

Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 21.Б07-мм

# Интринсики RISC-V для OCaml

*Габдрахманов Азат Райнурович*

Отчёт по учебной практике  
в форме «Решение»

Научный руководитель:  
ассистент кафедры системного программирования, Косарев Д. С.

Санкт-Петербург  
2024

# Оглавление

|                                        |           |
|----------------------------------------|-----------|
| <b>Введение</b>                        | <b>3</b>  |
| <b>1. Постановка задачи</b>            | <b>5</b>  |
| <b>2. Обзор доступных инструментов</b> | <b>6</b>  |
| 2.1. RISC-V . . . . .                  | 6         |
| 2.2. OCaml . . . . .                   | 7         |
| <b>3. Существующие решения</b>         | <b>11</b> |
| 3.1. Интринсик ARM «sqrt» . . . . .    | 11        |
| 3.2. Интринсики RVV для C . . . . .    | 12        |
| <b>4. Реализация</b>                   | <b>14</b> |
| 4.1. Функция vector_add . . . . .      | 14        |
| 4.2. Функция caml_find . . . . .       | 14        |
| <b>5. Эксперимент</b>                  | <b>16</b> |
| 5.1. Сложение . . . . .                | 16        |
| 5.2. Find . . . . .                    | 16        |
| <b>Заключение</b>                      | <b>17</b> |
| <b>Список литературы</b>               | <b>18</b> |

# Введение

В настоящее время преобладают высокоуровневые языки программирования, ведь используя различные высокоуровневые конструкции программист получает возможность сэкономить время написания программы при сравнимом качестве кода, что и в низкоуровневых языках. При этом зачастую реализация высокоуровневых языков программирования препятствует свободному использованию низкоуровневых инструкций или оптимизаций. Также случается, что компилятор высокоуровневого языка не имеет понятия о всех возможностях процессора.

Для внедрения в компилятор новых возможностей в него зачастую добавляют интринсики. Примером может служить RVV<sup>1</sup> для языка C, с помощью которого можно ускорять работу с массивами данных, используя расширение RISC-V V.

В языке OCaml компания Jane Street занимается расширением библиотеки интринсиков для архитектур ARM и AMD. В том числе добавляя векторные интринсики на основе NEON. Изучив их проект, появилась идея сделать аналогичные интринсики для компилятора OCaml, но для архитектуры RISC-V. Задуман поход, при котором программист будет вызывать из OCaml внешние (англ. external) функции с соответствующими именами. Когда компилятор будет встречать внешние вызовы, которые он знает, то будут вставляться соответствующие инструкции RISC-V. В противном случае компилятор будет полагаться, что реализация этих функций возьмется из других объектных файлов в процессе линковки.

Существует два компилятора OCaml: `ocamlc` — компилятор байт-кода и `ocamlpt` — компилятор нативного кода. `OCamlc` компилирует быстрее, но время исполнения дольше. Чаще всего используется компилятор `ocamlpt`, потому что после компиляции программа, зачастую, используется многократно.

В данном проекте планируется добавить интринсики в `OCamlpt`. На основе промежуточного представления `Cmm` [2] будет порождаться

---

<sup>1</sup><https://github.com/riscv/riscv-v-spec>

ассемблерный код с низкоуровневыми оптимизациями.

# 1. Постановка задачи

Целью работы является реализация интринсиков для компилятора OCaml под архитектуру RISC-V.

Для ее выполнения были поставлены следующие задачи:

1. Реализовать векторные интринсики.
2. Сравнить производительность инструкций.

## 2. Обзор доступных инструментов

### 2.1. RISC-V

В настоящее время RISC-V становится все более популярной. Это открытая и свободная архитектура процессора и система команд, основанные на принципах RISC<sup>2</sup>. Концепция RISC (Reduced Instruction Set Computer — компьютер с сокращенным набором команд) основывается на идее использования более простого и менее сложного кода для инструкций по сравнению с командами CISC (Complex Instruction Set Computer — компьютер со сложным набором команд). Важно уточнить, что термин «сокращенный» в данном контексте относится к сложности инструкции, а не к их количеству.

Рассмотрим причины, по которым RISC заменяет CISC, а также преимущества и недостатки каждого из подходов. В процессорах CISC присутствует много сложных инструкций, которые поддерживают высокоуровневые конструкции. Это было удобно в прошлом по следующим причинам:

- увеличение производительности при работе на ассемблере;
- с учетом медленной и дорогой памяти, использование сложных инструкций, которые выполняют много действий и минимально используют память, было предпочтительнее;
- компактность кода.

