# ИНСТРУМЕНТ ДЛЯ ОЦЕНКИ КАЧЕСТВА ПОКРЫТИЯ ИНСТРУКЦИЙ КОМПИЛЯТОРАМИ ДЛЯ АРХИТЕКТУРЫ RISC-V

Лекомцев А. А., Сухарев А. Д.
Кафедра системного программирования
Санкт-Петербургский государственный университет
Санкт-Петербург, Российская Федерация
E-mail: alelekomtsev@gmail.com, therain.i@yahoo.com

*В работе рассматривается задача анализа кодогенерации компиляторов и инструмент, разработанный для автоматизации ее решения*

## Введение

RISC-V – открытая расширяемая ISA (Instruction Set Architecture) с модульной структурой. Архитектура является результатом общемирового сотрудничества большого количества компаний, входящих в RISC-V International [1]. Спецификации доступны для свободного и бесплатного использования, что способствует формированию активного сообщества. Отличительной чертой архитектуры является ее простота в сравнении с другими ISA. Стандартный набор команд содержит только 47 базовых инструкций [2]. Добавление функциональности достигается за счет разработки расширений, часть которых входит в стандарт. Благодаря такой модульности обеспечивается высокая расширяемость архитектуры. Производителям аппаратного обеспечения требуется поддерживать только базовый набор, реализация дополнительных инструкций является опциональной.

Примерами дополнительных операций являются умножение и деление целых чисел, атомарные инструкции, векторные операции, операции с числами с плавающей запятой, инструкции для работы с байтами и отдельными битами. Например, bitmanip [3] – стандартный набор расширений RISC-V, который включает в себя группы инструкций, объединенные по их назначению: инструкции для быстрого вычисления адресов, для операций с битами и другие.

Расширение дает возможность использовать вместо нескольких базовых инструкций одну, что зачастую более оптимально как по времени исполнения, так и по энергопотреблению [3]. В связи с тем, что некоторые расширения были введены в стандарт относительно недавно, компиляторы не в полной мере поддерживают распознавание добавленных инструкций в исходном коде.

Поддержка компиляторами инструкций из многочисленных расширений и их использование для оптимизации является актуальной задачей в сообществе.

## I. Мотивация

Одна из часто встречающихся задач в программировании – задача конвертации чисел из представления LE в BE, и наоборот. В стандартном расширении RISC-V bitmanip присутствует инструкция rev8, которая инвертирует порядок байтов в переданном ей регистре [3]. Напишем код, решающий задачу, на языке C (Листинг 1) и сравним способность различных компиляторов транслировать его в rev8.

Сравнение результатов кодогенерации Clang 17.0.1 (Листинг 2) и GCC 13.2.0 (Листинг 3) с параметром `-march=rv64gc_zbb` и уровнем оптимизации `O2` показывает, что Clang транслирует код в rev8, когда как GCC не генерирует данную инструкцию.

```
#include <stdint.h>

uint64_t rev8(uint64_t rs) {
    uint64_t out = 0;
    for (int i = 0; i < 8; i++) {
        uint8_t tmp = rs >> (i * 8);
        out |= (uint64_t) tmp << ((7 - i) * 8);
    }
    return out;
}
```

Листинг 1 – Реализация алгоритма инверсии порядка байтов в 64 битном числе

```
rev8:
    rev8    a0, a0
    ret
```

Листинг 2 – Код, сгенерированный Clang 17.0.1

```
rev8:
    mv      a2,a0
    li      a4,0
    li      a0,0
    li      a6,56
    li      a1,64
.L2:
    srl     a5,a2,a4
    subw    a3,a6,a4
    andi    a5,a5,255
    sll     a5,a5,a3
    addiw   a4,a4,8
    or      a0,a0,a5
    bne     a4,a1,.L2
    ret
```

Листинг 3 – Код, сгенерированный gcc 13.2.0

Возникает потребность в сравнении возможностей кодогенерации различных компиляторов,

а также отслеживании соответствующих изменений.

