_regs->regs[instr.rsrc];
    break;

case SVM_OP_PUSHSR:
    if (p_regs->SP >= p_ctx->get_stack_size())
        return SVMI_STATUS_STACK_OVERFLOW;

    p_stack[p_regs->SP++] = p_regs->SR;
    break; /* no args */

case SVM_OP_POP:
    /* detect stack overflow */
    if (!p_regs->SP)
        return SVMI_STATUS_STACK_OVERFLOW;

    if (instr.mode == SVMI_ARG_REG)
        p_regs->regs[instr.rdst] = p_stack[p_regs->SP];

    p_regs->SP--;
    break; /* no args */

case SVM_OP_POPSR:
    /* detect stack overflow */
    if (!p_regs->SP)
        return SVMI_STATUS_STACK_OVERFLOW;

    p_regs->SR = p_stack[p_regs->SP++];
    break; /* no args */

    /* call proc/import */
case SVM_OP_CALL:
case SVM_OP_NCALL:
    /* read arg */
    ia = *((cell_t*)&p_pcode[p_regs->IP]);

```

```

        /* is SVM_OP_CALL? */
        if (instr.opcode == SVM_OP_CALL) {
            call_ctx.previous_SP = p_ctx->SP;
            call_ctx.return_address = p_ctx->IP + SVM_CELL_SIZE; //
[instruction (4bytes)][arg (SVM_CELL_SIZE)] (next code...)
            if (!p_ctx->push_call_context(call_ctx)) {
                /* call context stack overflowed */
                return SVMI_STATUS_CALL_CONTEXT_STACK_OVERFLOW;
            }
            /* change IP value to address */
            p_ctx->IP = ia;
            break;
        }
        /* is SVM_OP_NCALL? */
        /* is valid native func index? */
        if (ia >= (cell_t)m_vnatives_idx.size())
            return SVMI_STATUS_IMPORT_INDEX_OUT_OF_BOUNDS;

        /* get needed import native */
        p_imp = &m_pnatives[m_vnatives_idx[ia]];
        assert(p_imp->p_nativefunc && "p_imp->p_nativefunc was nullptr!");
        p_regs->A = p_imp->p_nativefunc(p_ctx);
        p_regs->IP_add(SVM_CELL_SIZE); //skip instr 4-bytes arg
        break; /* 1 arg */

    case SVM_OP_RET:
        if (!p_ctx->pop_call_context(call_ctx)) {
            /* call context stack overflowed */
            return SVMI_STATUS_CALL_CONTEXT_STACK_OVERFLOW;
        }
        /* restore registers */
        p_ctx->SP = call_ctx.previous_SP;
        p_ctx->IP = call_ctx.return_address;
        break; /* no args */

        /* TRIGGERED BRACKPOINT */
    case SVM_REG_BRK:
        if (m_pdbg_proc) {
            if (m_pdbg_proc(p_ctx) == SVMI_DBG_PROC_STATUS_STOP) {
                return SVMI_STATUS_EXECUTION_HALTED; //finish p-code
execution
            }
        }
        break; /* no args */

        /* HALT */
    case SVM_REG_HALT:
        return SVMI_STATUS_EXECUTION_HALTED; //finish p-code execution

    case SVM_REG_CEIL:
        break;

    case SVM_REG_FLOOR:
        break;
    }
    return SVMI_STATUS_INVALID_INSTRUCTION;
}

```

Приложение В

Класс генератора кода

```
class scc_code_emitter
{
    //!!! here class for collecting generated byte-code
    std::vector<uint8_t> m_code;

    template<class _type>
    void write(_type &value) {
        m_code.resize(m_code.size() + sizeof(_type));
        *((_type*)&m_code[m_code.size()]) = value;
    }

    /**
     * write_instruction
     * opcode
     * mode
     * rdst
     * rsrc
     * flags
     */
    void write_instruction(SVM_OP opcode, int mode, int rdst, int rsrc, int
flags) {
        /* for 0 arg instructions */
        int instr_int;
        SVM_instruction instruction;
        instruction.opcode = opcode;
        instruction.mode = mode;
        instruction.rsrc = rsrc;
        instruction.rdst = rdst;
        instruction.flags = flags;
        SVM_instr_write(instr_int, instruction);
        write<int>(instr_int);
    }

    /**
     * write_instruction with cell_t arg
     * opcode
     * mode
     * rdst
     * rsrc
     * flags
     * arg
     */
    void write_instruction(SVM_OP opcode, int mode, int rdst, int rsrc, int
flags, cell_t arg) {
        write_instruction(opcode, mode, rdst, rsrc, flags);
        write<cell_t>(arg);
        /* for 1 arg instructions */
    }

public:
    scc_code_emitter() {}
    ~scc_code_emitter() {}

    /* nop */
    inline void nop() {
        write_instruction(SVM_OP_NOP, 0, 0, 0, 0);
    }