Однако у этого подхода есть и недостатки:

- меньше регистров общего назначения, так как для декодирования команд требуется больше транзисторов;
- необходимость нескольких тактов для выполнения одной инструкции, несмотря на минимальный размер кода;

---

<sup>2</sup><https://ru.wikipedia.org/wiki/RISC-V>

- более высокая стоимость реализации (требуется больше транзисторов).

В контексте RISC, инструкции проще, из-за чего их может потребоваться больше. Однако цель заключается в том, чтобы компенсировать необходимость обработки большего числа команд за счет увеличения скорости выполнения каждой команды. Это можно достичь, в частности, с помощью реализации командного конвейера, который проще реализовать при использовании более простых команд<sup>3</sup>.

### 2.1.1. Набор инструкций RISC-V

Набор инструкций RISC-V имеет модульную структуру и может быть расширен [3]. Базовый набор инструкций достаточно компактен: всего 11 основных арифметических инструкций (большинство из них могут быть представлены в двух формах, образуя в сумме 21 инструкцию), 10 инструкций для работы с памятью и 8 инструкций для переходов. Всего – 39 инструкций. Одно из значительных преимуществ – наличие 32 регистров.

### 2.1.2. Расширения RISC-V

Существует множество расширений RISC-V. Расширения могут представлять собой относительно сложные операции по сравнению с базовым набором, например, инструкции для работы с плавающей точкой, атомарные операции, векторные инструкции, инструкции для обработки строк и другие. Это позволяет увеличить производительность обработки определенных задач, избегая при этом ненужной сложности для других приложений.

## 2.2. OCaml

OCaml — это высокоуровневый функциональный язык программирования, разработанный INRIA (Французский национальный институт

---

<sup>3</sup>[https://en.wikipedia.org/wiki/Reduced\\_instruction\\_set\\_computer](https://en.wikipedia.org/wiki/Reduced_instruction_set_computer)

исследований в области информатики и автоматики). Он сочетает в себе функциональное, императивное и объектно-ориентированное программирование, что делает его мощным и гибким инструментом для разработки программного обеспечения.

### **2.2.1. Особенности OCaml**

Основные особенности OCaml включают:

- **Статическая типизация:** OCaml автоматически проверяет типы во время компиляции, что помогает избегать ошибок типов во время выполнения.
- **Автоматическое управление памятью:** OCaml автоматически выделяет и освобождает память, что уменьшает риск утечек памяти и ошибок доступа к памяти.
- **Поддержка функционального программирования:** OCaml поддерживает функциональное программирование, что позволяет разработчикам писать более краткий и понятный код.
- **Поддержка объектно-ориентированного программирования:** OCaml поддерживает объектно-ориентированное программирование, что позволяет разработчикам создавать сложные структуры данных и абстрагировать поведение данных.
- **Высокая производительность:** OCaml компилируется в нативный код, что обеспечивает высокую производительность и эффективное использование ресурсов.

### **2.2.2. Инфраструктура OCaml**

Среда разработки OCaml включает компилятор OCaml, который может компилировать код OCaml в байт-код для выполнения на виртуальной машине OCaml, или в нативный код для выполнения на конкретной архитектуре процессора. Также доступны различные инструменты для

сборки, тестирования и отладки программ на OCaml, включая систему сборки Dune и интегрированную среду разработки (IDE) OCamlPro.

### 2.2.3. Применение OCaml

OCaml широко используется в академическом и промышленном мире для разработки различных видов программного обеспечения, включая системное программное обеспечение, сетевые сервисы, базы данных, веб-приложения, искусственный интеллект и многое другое. Компании, такие как Facebook, Jane Street и Docker<sup>4</sup>, используют OCaml в своих продуктах и инфраструктуре.

#### 2.2.4. Генерация переносимого байт-кода

ОСaml использует двухэтапный процесс для генерации исполняемых файлов. На первом этапе компилятор ОСaml (`ocamlc`) компилирует исходный код в байт-код. Байт-код представляет собой промежуточное представление кода, которое затем может быть выполнено на виртуальной машине ОСaml (`ocamlrun`).

Этот подход имеет несколько преимуществ. Во-первых, байт-код является переносимым, что позволяет запускать программы на ОСaml на различных архитектурах процессора и операционных системах без необходимости повторной компиляции. Во-вторых, генерация байт-кода обычно быстрее, чем компиляция в нативный код, что может ускорить процесс разработки.

