Министерство образования и науки Российской Федерации
Московский физико-технический институт (государственный университет)

Физтех-школа радиотехники и компьютерных технологий
Кафедра Микропроцессорных технологий в интеллектуальных системах управления
Syntacore

Выпускная квалификационная работа бакалавра

Гибкий подход к подъёму LLVM MIR кода открытой архитектуры RISC-V в SSA форму LLVM IR

Автор:

Студент Б01-110 группы
Романов Александр Викторович

Научный руководитель:

Владимиров Константин Игоревич

Москва 2025

Аннотация

Гибкий подход к подъёму LLVM MIR кода открытой архитектуры RISC-V в SSA форму LLVM IR

Романов Александр Викторович

Проблема бинарной совместимости програм и их переносимости на разные архитектуры без возможности перекомпиляции часто решается при помощи бинарных трансляторов. Существует большое колличество статических и динамических бинарных трансляторов. Большинство из них работают либо за счёт прямого сопоставления инструкциям и регистрам исходной архитектуры инструкции и регистры целевой архитектуры, либо за счёт паттерн матчинга. Такие решения делают сложным поддержание новых исходных архитектур ввиду чего поддержка относительно молодой микропроцессорной архитектуры RISC-V в существующих трансляторах либо отсутсвует, либо сильно ограничена.

В данной работе рассмотрен новый инструмент для подъёма машинно зависимого представления RISC-V кода LLVM MIR в высокоуровневое машинно-независимое представление LLVM IR и его применение для простой статической трансляции бинарного RISC-V кода на любую поддержанную LLVM архитектуру.

Содержание

1 Введение 5
1.1 Бинарная совместимость 5
1.2 Архитектура RISC-V 6
1.3 Компилятор LLVM 7
1.3.1 LLVM IR 9
1.3.2 LLVM MIR 10
1.3.3 Оптимизации в компиляторе LLVM 10
2 Постановка Задачи 11
3 Обзор существующих решений 12
3.1 Эмуляция 12
3.2 Бинарная трансляция 12
3.3 Лифтеры в высокоуровневое представление 13
3.4 Вывод 14
4 Исследование и построение решения задачи 15
4.1 Использование описания архитектуры из LLVM 15
4.2 Универсальная модель инструкции RISC-V 17
4.3 Собственное соглашение о вызовах 18
4.4 Вывод 19
5 Описание практической части 19
5.1 Модель инструкции RISC-V 19
5.1.1 Регулярные инструкции 20
5.1.2 Инструкции безусловного перехода 20
5.1.3 Инструкции условного перехода 20
5.1.4 Инструкции вызова функций 22
5.2 Имплементация транслятора 23
5.2.1 Конфигурация целевой архитектуры 23
5.2.2 Алгоритм трансляции 25
5.2.2.1 Создание объектов функции и типа контекста 25
5.2.2.2 Выгрузка регистров из контекста 26
5.2.2.3 Сохранение регистров в контекст 27

5.2.2.4 Замена инструкций 28
5.2.3 Оптимизация транслированного кода 29
5.2.3.1 Оптимизация стекового пространства 30
5.2.3.2 Подстановка тела функций 30
5.3 Применение инструмента к тестированию компилятора clang 31
5.4 Результаты 32
6 Заключение 34
Список Литературы 35

1 Введение

Проблема переноса программ между различными архитектурами, в ситуациях, когда перекомпиляция из изходных файлов сложна, либо вовсе невозможно является одной из актуальных проблем компьютерных технологий. Основным способом решения этой проблемы является бинарная трансляция, которая решает проблемы бинарной совместимости путём преобразования машинного кода исходной архитектуры в машинный код целевой архитектуры. С развитием новых архитектур и операционных систем такое преобразование кода становится всё более сложной, что делает развитие бинарных трансляторов важной задачей современной вычислительной техники.

1.1 Бинарная совместимость

Бинарный код состоит из закодированных инструкций для конкретной архитектуры комманд. При компиляции программы её код на высокоуровневом языке програмирования (Например C, C++ или Fortran) переводится в бинарный код целевой архитектуры и операционной системы.

Бинарной совместимостью называется возможность исполнения бинарного кода, скомпилированного под одну архитектуру команд и операционную систему на других устройствах и системах без модификации этой программы. Бинарная совместимость является одной из фундаментальных проблем в сфере компьютерных технологий в связи с постоянным развитием архитектур набора команд и операционных систем.

Основными проблемами для бинарной совместимости являются:

1. Различные архитектуры команд (ISA). Процессорные архитектуры являются главной причиной бинарной несовместимости. Процессоры каждой архитектуры исполняют свой уникальный набор команд и не работают с другими. Кроме различия в наборе инструкций архитектуры могут также отличаются размером инструкций. Например, X86 и RISC-V поддерживают инструкции разной длины, в то время как ARM фиксирует длину всех инструкций в 4 байта. Архитектуры также отличаются набором регистров, принципами доступа к памяти а также порядком байт (например big-endian или little-endian). В то время как разница в наборе инструкций чаще всего влечёт за собой быструю остановку программы из-за невалидной инструкции, разница в порядке доступов к памяти при прочих равных может вызывать непредсказуемое поведение программы.
2. Операционные системы (ОС) также играют большую роль в бинарной несовместимости. Набор и механизм системных вызовов отличается на разных платформах.

Например, `open` для Linux систем и для FreeBSD работают по разному, несмотря на общее название. Наборы системных вызовов также могут отличатся от версии к версии одной операционной системы. Например Windows не имеет фиксированного набора системных вызовов и они часто изменяются между версиями.

3. Соглашением о вызовах (ABI – Application Binary Interface) называется документ, описывающий обязаности вызывающей (caller) и вызываемой (callee) функций. Описание ABI стандартизирует то, как функция ожидает получить свои аргументы и как она возвращает свой результат (через выделенный регистр или програмный стек). Соглашения о вызовах также могут значительно отличаются даже внутри одной архитектуры. Например программа, написанная под RISC-V процессор с `lp64d` ABI не будет работать для RISC-V с `lp64`.
4. Наконец, окружение запуска (Набор доступных на момент запуска динамических библиотек) является критически важным для запуска программы и может значительно отличаться как от машины к машине, так и на разных версиях операционной системы. Например программа, слинкованная динамически для операционной системы Ubuntu не сможет найти динамические библиотеки на устройстве с операционной системой Arch Linux, т.к. эти библиотеки будут установлены по другим путям.

1.2 Архитектура RISC-V

RISC-V – это открытая и расширяемая архитектура команд, разработанная в университете Беркли (Калифорния, США) в 2013 году (см. [Wat+13]). Архитектура имеет модульную основу, что в совокупности с открытой лицензией позволяет кому угодно разрабатывать и создавать процессоры, с лёгкостью добавляя расширения, специфичные для их задачи, без необходимости платить за лицензию. Такая доступность и расширяемость архитектуры быстро сделала её популярной как в академической среде (В которой RISC-V де-факто является стандартом для обучения микроархитектуре), так и в среде разработчиков микроконтроллеров и даже выскопроизводительных систем. В отличии от многих других архитектур (Таких как X86 и ARM) RISC-V является крайне минималистичной и имеет меньше 50 инструкций в базовом наборе команд который, в который не входит даже аппаратное умножение и деление чисел. Такая степень модульности означает минимализацию поддержки, необходимой для обратной совместимости, в то время как конкурирующим платформам приходится долго с этим бороться.