    /* mov */
    inline void mov(SVM_REGS dst, cell_t value) {
        write_instruction(SVM_OP_MOV, SVMI_ARG_IMM, dst, 0, 0, value);
    }
}
```

```

}
inline void mov(SVM_REGS dst, SVM_REGS src) {
    write_instruction(SVM_OP_MOV, SVMI_ARG_REG, dst, src, 0);
}
inline void mov(SVM_REGS dst, SVM_REGS src, bool) {
    write_instruction(SVM_OP_MOV, SVMI_ARG_ADDR, dst, src, 0);
}

/* add */
/**
 * flags = 0 (default). For use float OP, set flag VMI_F_FP_OP
 */
inline void add(SVM_REGS dst, SVM_REGS src, int flags = 0) {
    write_instruction(SVM_OP_ADD, SVMI_ARG_REG, dst, src, flags);
}
inline void fadd(SVM_REGS dst, SVM_REGS src) {
    add(dst, src, VMI_F_FP_OP);
}
inline void add(SVM_REGS dst, cell_t value) {
    write_instruction(SVM_OP_ADD, SVMI_ARG_REG, dst, 0, 0, value);
}
inline void fadd(SVM_REGS dst, float value) {
    write_instruction(SVM_OP_ADD, SVMI_ARG_REG, dst, 0, VMI_F_FP_OP,
SVM_ftoc(value));
}

/* sub */
/**
 * flags = 0 (default). For use float OP, set flag VMI_F_FP_OP
 */
inline void sub(SVM_REGS dst, SVM_REGS src, int flags = 0) {
    write_instruction(SVM_OP_SUB, SVMI_ARG_REG, dst, src, flags);
}
inline void fsub(SVM_REGS dst, SVM_REGS src) {
    add(dst, src, VMI_F_FP_OP);
}
inline void sub(SVM_REGS dst, cell_t value) {
    write_instruction(SVM_OP_SUB, SVMI_ARG_IMM, dst, 0, 0, value);
}
inline void fsub(SVM_REGS dst, float value) {
    write_instruction(SVM_OP_SUB, SVMI_ARG_IMM, dst, 0, VMI_F_FP_OP,
SVM_ftoc(value));
}

/**
 * flags = 0 (default). For use float OP, set flag VMI_F_FP_OP
 */
inline void inc(SVM_REGS dst, int flags = 0) {
    write_instruction(SVM_OP_INC, SVMI_ARG_REG, dst, 0, flags);
}
inline void finc(SVM_REGS dst) {
    inc(dst, VMI_F_FP_OP);
}

/**
 * flags = 0 (default). For use float OP, set flag VMI_F_FP_OP
 */
inline void dec(SVM_REGS dst, int flags = 0) {
    write_instruction(SVM_OP_DEC, SVMI_ARG_REG, dst, 0, flags);
}
inline void fdec(SVM_REGS dst) {
    dec(dst, VMI_F_FP_OP);
}

/**

```

```

* flags = 0 (default). For use float OP, set flag VMI_F_FP_OP
*/
inline void mul(SVM_REGS dst, SVM_REGS src, int flags = 0) {
    write_instruction(SVM_OP_MUL, SVMI_ARG_REG, dst, src, flags);
}
inline void mul(SVM_REGS dst, cell_t value) {
    write_instruction(SVM_OP_MUL, SVMI_ARG_IMM, dst, 0, 0, value);
}
inline void fmul(SVM_REGS dst, float value) {
    write_instruction(SVM_OP_MUL, SVMI_ARG_IMM, dst, 0, VMI_F_FP_OP,
SVM_ftoc(value));
}
inline void fmul(SVM_REGS dst, SVM_REGS src) {
    mul(dst, src, VMI_F_FP_OP);
}

inline void div(SVM_REGS dst, SVM_REGS src, int flags = 0) {
    write_instruction(SVM_OP_DIV, SVMI_ARG_REG, dst, src, flags);
}
inline void fdiv(SVM_REGS dst, SVM_REGS src) {
    div(dst, src, VMI_F_FP_OP);
}
inline void div(SVM_REGS dst, cell_t value) {
    write_instruction(SVM_OP_DIV, SVMI_ARG_IMM, dst, 0, 0, value);
}
inline void fdiv(SVM_REGS dst, float value) {
    write_instruction(SVM_OP_DIV, SVMI_ARG_IMM, dst, 0, VMI_F_FP_OP,
SVM_ftoc(value));
}

inline void mod(SVM_REGS dst, SVM_REGS src, int flags = 0) {
    write_instruction(SVM_OP_MOD, SVMI_ARG_REG, dst, src, flags);
}
inline void fmod(SVM_REGS dst, SVM_REGS src) {
    mod(dst, src, VMI_F_FP_OP);
}
inline void mod(SVM_REGS dst, cell_t value) {
    write_instruction(SVM_OP_MOD, SVMI_ARG_IMM, dst, 0, 0, value);
}
inline void fmod(SVM_REGS dst, float value) {
    write_instruction(SVM_OP_MOD, SVMI_ARG_IMM, dst, 0, VMI_F_FP_OP,
SVM_ftoc(value));
}
//...
};

```


Приложение Г

Определения кодов операций, флагов, режимов адресации и функций записи и чтения инструкций.