### 2.2.5. Компиляция в быстрый нативный код

В дополнение к компиляции в байт-код, OSaml также поддерживает компиляцию в нативный код с использованием компилятора `osamlpt`. Это позволяет генерировать эффективный исполняемый код, который может быть напрямую выполнен на процессоре без необходимости виртуальной машины.

<sup>4</sup>[https://dev.to/yawaramin/ocaml-for-business\protect\discretionary{\char\hyphenchar\font}{\f}{\f}{}{1d0l}](https://dev.to/yawaramin/ocaml-for-business\protect\discretionary{\char\hyphenchar\font}{\f}{\f}{}{1d0l)

Компиляция в нативный код обычно приводит к более быстрому исполнению программы, но процесс компиляции медленнее, и полученный исполняемый файл будет специфичным для конкретной архитектуры процессора.

### 2.2.6. Представление целых чисел в OCaml

Целые числа в OCaml представляются в специфическом формате, который оптимизирован для эффективности. Конкретно, целое число  $x$  хранится как  $(x \ll 1) + 1$  [1]. Это означает, что число 5, например, будет храниться как 1011, где последний бит отмечает, что это целое число, а не ссылка.

Этот подход имеет несколько преимуществ:

- Сборщику мусора требуется выполнить меньше работы, поскольку он не должен учитывать целые числа.
- Получение целого числа не требует разыменовывания указателя.
- Создание целого числа не требует выделения памяти.
- Фрагментация памяти уменьшается, так как целые числа не занимают дополнительное пространство в памяти.

Однако этот подход также имеет свои недостатки. В частности, выполнение арифметических операций с целыми числами требует дополнительных шагов:

- Для выполнения операции  $x + y$ , OCaml выполняет операцию  $x + y - 1$  на уровне процессора.
- Для операции  $x * y$ , OCaml вычисляет  $(x \gg 1) \cdot (y - 1) + 1$ .
- Для операции  $x / y$ , OCaml вычисляет  $((x \gg 1) / (y \gg 1)) \ll 1 + 1$ .
- Для операции  $x \text{ lsl } y$ , OCaml вычисляет  $((x - 1) \ll (y \gg 1)) + 1$ .

Эти дополнительные шаги могут увеличивать время выполнения арифметических операций.

### 3. Существующие решения

В данном разделе рассматривается поддержка интринсиков в проекте Flambda2<sup>5</sup>. В этом проекте вызов внешней функции с определённым названием заменяется на конкретные инструкции.

Объявление внешней функции в OCaml выглядит следующим образом:

#### Листинг 1: Объявление внешней функции

```
external popcnt: int -> int = "c_popcnt"
```

Процесс компиляции кода с вызовом внешней функции заключается в следующем: пусть функция находится в файле `lib.c`, а её вызов в `a.ml`. Эти файлы будут скомпилированы в объектные файлы, но в объектном файле OCaml вместо адреса функции `c_popcnt` будут стоять нули, так как она является внешней. После компоновки в объектном файле появится конкретный адрес.

#### 3.1. Интринсик ARM «sqrt»

Замена вызова функции на конкретную инструкцию происходит после анализа промежуточного представления Cmm.

#### Листинг 2: Анализ Cmm

```
| Cextcall { func = "sqrt" } ->
(Ispecific Isqrtf, args)
```

Этот код означает, что вызов внешней функции с названием `sqrt` заменяется на значение алгебраического типа для специфичных операций ARM.

#### Листинг 3: Генерация ассемблерного кода в зависимости от операции

```
| Lop(Ispecific Isqrtf as op) ->
  let instr = (match op with
               | Ispecific Isqrtf -> "fsqrt")
```

---

<sup>5</sup><https://github.com/ocaml-flambda/flambda-backend>

```

| -                                     -> assert false) in
{emit_string instr}      {emit_reg i.res.(0)},
{emit_reg i.arg.(0)}`

```

Последняя строка генерирует конкретную ассемблерную инструкцию.

## 3.2. Интринсики RVV для C

RISC-V Vector Extension (RVV) предоставляет мощные возможности для работы с векторными данными, которые могут быть использованы в различных высокопроизводительных вычислительных задачах. Интринсики RVV позволяют программистам писать код на C, который напрямую обращается к возможностям RVV, обеспечивая высокую производительность и эффективное использование аппаратных ресурсов.