Т.к. RISC-V имеет модульную архитектуру, то вопрос бинарной совместимости остро стоит даже между разными конфигурациями RISC-V микропроцессоров. Например, программа, написанная под rv64ic (включающий в себя расширение для сжатых инструкций) никогда не запустится на устройстве с rv64i, на котором эти сжатые инструкции не поддержаны. Учитывая наличие не только стандартных, но и пользовательских расширений, RISC-V представляет собой бесконечный набор конфигураций процессорных ядер, бинарная совместимость программ между которыми является скорее исключением, чем правилом. Это придаёт вопросу бинарной трансляции между разными RISC-V ядрами высокий приоритет.

1.3 Компилятор LLVM

Компилятор – это программа, переводящая код программы с высокоуровневого языка програмирования в машинный код заданной архитектуры. В современном мире подавляющее число компиляторов являются оптимизирующими, т.е. компиляторами, производящими широкий спектр оптимизаций над исходным кодом. В своей работе компиляторы используют различные промежуточные представления для исходного кода, наиболее частыми из которых являются:

- AST (Abstract Syntax Tree) – Дерево абстрактого синтаксиса является структурой данных, отражающей программу написанную на высокоуровневом языке програмирования. Такое дерево называется абстрактным, так как оно отражает не реальный синтаксис программы, а лишь его смысловую часть, необходимую для дальнейшей компиляции программы. Естественным образом AST является специфичным для конкретного языка програмирования представлением.
- HIR (High-level Intermediate Representation) – Высокоуровневое промежуточное представление, абстрагирующее код от исходного языка програмирования, но всё ещё независимое от архитектуры. Состоит из виртуальных инструкций, сохраняющих семантику программы, что позволяет производить на данном представлении большое число машиннонезависимых оптимизаций (Например продвижение констант, удаление мёртвого кода или подстановка функций). Такое высокоуровневое представление прекрасно подходит для оптимизации, анализа и преобразования кода, незадумываясь о языке програмирования из которого он был получен или о целевой архитектуре.
- LIR (Low-level Intermediate Representation) – представление, в которое компилятор переводит HIR после того, как все возможные высокоуровневые оптимизации были выполнены. Низкоуровневое представление, как следует из его названия, близко к

машинному коду и состоит уже из конкретных инструкций целевой архитектуры (Или псевдо-инструкций, которые раскрываются в конкретные инструкции в процессе дальнейшей компиляции). Кроме выбора инструкций и регистров целевой архитектуры, LIR также позволяет производить ряд машинно-зависимых оптимизаций, т.е. изменение машинных инструкций, регистров или их порядка с целью повышения производительности. Примером такой оптимизации является замена инструкции `ADD` инструкцией `LEA` на архитектуре X86, с целью повышения производительности.

Поговорим чуть более подробно о высокоуровневом промежуточном представлении. Важным понятием является SSA (Static Single Assignment) форма кода, придуманная в 1980-х годах [CLZ86]. SSA отличается от других представления HIR тем, что каждая переменная в нём присваивается только один раз. Преимущества такой формы кода заключается в том, что все значения никогда не изменяются, это позволяет строить цепочки определений и использований (def-use chains), по которым удобно анализировать зависимости инструкций по данным. В точках схождения графа управления SSA код вставляет φ-функции, которые принимают пары из значений и базовых блоков, из которых пришло управление (Изначально придуманные в [RWZ88]). Сейчас SSA используется во всех конкурентноспособных компиляторах, в том числе GNU Compiler Collection и LLVM.

Наконец поговорим о компиляторе LLVM. LLVM (акроним от Low Level Virual Machines) – компиляторная инфраструктура, придуманная Крисом Латтнером в 2004 году [LA04]. Данная платформа выгодно отличается от других кмпиляторов (например GCC) своей модульной структурой, позволяющей переиспользовать отдельные её компоненты по отдельности. Такая простота в использовании позволила LLVM стать одной из самых популярных компиляторных платформ. На основе инфраструктуры LLVM разработаны:

- Компиляторы таких высокоуровневых языков програмирования как C, C++, Fortran, Rust, Go и т.д.
- Отладчики (LLDB)
- JIT (Just In Time) компиляторы для таких языков как Java, Lua и Scala
- Генераторы случайных тестов [BA24]

Для дальнейших изложений необходимо познакомиться с HIR и LIR представлениями из инфраструктуры LLVM.

1.3.1 LLVM IR

LLVM IR – высокоуровневое промежуточное представление (HIR), используемое компиляторами на основе LLVM. LLVM IR имеет вид набора модулей (Чаще всего полученных из единиц трансляции высокоуровневых языков програмирования), каждый из которых может содержать функции, глобальные объекты и метаинформацию, необходимую для дальнейшей работы с этими модулями. Функции состоят из базовых блоков, которые, в свою очередь, представляют собой список последовательно исполняющихся инструкций, заканчивающийся инструкцией-терминатором. Терминаторы – инструкции, указывающие, какому базовому блоку следует передать управление. Название «Терминатор» появляется из того, что эти инструкции заканчивают базовые блоки.

Все инструкции в LLVM IR работают над значениями в SSA форме. Каждая инструкция, базовый блок или глобальный объект определяют новое уникальное значение, которое далее может быть операндом любой другой инструкции. Все операнды инструкций и их результаты имеют строго определённый тип. При схождении управления LLVM IR вставляет φ-функции для выбора нового значения. В качестве примера приведём функцию на LLVM IR, рекурсивно вычисляющую факториал 32-битного числа и записывающую каждое промежуточное значение в глобальную переменную:

```LLVM
@res = global i32 0, align 4
define i32 @fact(int)(i32 %0) {
  %2 = icmp eq i32 %0, 1
  br i1 %2, label %3, label %5
3:
  %4 = phi i32 [ %8, %5 ], [ 1, %1 ]
  ret i32 %4
5:
  %6 = add nsw i32 %0, -1
  %7 = call i32 @fact(int)(i32 %6)
  %8 = mul nsw i32 %7, %0
  store i32 %8, ptr @res, align 4
  br label %3
}
```

Листинг 1. Вычисление факториала на LLVM IR

1.3.2 LLVM MIR

LLVM MIR (Machine IR)– низкоуровневое промежуточное представление (LIR), используемое компиляторами на основе LLVM. MIR код, так же как и LLVM IR состоит из функций, базовых блоков и инструкций. Однако это представление уже не является SSA формой и инструкции в его составе соответствуют конкретным инструкциям целевой архитектуры или являются псевдо-инструкциями, которые раскрываются в настоящие по мере компиляции. Как уже было сказано, MIR работает не с SSA значениями, а уже с физическими или виртуальными регистрами. В данной работе мы будет рассматривать MIR работающий только с физическими регистрами.

```MIR
name: fact
body: |
  bb.0: successors: %bb.1, %bb.3
    $x8 = COPY $x10
    $x10 = ADDIW $x10, -1
    BNE $x10, $x0, %bb.3
  bb.1: successors: %bb.2
    $x10 = ADDI $x0, 1
  bb.2:
    PseudoRET $x10
  bb.3: successors: %bb.2
    liveins: $x8, $x10
    PseudoCALL @fact, implicit-def $x1, implicit $x10, implicit-def $x2, implicit-def $x10
    $x10 = MULW $x10, $x8
    J %bb.2
```

Листинг 2. Вычисление факториала на LLVM MIR для RISC-V

В Листинг 2 видно конкретные инструкции RISC-V, а так же две псевдоинструкции: `PseudoCALL` и `PseudoRET`, которые при компиляции этого кода раскрылись бы в соответствующие инструкции RISC-V. Как видно, MIR – это представление, позволяющее работать с конкретными инструкция и регистрами целевой архитектуры, не теряя при этом информации о базовых блоках, функциях и других высокоуровневых объектах.