```
/**
 * simple virtual machine
 */
#pragma once
#include <stdint.h>
#include "bitop.h"

/* processor state flags */
#define VMSRF_ZF (1 << 0) /*< zero flag */
#define VMSRF_SF (1 << 1) /*< sign flag */
#define VMSRF_CF (1 << 2) /*< carry flag */

/* cell registers and stack align type */
typedef int cell_t;
typedef unsigned int ucell_t;

#define SVM_CELL_SIZE sizeof(cell_t)

/* float to cell */
#define SVM_ftoc(x) (*((cell_t *)&x))
#define SVM_ctof(x) (*((float *)&x))

/* SVM registers */
enum SVM_REGS : uint8_t {
    /* general purpose registers */
    SVM_REG_A, SVM_REG_B, SVM_REG_C, SVM_REG_D,
    SVM_REG_X, SVM_REG_Y, SVM_REG_Z, SVM_REG_W,
    SVM_REG_SR,

    /* IP */
    SVM_REG_IP,
    SVM_REG_SP,
    SVM_REG_CS,
    SVM_REG_DS,
    SVM_REG_SS
};

/* instruction flags */
enum SVMI_ARG_TYPE {
    SVMI_ARG_REG = 0,
    SVMI_ARG_ADDR,
    SVMI_ARG_IMM
};

/**
 * instruction structure
 *
 * 16 bit [0 -15] - operation code
 * 2 bit [16-17] - mode (SVMI_ARG_REG|SVMI_ARG_ADDR|SVMI_ARG_IMM)
 * 4 bit [18-21] - dest register
 * 4 bit [22-25] - source register
 * 6 bit [26-31] - flags
 */

#define VM_I_SIZE (sizeof(int)) // VM each instruction size
#define VM_I_OPC_BITS (16)
#define VM_I_MODE_BITS (2)
#define VM_I_REG_DIR_BITS (4)
```

```

enum SVM_OP : uint32_t
{
    SVM_OP_NOP = 0,
    SVM_OP_MOV,
    SVM_OP_ADD,
    SVM_OP_INC,
    SVM_OP_SUB,
    SVM_OP_DEC,
    SVM_OP_MUL,
    SVM_OP_DIV,
    SVM_OP_MOD,    //TODO: K.D. FMOD NOT IMPLEMENTED IN VM!

    SVM_OP_SHL,
    SVM_OP_SHR,
    SVM_OP_AND,
    SVM_OP_OR,
    SVM_OP_XOR,

    SVM_OP_NOT,
    SVM_OP_CMP,

    SVM_OP_JMP,
    SVM_OP_JZ,
    SVM_OP_JNZ,
    SVM_OP_JE,
    SVM_OP_JNE,
    SVM_OP_JL,
    SVM_OP_JLE,
    SVM_OP_JG,
    SVM_OP_JGE,

    SVM_OP_LOOP,

    SVM_OP_CASE,

    SVM_OP_PUSH,
    SVM_OP_PUSHSR,
    SVM_OP_POP,
    SVM_OP_POPSR,

    SVM_OP_CALL,
    SVM_OP_NCALL,
    SVM_OP_RET,

    SVM_REG_BRK,
    SVM_REG_HALT,

    SVM_REG_STOI,
    SVM_REG_ITOS,
    SVM_REG_FTOI,
    SVM_REG_ITOF,
    SVM_REG_ROUND,
    SVM_REG_CEIL,
    SVM_REG_FLOOR
};

struct SVM_instruction {
    SVM_OP opcode;
    int     mode;
    int     rsrc;
    int     rdst;
    int     flags;
};

```

```

/* instruction flags */
#define VMI_F_FP_OP (1 << 0) //is floating point ariphmetical operation?

inline void SVM_instr_write(int& insruction, const SVM_instruction& src)
{
    insruction = 0;
    WRITE_BITS(insruction, src.opcode, 0, 15);
    WRITE_BITS(insruction, src.mode, 16, 17);
    WRITE_BITS(insruction, src.rsrc, 18, 21);
    WRITE_BITS(insruction, src.rdst, 22, 25);
    WRITE_BITS(insruction, src.flags, 26, 31);
}

inline void SVM_instr_fetch(SVM_instruction& dst, int instruction)
{
    dst.opcode = (SVM_OP)READ_BITS(instruction, 0, 15);
    dst.mode = READ_BITS(instruction, 16, 17);
    dst.rsrc = READ_BITS(instruction, 18, 21);
    dst.rdst = READ_BITS(instruction, 22, 25);
    dst.flags = READ_BITS(instruction, 26, 31);
}

```