Пример использования интринсиков RVV в C для векторного сложения может выглядеть следующим образом:

### Листинг 4: Пример использования интринсиков RVV в C

```

#include <riscv_vector.h>

void vector_add(int *a, int *b, int *c, size_t n) {
    size_t vl;
    for (size_t i = 0; i < n; i += vl) {
        vl = vsetvl_e32m1(n - i);
        vuint32m1_t va = vle32_v_u32m1(&a[i], vl);
        vuint32m1_t vb = vle32_v_u32m1(&b[i], vl);
        vuint32m1_t vc = vadd_vv_u32m1(va, vb, vl);
        vse32_v_u32m1(&c[i], vc, vl);
    }
}

```

Этот код показывает, как можно использовать интринсики для работы с векторными регистрами, загружая данные из массивов `a` и `b`, выполняя векторное сложение и сохраняя результат в массив `c`.

Использование интринсиков RVV позволяет значительно ускорить выполнение задач, требующих высокой производительности, таких как численные расчёты, обработка сигналов и другие научные и инженерные приложения.

## 4. Реализация

В данном разделе обсуждаются две важные функции, реализованные в проекте: «`vector_add`» и «`caml_find`».

### 4.1. Функция `vector_add`

Функция *vector\_add* предназначена для выполнения векторного сложения. Она принимает два вектора в качестве входных данных, выполняет сложение их элементов, а затем сохраняет результат в выделенном месте памяти.

Процесс реализации этой функции включает в себя несколько этапов:

1. Вначале сохраняются входные векторы и создается место для хранения результата.
2. Затем происходит циклическое выполнение операции сложения с использованием векторных инструкций. Это позволяет обрабатывать сразу несколько элементов, что увеличивает скорость выполнения.
3. После завершения цикла результат сохраняется в памяти.

### 4.2. Функция `caml_find`

Функция *caml\_find* служит для поиска заданного элемента в массиве. Она принимает массив и искомый элемент в качестве входных данных и возвращает индекс первого вхождения элемента в массив, или сообщение об ошибке, если элемент не найден.

Процесс реализации этой функции включает в себя несколько этапов:

1. Вначале сохраняется входной массив.
2. Затем происходит циклическое сравнение элементов массива с искомым элементом. Если совпадение найдено, возвращается его индекс.
3. Если после завершения цикла совпадение не найдено, возвращается -1.

Эти функции являются примером использования векторных инструкций, которые позволяют увеличить производительность за счет параллельной обработки данных. Однако стоит отметить, что использование векторных инструкций требует тщательного планирования и может быть неэффективным при работе с небольшими объемами данных или данными, размер которых не кратен размеру вектора.

## 5. Эксперимент

### 5.1. Сложение

Для замеров использовались интринсики для сложения и поиска в массивах. Результаты замеров приведены ниже. Для замеров использовался миникомпьютер RISC-V Sipeed LicheePi 4A Risc-V TH1520

Таблица 1: Результаты замеров сложения

| Array Length | Vector Addition Time | Scalar Addition Time |
|--------------|----------------------|----------------------|
| 1,000,000    | 0.0231 s             | 0.0522 s             |
| 10,000,000   | 0.3164 s             | 0.5412 s             |
| 100,000,000  | 2.8133 s             | 5.2266 s             |

### 5.2. Find

Таблица 2: Результаты замеров поиска

| Array Length | Vector Find Time | Scalar Find Time |
|--------------|------------------|------------------|
| 10,000,000   | 2.475 ns         | 2.650 ns         |
| 100,000,000  | 2.525 ns         | 2.800 ns         |

Как видно, использование векторных инструкций приводит к существенному ускорению сложения массивов, но к незначительному в случае find.

# Заключение

**На данный момент достигнуты следующие результаты:**

1. Реализованы векторные инстринсики из
2. Проведено сравнение производительности инструкций.

**Код проекта** доступен в GitHub-репозитории: <https://github.com/Azatic/ocaml>

## Список литературы

- [1] Brankov Vladimir. What is gained and lost with 63-bit integers? — URL: <https://blog.janestreet.com/what-is-gained-and-lost-with-63-bit-integers/> (дата обращения: 16 июня 2024 г.).
- [2] YARON MINSKY ANIL MADHAVAPEDDY. Real World OCaml. — URL: <https://dev.realworldocaml.org/compiler-backend.html>.
- [3] В. А. Фролов В. А. Галактионов В. В. Санжаров. Исследование технологии RISC-V. — URL: [https://www.ispras.ru/proceedings/docs/2020/32/2/isp\\_32\\_2020\\_2\\_81.pdf](https://www.ispras.ru/proceedings/docs/2020/32/2/isp_32_2020_2_81.pdf).