1.3.3 Оптимизации в компиляторе LLVM

Являясь развитой компиляторной инфраструктурой, LLVM предоставляет большое число различных компиляторных оптимизаций. Вместо отдельных функций с нестабильным интерфейсом LLVM предоставляет стабильный интерфейс проходов

(passes), единый для всех реализованных оптимизаций. Всё это позволяет внешним проектам переиспользовать внутренние оптимизации кода (как LLVM IR, так и MIR).

2 Постановка Задачи

Разработка бинарных трансляторов является сложной задачей ввиду большого набора архитектур и особенностей систем их комманд. В задаче бинарной трансляции также стоит учитывать производительность транслированного кода, ради улучшения которой над транслированным кодом приходится производить различные оптимизации, написание которых само по себе является достаточно сложной задачей. Также имеет значимость задача восстановления высокоуровневого представления бинарных програм для их анализа, формальной верификации и оптимизации. Наиболее развитым из таких высокоуровневых представлений является LLVM IR. Актуальность открытой и расширяемой архитектуры RISC-V также растёт и всё больше промышленных производителей начинают выпускать чипы на базе этого набора команд.

Предлагается разработать способ поднятия машинного кода открытой и расширяемой архитектуры RISC-V в высокоуровневое представление LLVM IR. В связи с непрерывной разработкой новых расширений для RISC-V нужен способ быстрой их поддержки. Таким образом поднять код требуется способом, позволяющим лёгкое добавление поддержки новых расширений этой архитектуры.

Итого основная цель данной работы – разработать инструмент для восстановления LLVM IR из машинного кода открытой и расширяемой архитектуры RISC-V.

Данную цель разобьём на следующие шаги:

1. Изучить существующие подходы к восстановлению высокоуровневого представления для разных архитектур набора комманд.
2. Разработать модель для семантического переноса инструкций RISC-V, а именно для rv64im и rv32im.
3. Разработать инструмент перевода кода в LLVM IR на основе построенной модели и инфраструктуры LLVM.

3 Обзор существующих решений

Наиболее распространёнными подходами к решению бинарной несовместимости являются бинарные трансляторы и эмуляторы. Рассмотрим оба понятия.

3.1 Эмуляция

Эмуляторы – инструменты, позволяющие исполнять машинный код другой архитектуры при помощи симуляции всей исходной архитектуры, включая симуляцию регистров, особенностей подсистемы памяти и конвеера исполнения. Эмуляторы часто являются самым простым способом исполнения кода несовместимой архитектуры. Такой подход отличается невысокой производительностью, следующей из затрат на точную симуляцию всех параметров исходной системы.

3.2 Бинарная трансляция

Бинарная трансляция – процесс перевода бинарного кода исходной архитектуры в бинарный код целевой архитектуры. Бинарная трансляция может происходить на уровне процессорных микросхем, либо с помощью программ, называемых бинарными трансляторами. В данной работе мы будем фокусироваться на програмной бинарной трансляции. Такие трансляторы можно разделить на два типа: Статические и динамические.

- Статическими бинарными трансляторами называются программы, принимающие на вход исполняемый бинарный файл исходной архитектуры целиком и преобразующие его в исполняемый файл целевой архитектуры. В процессе статической бинарной трансляции процесс перевода кода на другую архитектуру и его исполнение разделены: Сначала переводится весь код целиком, после он может быть исполнен. Такой вид бинарной трансляции может быть выполнен без возможности исполнить исходный код. Этот подход усложняется тем, что не весь исполняемый код программы может быть доступен бинарному транслятору. Например, некоторые куски кода и метки могут достигаться программой через косвенные переходы (передача управления при помощи прыжка по адресу, записанному в регистре или в памяти), которые может быть тяжело или вовсе невозможно анализировать без предварительного исполнения. Часто статическая бинарная трансляция производится за счёт декомпиляции – перевода кода в выскоуровневое представление (Например с помощью [Roh19]), обратной инженерии и последующей компиляции на целевую архитектуру.

- Динамические бинарные трансляторы – программы, переводящие исходный машинный код на целевую архитектуру по необходимости. В этом подходе транслируется маленький кусок кода (чаще всего в пределах одного базового блока), после чего сразу исполняется на целевой архитектуре и контекст исполнения (значения регистров и памяти) сохраняется. При достижении инструкций перехода начинает транслироваться новый кусок кода или (в случае циклов) исполняться уже транслированный. Стоит отметить, что динамическая трансляция отличается от эмуляции архитектуры, ведь при таком подходе инструкции исходной архитектуры напрямую переводятся в инструкции целевой архитектуры, без симуляции такого контекста исходной архитектуры, как регистров, таблицы прерываний и специфики памяти. Это означает, что динамические бинарные трансляторы в среднем обладают большей производительностью чем эмуляторы, что позволяет применять их в более широком спектре задач. Приведём наиболее иллюстративные примеры динамических бинарных трансляторов:
  - Rosetta – транслятор, который использовался компанией Apple для упрощения перехода их персональных компьютеров с Архитектуры PowerPC на X86 в 2005 году [Ros25].
  - IA-32 Execution Layer – динамический бинарный транслятор, разработанный компанией Intel в 2003 году для производительного исполнения 32-битных программ на более новой 64-битной архитектуре Itanium [Bar+03].
  - Berberis – относительно новый инструмент, разработанный Google для запуска RISC-V Android приложений на устройствах с архитектурой X86_64 [Ber25].

Теперь, получив общее представление о эмуляторах и бинарных трансляторах, мы можем поговорить о способах подъёма машинный код в высокоуровневое представление.

3.3 Лифтеры в высокоуровневое представление

Большая часть бинарных трансляторов работает за счёт прямого сопоставления инструкций и регистров целевой архитектуры инструкциям и регистрам исходной (см [Ros25], [Bel05] и [Ber25]). Такой подход делает процесс бинарной трансляции уникальным для каждого сочетания из исходной и целевой архитектур, что делает поддержку новых целевых архитектур сложной задачей. Подъём кода в промежуточное представление и его перекомпиляция на целевую архитектуру – принцип, позволяющий сильно упростить добавление новых целевых архитектур. Такой подход можно применять как для статической, так и для динамической бинарной трансля-

ции. Наиболее популярными высокоуровневыми представлениями для подъёма кода являются язык C и LLVM IR. Мы сфокусируемся на подъёме в машинного кода в LLVM IR, т.к. подъём в C чаще применяется для обратной разработки, а не для бинарной трансляции. Рассмотрим существующие инструменты для получения LLVM IR из машинного кода:

- llvm-mctoll – инструмент, разработанный компанией Microsoft [YS19]. Проект хорошо поддерживает подъём из X86 и ARM кода. Поддержка других архитектур отсутствует, что делает невозможным его использование для архитектуры RISC-V.
- mcsema – другой популярный инструмент для подъёма кода в LLVM IR. Данная программа поддерживает следующие архитектуры: X86, ARM и SPARC, таким образом снова невозможно её использование для наших задач [Fra25].
- rellume – единственный инструмент, способный поднимать не только код под ARM и X86, но и подмножество RISC-V [ES20]. К сожалению, поддерживается только 197 инструкций из всей спецификации архитектуры RISC-V и поддержка новых расширений в этом инструменте осложнена.
- biotite – новый инструмент для подъёма кода, работающий с подмножеством RISC-V кода, но требующий информацию об исходном коде программы, что не подходит для наших задач [Che+25].

3.4 Вывод

После проведения анализа существующих инструментов для подъёма машинного кода в LLVM IR были выявлены закономерности, которым подчиняются все существующие решения. Оценка присущих им недостатков позволяет выявить возможное направление дальнейшего развития этого направления.