Нами не было найдено инструментов, автоматизирующих этот процесс и подходящих под данные требования. Ближайшими аналогами можно назвать наборы тестов и утилиты для их запуска, используемые в компиляторе GCC и проекте LLVM. Основная проблема, с которой мы столкнулись при использовании данных инструментов – разный формат тестов. Например, тестируемый код в проекте LLVM написан на языке LLVM IR [4], тогда как в GCC – на языке C [5], что усложняет сравнение. Следовательно, было принято решение разработать инструмент для анализа возможностей кодогенерации различных компиляторов с единым форматом тестов.

## II. Разработка инструмента

Основными требованиями для инструмента стали поддержка единого формата тестов, а также возможность сравнения кодогенерации различных компиляторов на разных уровнях оптимизации. В качестве дополнительных требований были выдвинуты:

- гибкость конфигурации;
- скорость выполнения;
- простота добавления функциональности;
- кросс-платформенность.

Для тестируемого кода был выбран язык C, так как за основу тестируемых компиляторов были взяты Clang и GCC. Также было принято решение структурировать тесты по каталогам, соответствующим отдельным инструкциям, что позволило писать несколько тестов для одной инструкции. Для удобного сравнения возможностей кодогенерации компиляторов формат CSV был выбран в качестве основного представления результатов тестирования.

Для написания инструмента был выбран язык программирования Python, что обеспечило высокую скорость разработки и кроссплатформенность проекта.

Высокая скорость выполнения тестов была достигнута за счет реализации асинхронного исполнения с использованием библиотеки asyncio из стандартной библиотеки Python. В сравнении с синхронным запуском тестов такой подход обеспечил трехкратный прирост в производительности инструмента.

Для конфигурации инструмента был выбран формат json. В конфигурационном файле пользователь может установить каталог, в котором находятся тесты, путь до тестируемого компилятора, а также параметры его запуска и набор интересующих уровней оптимизации. Так достигается высокая гибкость конфигурации тестирования.

## III. Пример работы инструмента

Для проверки возможностей кодогенерации необходимо было написать ряд тестов. В качестве тестовых случаев было принято решение использовать инструкции из расширения bitmanip. В документации для каждой инструкции этого расширения есть псевдокод, который описывает семантику инструкции [3]. Этот псевдокод был адаптирован для отдельных инструкций, а также были описаны дополнительные случаи, которые компилятор может транслировать в инструкции расширения. Тестируемыми компиляторами были выбраны Clang 17.0.4 и GCC 13.01.0, учитывались все уровни оптизимации. Пример результатов тестирования был добавлен в репозиторий в каталог example [5].

Для тестирования было составлено 96 тестовых сценариев, которые задавались одним из 24 шаблонов и одним из 4 уровней оптимизации компилятора. Clang 17.0.4 породил ожидаемую инструкцию в 45 случаях, GCC 13.01.0 – в 39 случаях.

## Заключение

Некоторые расширения RISC-V были введены в стандарт относительно недавно, а соответственно компиляторы не в полной мере поддерживают распознавание новых инструкций в исходном коде программ. В сообществе актуальна задача сравнения возможностей кодогенерации различных компиляторов и их версий для инструкций из различных расширений.

Разработанный инструмент помогает автоматизировать процесс такого сравнения и анализа, предоставляя единый формат тестов с возможностью гибкой конфигурации.

## Список литературы


1. RISC-V International Members https://riscv.org/members/ [Электронный ресурс] / URL: https://riscv.org/members/ – 2023
2. The RISC-V Instruction Set Manual [Электронный ресурс] / URL: `https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf` – 2023
3. RISC-V Bit-manipulation A, B, C and S Extensions [Электронный ресурс] / URL: `https://five-embeddev.com/riscv-bitmanip/draft/bitmanip.html` – 2023
4. Электронный ресурс / URL: `https://github.com/llvm/llvm-project/blob/main/llvm/test/CodeGen/RISCV/rv32zba.ll` – 2023
5. Электронный ресурс / URL: `https://github.com/GCC-mirror/gcc/blob/master/gcc/testsuite/gcc.target/riscv/zba-shadd.c` – 2023
6. Репозиторий с реализацией инструмента [Электронный ресурс] / URL: `https://github.com/vacmannnn/riscv-check` – 2023