**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](#_Toc182350787)

[**ОСНОВНАЯ ЧАСТЬ** 6](#_Toc182350788)

[**РАЗРАБОТКА ЯЗЫКА ПРОГРАММИРОВАНИЯ** 7](#_Toc182350789)

[**ТРАНСЛЯЦИЯ ПРОГРАММЫ** 8](#_Toc182350790)

[**РАЗРАБОТКА ЛЕКСИЧЕКОГО АНАЛИЗАТОРА** 9](#_Toc182350791)

[**ГРАММАТИКА СИНТАКСИЧЕСКОГО РАЗБОРА** 11](#_Toc182350792)

[**РАЗРАБОТКА СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА** 12](#_Toc182350793)

[**ГЕНЕРАЦИЯ КОДА И АРХИТЕКТУРА ВИРТУАЛЬНОЙ МАШИНЫ** 13](#_Toc182350794)

[**ЗАКЛЮЧЕНИЕ** 18](#_Toc182350795)

[**Список использованной литературы** 19](#_Toc182350796)

[**Приложение А** 20](#_Toc182350797)

[**Приложение Б** 29](#_Toc182350798)

[**Приложение В** 38](#_Toc182350799)

[**Приложение Г** 41](#_Toc182350800)

# **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 лескическому анализатору (lexer).

После того как лексер получил массив символов и его размер, можно приступать 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 строковой 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 зависимы 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.

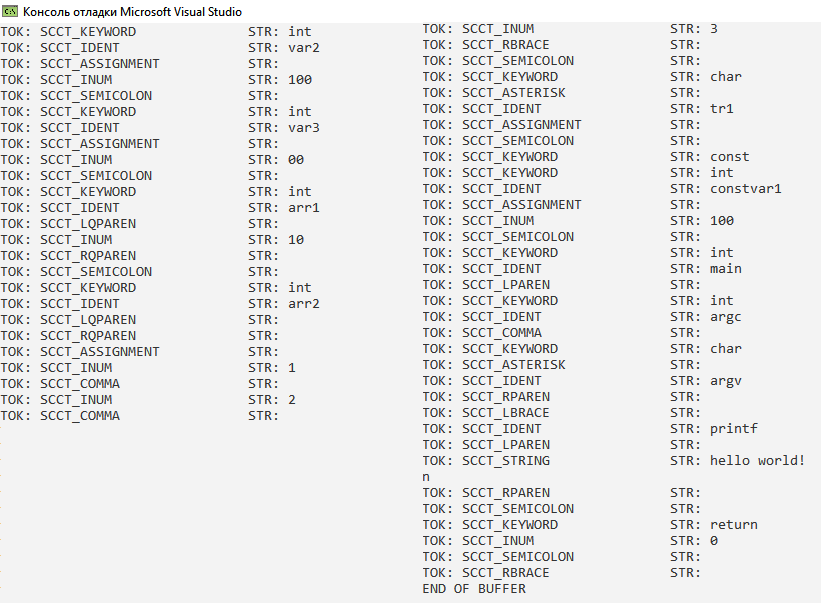


Рисунок 1 – Тестирование лексического анализатора

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 delimeters \*/

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 c плавающей запятой Результат сохраняется в регистре A |
| DEC R | Декремент целого числа в регистре |
| FDEC R | Декремент числа c плавающей запятой |
| MUL A, B MUL A, imm32 | Умножение целых чисел Результат сохраняется в регистре A |
| FMUL A, B  FMUL A, imm32 | Умножение чисел c 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](http://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\_RBRACE;

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 correc t \*/

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 poinnt? \*/

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 poinnt? \*/

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 poinnt? \*/

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) {

/\* sero 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\_idxs.size())

return SVMI\_STATUS\_IMPORT\_INDEX\_OUT\_OF\_BOUNDS;

/\* get needed import native \*/

p\_imp = &m\_pnatives[m\_vnatives\_idxs[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);

}