Список инструментов, позволяющих поднимать код архитектуры RISC-V в LLVM IR мал и включает в себя всего две программы (см. [ES20] и [Che+25]). Подавляющее большинство существующих решений поддерживают только архитектуры X86 и ARM.

Существующие инструменты подъёма кода открытой архитектуры RISC-V работают лишь с небольшим её подмножеством. Проблема в поддержке всего подмножества RISC-V заключается в её модульной расширяемой системе. С появлением новых стандартных и вендорских расширений требуется модификация исходного кода лифтера и ручная обработка новых инструкций. Эта задача становится сложной из-за того, что в большинстве существующих решений всем инструкциям или их комбинациям исходной архитектуры вручную сопоставляются инструкции LLVM IR,

что порождает большое число краевых случаев и возможных комбинаций для распознавания паттернов.

Поддержка новых расширений для RISC-V также осложняется необходимостью поддерживать низкоуровневую информацию о регистрах и инструкциях. Такая информация необходима для декодирования бинарного машинного кода. Многие из представленных инструментов используют собственное описание инструкций и работают с ними через это самое описание. Например, в [Che+25] всем поддержанным инструкциям вручную сопоставляются их названия и возможные операнды. Далее бинарные файлы декодируются в два шага:

- С помощью программы-дизассемблера исходный машинный код переводится в своё текстовое представление
- Далее, используя информацию о названиях инструкций и их операндах, полученный текст дизассемблера разбивается на инструкции

Очевидным является то, что ручное описание каждой инструкции является предметом потенциальных ошибок. В контексте расширяемой архитектуры RISC-V это значит увеличение числа ошибок с увеличением числа поддерживаемых расширений. Также описание синтаксиса некоторых инструкций (например векторных) требует большой работы.

Таким образом, существующие инструменты для подъёма RISC-V кода в LLVM IR поддерживают лишь малое число инструкций этой архитектуры. Внедрение новых расширений затруднено и архитектура проектов не позволяет работы с вендорскими расширениями без модификации исходного кода и перевыпуска инструмента.

4 Исследование и построение решения задачи

В данной главе мы обсудим возможные решения проблем существующих лифтеров в LLVM IR. Отличительной чертой предложенного подхода будет являться минимализация ручного описания архитектуры.

4.1 Использование описания архитектуры из LLVM

Прежде всего мы решим проблемы поддержания низкоуровневой информации об инструкциях, регистрах и расширениях. Как уже было сказано, многие существующие инструмены поддерживают собственное описание инструкций для работы с машинным кодом. Предлагается полностью решить эту проблему, переиспользовав уже существующие описания. LLVM, являясь компиляторной инфраструктурой, обязан уметь корректно порождать и читать машинный код. Большим плюсом этой

платформы является развитое описание всех инструкций и всех расширений RISC-V (Как и подавляющего большинства других архитектур, таких как X86, ARM, MIPS и даже SPIRV). LLVM обладает информацией о кодировках, операндах, семантике ассемблера всех инструкций из большинства существующих расширений RISC-V. Вместо того, чтобы описывать всю эту информацию вручную предлагается преиспользовать существующее описание инструкций из LLVM. Большим плюсом такого подхода является снижение вероятности ошибки. Все поддержанные в LLVM расширения косвенно тестируются компилятором `clang`. Это означает, что если в описании некого расширения допущена ошибка, то во время компиляции под архитектуру процессора, включающего это расширение, либо произойдёт ошибка, либо исполнение программы будет некорректным, что приведёт к мотивации исправить недочёт в описании для стабильной работы компилятора `clang`. Таким образом в нашем инструменте будет доступна корректная информация о всех расширениях RISC-V, поддержанных компилятором `clang` (Таких расширений на момент написания данной работы больше 100).

Рассмотрим подробнее средства средства взаимодействия с машинным описанием, предоставляемые библиотекой LLVM. Как уже было сказано во введении, MIR является внутренним представлением машинного кода в LLVM. Самым важным из примитивов машинной абстракции MIR является класс инструкции `MachineInstr`. Он предоставляет такую информацию о конкретной инструкции в коде, как её тип, число и типы её операндов и т.д, но является общим для инструкций из всех поддержанных архитектур и расширений RISC-V. С помощью этого объекта можно определить, является ли инструкция передачей управления, работой с памятью или арифметической операцией. Также `MachineInstr` предоставляет информацию о том, какие из операндов инструкции являются регистрами, числами и адресами меток. Данная абстракция удобна тем, что позволяет понять самые важные свойства инструкции не вдаваясь в подробности того, какая именно это инструкция. В дальнейшей работе мы будем работать с машинным кодом, разбитым на функции, являющиеся списком `MachineInstr`. Инфраструктура LLVM позволяет легко осуществить подобное разбиение, в результате которого мы получим MIR представление исходного машинного кода, при этом не описывая вручную ни одну из инструкций спецификации архитектуры RISC-V. Это значительно упрощает поддержку большого числа расширений.

4.2 Универсальная модель инструкции RISC-V

Ещё одним препядствием к простому поддержанию новых расширений RISC-V в существующих инструментах является то, каким образом инструкциям из машинного кода сопоставляется операции в LLVM IR. Чаще всего подъём кода происходит путём индивидуальной обработки каждой из поддержанных инструкций в исходном коде. Это означает, что в определённом месте исходного кода инструмента присутствует обработка сотен краевых случаев (вообще говоря по одному на каждую поддержанную инструкцию). В данной работе предлагается минимизировать число таких краевых случаев путём разбиения всех поддержанных инструкций на типы (полученные из MIR как описано в Раздел 4.1) и дальнейшего описания всех инструкций в фиксированном для каждого из типов формате. Итого предлагается заменить обработку подъёма каждой инструкции (число которых измеряется в сотнях) на обработку нескольких классов инструкций и дальнейшей шаблонной подстановки LLVM IR кода, соответствующего каждой инструкции из класса. Такой подход сильно упрощает сопоставление LLVM IR каждой из инструкций и задаёт конкретный формат этого сопоставления, что позволяет поддерживать новые инструкции в отрыве от исходного кода и его краевых случаев.

Отдельно рассмотрим предлагаемый принцип шаблонизации инструкций RISC-V. Конкретные детали будут различаться для разных классов инструкций, но структура решения будет неизменна. Для осознания принципа разберём самый простой и наиболее часто встречающийся класс инструкций – арифметические инструкции или работа с памятью. Данный класс инструкций предлагается назвать «регулярным» и для сопоставления LLVM IR кода инструкциям этого класса предлагается использовать функции LLVM IR. Другими словами, каждой инструкции, например `ADD`, мы сопоставляем определение функции на LLVM IR по следующему принципу:

- Каждый входной операнд (регистр или число) становится аргументом этой функции.
- Каждый выходной операнд-регистр (все инструкции имеют не более одного) становится возвращаемым значением этой функции.

Рассмотрим данный подход на примере инструкции `ADD` из 64-битного варианта архитектуры RISC-V rv64i. В MIR данная инструкция может иметь вид, указанный в Листинг 3. Ей, по описанным выше правилам, сопоставится функция из Листинг 4. Далее при последовательном переводе иструкци MIR на место инструкции `ADD` мы вставим вызов функции `@ADD`, передав текущие значения входных регистров (`X11` и

X12) как аргументы вызова и сохраним возвращаемое значение как новое значение регистра X10 (см. Листинг 5).

```
$X10 = ADD $X11, $X12
```
MIR

Листинг 3. Инструкция ADD в LLVM IR

```
define i64 @ADD(i64 %rs1, i64 %rs2) {
  %4 = add i64 %rs1, %rs2
  ret i64 %4
}
```
LLVM

Листинг 4. Сопоставленная инструкции ADD функция LLVM IR

```
...
 %new_X10 = call i64 @ADD(i64 %X11, i64 %X12)
...
```
LLVM

Листинг 5. Поднятая в LLVM IR инструкция ADD

4.3 Собственное соглашение о вызовах

Кроме большого числа расширений, RISC-V отличается от других архитектур наличием нескольких соглашений о вызовах функций. Большинство существующих инструментов для подъёма кода в LLVM IR описывают ABI подежриваемых исходных архитектур и при работе с кодом, содержащим несколько функций, стараются восстанавливать сигнатуру исходной функции при помощи таких инструментов как паттерн-матчинг. В связи с наличием нескольких возможных соглашений о вызовах в RISC-V коде мы будем избегать этого. Вместо восстановления сигнатуры функции в этой работе предлагается ввести собственное соглашение о вызовах, которое будет скрывать за собой ABI исходного кода (Аналогично представленному в [PB25]).

Разберём предлагаемое решение. Для этого определим класс контекста (state) как структуру, содержащую текущие значения используемых регистров RISC-V. В случае архитектуры rv64im это – 32 регистра общего назначения с именами X0-X31. Далее каждой функции присвоим сигнатуру, в которой она принимает указатель (ссылку) на наш класс текущего контекста и не возвращает ничего. Иными словами, на языке C поднятая функция foo имела бы вид:

```
void foo(struct state *st);
```
C

Листинг 6. Сигнатура поднятой функции на языке C

Или эквивалентное представление на LLVM IR:

```
declare void @foo(ptr %state)
```
LLVM

Листинг 7. Сигнатура поднятой функции на LLVM IR

Поясним, как такой подход упрощает работу с ABI. Предположим, что в нашем соглашении о вызовах аргументы функции хранятся в регистрах `X10-X15`, а возвращаемое значение сохраняется в регистр `X10` (Аналог `lp64` ABI). Тогда, в процессе перевода кода в LLVM IR инструкции инициализации аргументных регистров переведётся в обновление значений этих самых регистров в нашем классе контекста (Вне зависимости от реального используемого функцией числа аргументов). После этого вызываемая функция получит указатель на этот контекст и далее будет загружать значения регистров из контекста. Заметим, что если бы аргументными регистрами являлись `X5-X10`, то в поднятии кода ничего бы не изменилось, вызывающая функция в таком случае обновила бы значения регистров с пятого по десятый в контексте, а вызываемая использовала бы значения этих регистров из контекста. То же верно и для возвращаемого значения функции, регистр возвращаемого значения будет обновлён в контексте вызываемой функцией и интерпретирован вызывающей функцией как результат.

4.4 Вывод

Таким образом мы установили три решения для трёх разных проблем, присутствующих в существующих инструментах для подъёма кода в LLVM IR. Предложенный подход позволяет значительно снизить сложность поддержки новых расширений RISC-V, переиспользовав как можно больше средств, предоставляемых инфраструктурой LLVM. Объём ручного описания архитектуры сведён к минимуму и включает в себя только непосредственное сопоставление LLVM IR кода исходным инструкциям RISC-V.

5 Описание практической части

В данной главе представлена имплементация идей и подходов к решению задачи подъёма MIR кода архитектуры RISC-V в LLVM IR, описанных в Глава 4. Построена модель инструкций этой архитектуры и описано применение её для трансляции кода RISC-V в высокоуровневое представление LLVM IR.

5.1 Модель инструкции RISC-V

Начнём с построения модели инструкции архитектуры RISC-V, представленной в Раздел 4.2. Как уже было сказано, мы разделим все инструкции из спецификации RISC-V на несколько классов, все инструкции внутри которых будут обрабатываться одинаково. Опишем все выделенные в данной работе классы инструкций.

5.1.1 Регулярные инструкции

Класс «регулярных» инструкций был описан в Раздел 4.2, в этот класс попадёт большинство инструкций из спецификации, в том числе `ADD`, `SLLI`, `MUL` и т.д. В терминах предикатов из MIR инструкция принадлежит к этому классу если выражение на Листинг 8 верно. Как уже было отмечено, инструкции этого класса поднимаются простой итеративной заменой на вызовы соответствующих функций.

```cpp
bool is_regular_instruction(const MachineInstr &minst) {
  return !minst.isTerminator() && !minst.isCall() && !minst.isReturn();
}
```

Листинг 8. Условие принадлежности к классу регулярных инструкций

5.1.2 Инструкции безусловного перехода

Инструкции безусловного перехода – операции, передающие управление другому базовому блоку сразу по их достижению исполнителем. Условием принадлежности к этому классу в MIR является удовлетворение предикату на Листинг 9.

```cpp
bool is_unconditional_branch(const MachineInstr &minst) {
  return minst.isUnconditionalBranch();
}
```

Листинг 9. Условие принадлежности к классу инструкций безусловного перехода

Перевод инструкций из этого класса выполняется путём простой замены этой инструкции на инструкцию безусловного перехода из LLVM IR[1] с блоком назначения, полученным из соответствующего операнда `MachineInstr`. Таким образом простой пример кода с инструкцией безусловного перехода `J` из Листинг 10 переводится в LLVM IR, представленный на Листинг 11.

```mir
bb.0:
  ...
bb.1:
  ...
  J bb.0
```

Листинг 10. Пример MIR кода с инструкцией `J`

```llvm
block0:
  ...
block1:
  ...
  br label %block0
```

Листинг 11. Пример MIR кода с инструкцией `J`

5.1.3 Инструкции условного перехода

Инструкциями условного перехода назовём инструкции, передающие управление другим базовым блокам при соблюдении определённых условий. В терминах предикатов MIR инструкция принадлежит этому классу если верен предикат описанный на Листинг 12.

[1]За подробной информации об инструкции `br` и других инструкциях LLVM IR можно обратиться к [LLV25]

```cpp
bool is_conditional_branch(const MachineInstr &minst) {
  return minst.isConditionalBranch();
}
```

Листинг 12. Условие принадлежности к классу инструкций условного перехода

Определить базовый блок, которому возможна пердача управления, легко обратившись к операндам `MachineInstr`. Условие перехода, однако, определить из описания LLVM невозможно. Воспользуемся тем, что условием всех условных переходов в архитектуре RISC-V является результат сравнения двух регистров[2]. Будем получать условие перехода для инструкций через сопоставление каждой из этих функций булевой функции на LLVM IR. Сопоставлять предлагается по следующему принципу: Значение всех входных операндов-регистров инструкции (выходных операндов нет) передаются в функцию как аргументы. Функция выполняет соответствующую инструкции проверку над этими операндами и возвращает результат в виде булевого значения. Далее вставляется инструкция условного перехода LLVM IR , условием которой является значение, полученное из вызова описанной выше функции. Назначением перехода будет соответствовать базовый блок, полученный из последнего операнда инструкции. В отличии от инструкций условного перехода, инструкция условного перехода `br` в LLVM IR имеет ещё один операнд – базовый блок, в который она передаст управление если условие не выполнено. Значение данного операнда будем выставлять в следующий за этой инструкцией базовый блок/инструкцию.

Приведём пример такого сопоставления для инструкции `BLT` (Branch if Less Then – Переход если меньше). Данная инструкция принимает на вход два регистра и совершает переход, если значение в первом регистре меньше чем во втором. Соответствующая данной инструкции функция в LLVM IR будет иметь вид:

```llvm
define i1 @BLT(i64 %0, i64 %1) {
  %3 = icmp slt i64 %0, %1
  ret i1 %3
}
```

Листинг 13. Функция-условие перехода, соответствующая инструкции `BLT`

Тогда при переводе кода, представленного на Листинг 14, получим LLVM IR представленный на Листинг 15.

[2]Некоторые инструкции из расширения C (например `C_BEQZ`) имеют один входной регистр, который они сравнивают с нулём, однако они не требуют специальной обработки и являются частным случаем инструкций условного перехода в нашей классификации

```
bb.0:                                                    MIR
  ...
  BLT $x10, $X11, bb.0
bb.1:
  ...
```

Листинг 14. Пример MIR кода с инструкцией `BLT` для перевода в LLVM IR

```
block0:                                                  LLVM
  ...
  %cond = i1 call @BLT(i64 %x10, i64 %x11)
  br i1 %cond, label %block0, label %block1
block1:
  ...
```

Листинг 15. Полученный при переводе инструкции `BLT` LLVM IR

5.1.4 Инструкции вызова функций

Вызов функции – передача управления определённой функции с последующим возвратом управления в точку вызова. В отличии от инструкций условного и безусловного перехода, инструкции вызова не прерывают базовые блоки и могут находиться внутри них. С точки зрения MIR инструкция принадлежит этому классу если выполняется предикат, представленный на Листинг 16:

```
boo is_call(const MachineInstr &minst) {                 CPP
  return minst.isCall();
}
```

Листинг 16. Условие принадлежности к классу инструкций вызова

Работа с вызовами функций в нашем трансляторе намного интереснее работы с другими классами инструкций. Как было отмечено в Раздел 4.3, вместо попыток воспроизвести изначальную сигнатуру функции мы будем пользоваться собственным соглашением о вызовах, в котором функции общаются друг с другом, передавая единственный аргумент – ссылку на текущий контекст исполнения.

Работа с контекстом исполнения означает, что на момент вызова и возврата из функций в этом контексте должно храниться актуальное состояние регистров. Таким образом, необходимо записывать обновлённые значения каждого из регистров в контекст. При переводе инструкции вызова `JAL` в RISC-V мы будем заменять её вызовом соотвтетсвующей функции в LLVM IR (см. Листинг 17 и Листинг 18).

```
  JAL bar                                                MIR
```

Листинг 17. Пример MIR кода с инструкцией `JAL` для перевода в LLVM IR

```llvm
call void @bar(ptr %state)
```

Листинг 18. Полученный при переводе инструкции `JAL` LLVM IR

5.2 Имплементация транслятора

Наконец опишем полный алгоритм работы транслятора из RISC-V MIR кода в LLVM IR, основанного на описанных выше принципах (код доступен на [llv25]).

5.2.1 Конфигурация целевой архитектуры

Опишем конкретный использованный формат описания инструкций, о котором говорилось в Раздел 5.1. Как было сказано выше, большинству инструкций из спецификации RISC-V сопоставляются функции из LLVM IR. В ходе работы было принято решение вынести данное описание за пределы исходного кода инструмента. Таким образом пользователь сможет настраивать инструмент для подъёма необходимой ему конфигурации RISC-V процессора без необходимости перекомпиляции проекта из исходного кода.

В качестве языка для описания инструкций был выбран язык разметки YAML (см. [YAM25]). Такой выбор был сделан из-за распространённости инструментов для работы с этим представлением и его простоты к написанию. YAML (рекурсивный акроним для YAML Ain't Markup Language) – Человекочитаемый язык сериализации данных, разработанный в 2001 году. YAML используется для написания конфигурационных файлов, а так же для хранения и передачи данных. Перечислим главные концепции YAML, которыми далее будем пользоваться:

- **Скаляры** – самый примитивный из типов данных, используемых в YAML. Скалярами могут являться целый числа, числа с плавающей точкой или строки. Строки могут содержать символы перехода на новую строку, а ковычки являются опциональными. Особым видом скаляров является `null`, обозначающий отсутствие значения.
- **Сопоставления (словари)** – тип данных, представляющий собой неупорядоченный набор, ассоциирующий ключи со значениями. Сопоставления задаются в YAML через значение, отделённое от своего ключа двоеточием. Ключи обязаны быть уникальными, а значения могут иметь любой из доступных типов: скаляры, другие словари или списки.
- **Списки (коллекции)** – объекты, представляющие собой упорядоченный набор из любых данных. YAML предоставляет два формата задания списков: блочный и однострочный. В блочном формате элементы списка задаются на новой строке,

начинающейся с символа `-`, а в однострочном пишутся внутри квадратных скобок и разделяются запятыми (см. Листинг 19).

```yaml
foo:
- 1
- 2
- 3
bar: [1, 2, 3]
```

Листинг 19. Пример списков в YAML (Здесь и `foo`, и `bar` хранят последовательность чисел от 1 до 3)

Формат YAML конфигурации был выбран следующим:

- Верхнеуровневый ключ `instructions` начинает перечисление описаний для каждой из необходимых инструкций
- Каждая из инструкций в свою очередь хранит текстовое представление соответствующей ей функции LLVM IR под ключём `func`.

Таким образом, пример входной конфигурации архитектуры можно увидеть на Листинг 20. Конфигурационный файл в заданном формате считывается транслятором и описанные в нём функции используются для подъёма MIR кода RISC-V. Функции для инструкций условного перехода задаются в том же формате, пример для инструкции `BLT` представлен на Листинг 21.

```yaml
instructions:
  - OR:
      func: |
        define i64 @OR(i64 %0, i64 %1) {
          %3 = or i64 %1, %0
          ret i64 %3
        }
  - XOR:
      func: |
        define i64 @XOR(i64 %0, i64 %1) {
          %3 = xor i64 %1, %0
          ret i64 %3
        }
```

Листинг 20. Пример конфигурации архитектуры в формате YAML для инструкций `OR` и `XOR`

```yaml
instructions:
  - BLT:
      func: |
        define i1 @BLT(i64 %0, i64 %1) {
          %3 = icmp slt i64 %0, %1
          ret i1 %3
        }
```

Листинг 21. Пример конфигурации архитектуры в формате YAML для инструкции `BLT`

5.2.2 Алгоритм трансляции

Наконец, последовательно опишем алгоритм трансляции модуля LLVM MIR.

5.2.2.1 Создание объектов функции и типа контекста

Начнём с построения класса контекста исполнения `state`. На данный момент данный класс будет хранить в себе исключительно состояний регистров. Таким образом создаётся структура (`struct`) в LLVM IR, хранящая в себе массив значений регистров. Массив используется вместо контейнера, потому что все регистры общего назначения архитектуры RISC-V пронумерованы (От `X0` до `X31` в случае `rv32i`/`rv64i` и от `X0` до `X15` в случае `rv32e`/`rv64e`). Ширина, необходимая для хранения значения регистров выбирается в зависимости от того, поднимается ли код 32-битного или с 64-битного варианта архитектуры RISC-V[3]. Таким образом структура контекста в LLVM IR будет иметь вид:

```llvm
%state = type { [32 x i64] }
```

Листинг 22. Структура контекста для архитектуры `rv64i`

Как было описано в Раздел 1.3.2, Модули в MIR состоят из машинных функций (класс `MachineFunction` в библиотеке LLVM). Прежде всего для каждой машинной функции необходимо создать соответствующую функцию LLVM IR, с сигнатурой, описанной в Раздел 4.3. Т.е. создаётся функция с тем же именем, что и у машинной, которая имеет пустое возвращаемое значение `void` и единственный аргумент `%regstate` типа `ptr`.

Далее для всех базовых блоков изначальной MIR функции необходимо создать новый базовые блоки в LLVM IR. В дальнейшей работе возможно добавление новых базовых блоков для работы с особенностями LLVM IR, но функция будет иметь как минимум столько же блоков, сколько исходная `MachineFunction`. В процессе создания базовых блоков необходимо строить отображение из старых машинных блоков в

[3]Аналогично возможна поддежрка 128-битной архитектуры RISC-V

новые блоки LLVM IR. Данная информация будет необходима при дальнейшей работе с потоком управления.

5.2.2.2 Выгрузка регистров из контекста

Перед началом итеративного подъёма инструкций в MIR функции необходимо выгрузить значения регистров из контекста. Для этого в LLVM необходимо сначала получить адресс массива регистров (назовём его `%GPRS`) из контекста. Это делается через инструкцию `getelementptr`, принимающей в нашем случае два индекса. Первый индекс – индекс структуры контекста в возможном массиве этих контекстов, скрывающимся под указателем `%regstate`, полученным как аргумент функции. В нашем случае этот индекс всегда равен нулю. Второй индекс обозначает номер элемента в структуре `state`. В нашем случае массив регистров – первый элемент структуры, поэтому данный индекс также будет равен нулю. Теперь необходимо получить адрес каждого из регистров (назовём его `%x_i_ptr`, где `i` – номер регистра), хранимых в массиве, указатель на который был только что получен. Для этого снова используется инструкция `getelementptr` первый индекс которой равен нулю как индекс единственного массива в возможном массиве массивов `%GPRS`, а второй равен номеру регистра. После этого загружается значение регистра `%x_i` из указателя `%x_i_ptr`. Загрузка производится при помощи инструкции `load`, загружаемый тип которой выставлен в целочисленной число с размером в ширину регистра (т.е. `i64` для `rv64`).

Теперь полученные из контекста значения регистров складываются на стек. Для этого необходимо выделить для них стековую ячейку `%x_i_stack`. Сделаем это при помощи инструкции `alloca` с соответствующим регистру типом. Теперь можно записать выгруженное ранее значение регистра в подготовленную стековую ячейку при помощи инструкции `store`. Описанные операции выгрузки значений из контекста, выделения места на стеке и сохранение на стек необходимо повторить для всех используемых регистров (см. Листинг 23). Полученный LLVM IR код представлен на Листинг 24.

```cpp
void load_registers_from_state(IRBuilder &builder,
                               std::map<unsigned, Value *> &stack_frame) {
  auto *zero = ConstantInt::zero();
  auto *regs_array = builder.CreateGEP(state_type, state_arg, {zero, zero});
  for (unsigned i = 0; i < num_regs; ++i) {
    auto *idx = ConstantInt::Create(i);
    auto *reg_addr = builder.CreateGEP(array_type, regs_array, {zero, idx});
    auto *val = builder.CreateLoad(reg_type, reg_addr);
    if (!stack_frame.contains(i))
      stack_frame[i] = builder.CreateAlloca(reg_type);
    builder.CreateStore(reg_type, reg_value, stack_frame[i]);
  }
}
```

Листинг 23. Алгоритм выгрузки регистров из контекста

```llvm
define void @foo(ptr %regstate) {
  %GPRS = getelementptr %state, ptr %regstate, i32 0, i32 0
  %x_0_ptr = getelementptr inbounds [32 x i64], ptr %GPRS, i32 0, i32 0
  %x_0_stack = alloca i64, align 8
  %x_0 = load i64, ptr %x_0_ptr, align 8
  store i64 %x_0, ptr %x_0_stack, align 8
  %x_1_ptr = getelementptr inbounds [32 x i64], ptr %GPRS, i32 0, i32 1
  %x_1_stack = alloca i64, align 8
  %x_1 = load i64, ptr %x_1_ptr, align 8
  store i64 %x_1, ptr %x_1_stack, align 8
  ...
  ret void
}
```

Листинг 24. Код аллокации стекового окна и выгрузки 64-битных регистров для функции `foo`

5.2.2.3 Сохранение регистров в контекст

Перед возвратом из функции и перед вызовом других функций необходимо сохранить обновлённые значения регистров в контекст. Опишем алгоритм такого сохранения:

1. Зная значения стековых слотов для регистров `%x_i_stack` и адресс их массива в контексте `%GPRS`, будем загружать значения со стека при помощи инструкции `load` и сохранять их в соответствующие ячейки контекста `%x_i_ptr` (см. псевдокод на Листинг 25). Пример получившегося LLVM IR кода представлен на Листинг 26.

```cpp
void load_registers_from_state(IRBuilder &builder,
                               std::map<unsigned, Value *> &stack_frame) {
  auto *zero = ConstantInt::zero();
  auto *regs_array = builder.CreateGEP(state_type, state_arg, {zero, zero});
  for (unsigned i = 0; i < num_regs; ++i) {
    auto *idx = ConstantInt::Create(i);
    auto *reg_addr = builder.CreateGEP(array_type, regs_array, {zero, idx});
    auto *val = builder.CreateLoad(reg_type, reg_addr);
    if (!stack_frame.contains(i))
      stack_frame[i] = builder.CreateAlloca(reg_type);
    builder.CreateStore(reg_type, reg_value, stack_frame[i]);
  }
}
```

Листинг 25. Алгоритм сохранения регистров в контекст

```llvm
define void @foo(ptr %regstate) {
  ...
  %x_30 = load i64, ptr %x_30_stack, align 8
  store i64 %x_30, ptr %x_30_ptr, align 8
  %x_31 = load i64, ptr %x_31_stack, align 8
  store i64 %x_31, ptr %x_31_ptr, align 8
  ret void
}
```

Листинг 26. Код сохранения 64-битных регистров в контекст

5.2.2.4 Замена инструкций

Теперь, сохранив значения всех регистров, можно начать заполнять LLVM IR переведёнными машинными инструкциями. Для этого пройдёмся по всем базовым блокам в машинной функции и будем вставлять их поднятое представление в соответствующие блоки LLVM IR. Итого для каждой инструкции применяется следующая операция:

1. Если инструкция является регулярной – вставить вызов соответствующей ей функции LLVM IR.
2. Если инструкция является безусловным переходом – вставить инструкцию безусловного кода LLVM IR с назначением равным блоку, соответствующему текущему MIR блоку.
3. Если инструкция является условным переходом – вставить вызов соответствующей ей функции-условия, после чего вставить инструкцию условного перехода `br i1` в LLVM IR. При соблюдении условия назначением перехода является блок, соот-

ветствующий блоку-назначению из MIR. В противном случае блоком назначения является следующий за текущим базовый блок.

4. Если инструкция является вызовом функции – сохранить значения всех регистров в контекст, вставить вызов соответствующей LLVM IR функции, передав ей указатель на контекст, после чего снова выгрузить обновлённые значения регистров на стек.
5. Если инструкция является инструкцией возврата – сохранить значения всех регистров в контекст и вставить инструкцию `ret`

Псевдокод данной операции представлен на Листинг 27

5.2.3 Оптимизация транслированного кода

Выше был описан алгоритм подъёма модуля MIR кода архитектуры RISC-V в LLVM IR. Легко заметить усложнение транслируемого кода при его подъёме. В несколько раз увеличено число вызовов функций (В среднем по одному на каждую инструкцию исходного кода). Также на каждый регистр была выделена отдельная стековая ячейка. При этом каждое чтение регистра заменено на загрузку элемента со стека, а

```cpp
Value *lift_instruction(const MachineInstr &minst,
                        const blocks_map &blocks, IRBuilder &builder) {
  if (is_regular(minst))
    return builder.CreateCall(get_function_for(minst));
  if (is_unconditional_branch(minst))
    return builder.CreateBr(bb[get_destination_block(minst)]);
  if (is_conditional_branch(minst)) {
    auto *cond = builder.CreateCall(get_function_for(minst));
    return builder.CreateCondBr(cond, get_true_block(minst), get_false_block(minst));
  }
  if (is_call(minst)) {
    save_registers_to_state(builder);
    auto *call = builder.CreateCall(get_dest_function(minst));
    load_registers_from_state(builder);
    return call;
  }
  if (is_return(minst)) {
    save_registers_to_state(builder)
    builder.CreateRetVoid();
  }
  ...
}
```

Листинг 27. Псевдокод операции подъёма одной инструкции

каждая запись в регистр заменена на запись в память. Такие изменения структуры кода не только делают его более сложным для анализа, но и влекут за собой потенциальные потери в производительности[4]. В этом разделе будут рассмотрены использованные меры по упрощению структуры и повышению производительности транслированного кода. LLVM является развитой компиляторной инфраструктурой и, как было описано ранее (см. Раздел 1.3.3), имеет большое число встроенных оптимизаций как высокого, так и низкого, машинного уровня. Мы будем использовать предосталвенные LLVM оптимизации в виде проходов.

5.2.3.1 Оптимизация стекового пространства

Прежде всего избавимся от лишних аллокаций стекового пространства. Инфраструктура LLVM предоставляет оптимизацию SROA (Scalar Replacement of Aggregates – Замещение агрегатов скалярами). Данная оптимизация заменяет аллокации агрегатных типов (Чаще всего аллокации массивов с помощью инструкции `alloca`) на отдельные ячейки стека, полученные несколькими инструкциями `alloca`. Также данная оптимизация способна полностью удалять выделение стека и строить полноценную SSA форму кода. Мы применим этот оптимизационный проход к полученному при трансляции LLVM IR для удаления ячеек стека и построения более простой для анализа и дальнейших оптимизаций SSA форму LLVM IR. Пример кода до и после такой оптимизации можно видеть на Листинг 28 и Листинг 29, важно обратить внимание на исчезновение инструкций загрузки и записи, а также на появление характерных для SSA формы φ-функций.

5.2.3.2 Подстановка тела функций

Для уменьшения числа лишних вызовов и дальнейшей оптимизации построенной SSA формы кода применим оптимизацию подстановки тела функций. Данная оптимизация заменяет вызов функции подстановкой кода этой самой функции. При этом значения параметров функции заменятся значениями поданных аргументов, которые напрямую подставляются в операции из тела этой самой функции. Далее оптимизация может обнаружить и убрать лишние SSA значения в LLVM IR. Таким образом, применив подстановку функций к LLVM IR коду, представленному на Листинг 29, получим ещё более оптимизированный код, приведённый на Листинг 30.

[4]Эта работа не концентрируется на повышении производительности транслированного кода, но предпринимает базовые шаги к её улучшению

```llvm
106: ; preds = %106, %1
  %107 = load i64, ptr %87, align 8
  %108 = load i64, ptr %75, align 8
  %109 = call i64 @ADD(i64 %107, i64 %108)
  store i64 %109, ptr %54, align 8
  %110 = load i64, ptr %48, align 8
  %111 = load i64, ptr %60, align 8
  %112 = call i64 @MULW(i64 %110, i64 %111)
  store i64 %112, ptr %33, align 8
  %113 = load i64, ptr %81, align 8
  %114 = call i64 @ADD(i64 %113, i64 0)
  store i64 %114, ptr %81, align 8
  %115 = load i64, ptr %81, align 8
  %116 = load i64, ptr %9, align 8
  %117 = call i64 @SRL(i64 %115, i64 %116)
  store i64 %117, ptr %60, align 8
  %118 = load i64, ptr %60, align 8
  %119 = call i1 @BEQ(i64 %118, i64 0)
  br i1 %119, label %106, label %120
```

Листинг 28. Блок LLVM IR до оптимизации SROA

```llvm
70: ; preds = %70, %1
  %.079 = phi i64 [ %69, %1 ], [ %74, %70 ]
  %.077 = phi i64 [ %68, %1 ], [ %73, %70 ]
  %71 = call i64 @ADD(i64 %59, i64 %51)
  %72 = call i64 @MULW(i64 %66, i64 %.079)
  %73 = call i64 @ADD(i64 %.077, i64 0)
  %74 = call i64 @SRL(i64 %73, i64 %7)
  %75 = call i1 @BEQ(i64 %74, i64 0)
  br i1 %75, label %70, label %76
```

Листинг 29. Блок LLVM IR после оптимизации SROA

```llvm
68 ; preds = %68, %1
  %.079 = phi i64 [ 0, %1 ], [ %72, %68 ]
  %69 = add i64 %51, %59
  %70 = mul i64 %66, %.079
  %71 = and i64 %7, 63
  %72 = lshr i64 0, %71
  %73 = icmp eq i64 %72, 0
  br i1 %73, label %68, label %74
```

Листинг 30. Блок LLVM IR после подстановки тела функции

5.3 Применение инструмента к тестированию компилятора clang

Полученный транслятор из RISC-V машинного и MIR кода был использован для тестирования компилятора `clang`. Тестирование производилось путём генерации случайного тестового RISC-V MIR кода, которые после этого компилировался до машинного кода RISC-V с минимальным уровнем оптимизаций `-O0`, после чего

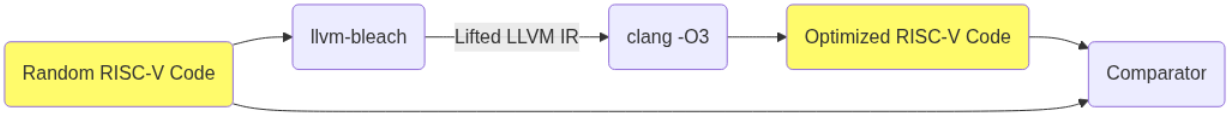

Рис. 1. Поддержка расширений RISC-V в разработанном инструменте llvm-bleach

поднимался в LLVM IR через представленный в этой работе инструмент llvm-bleach и снова компилировался до RISC-V машинного кода с максимальным уровнем компиляторных оптимизаций `-O3`. Обе полученный программы далее исполнялись на функциональном симуляторе архитектуры RISC-V (см. [Spi25]), и сравнивались значения регистров. Схематически процесс тестирования описан на Рис. 1.

В результате такого тестирования была обнаружена проблема, при которой компилятор `clang` переставлял местами инструкции выставления режима округления для чисел с плавающей точкой и арифметические операции с плавающей точкой, что приводило к расхождению в результатах. Подробное описание ошибки можно найти в [Mac25].

5.4 Результаты

При описании инструкций архитектуры RISC-V в предложеном формате (см. Раздел 5.2.1) было реализовано 50 инструкций из базового набора команд `rv64i/e` (94%), 50 инструкций для `rv32i/e` (93%), также полностью поддержаны все 12 инструкции из расширения `M`. Дополнительно были поддержаны некоторые инструкции из расширений для чисел с плавающей точкой разной точности `F` и `D`. Визуализация поддежанных инструкций представлена на Рис. 2.

Был разработан инструмент, названный `llvm-bleach` и способный к подъёму машинного и MIR кода RISC-V в высокоуровневое промежуточное представление LLVM IR. Данный инструмент был применён в статической бинарной трансляции кода с `rv64im` на `X86`, а так же тестирования компилятора `clang`.

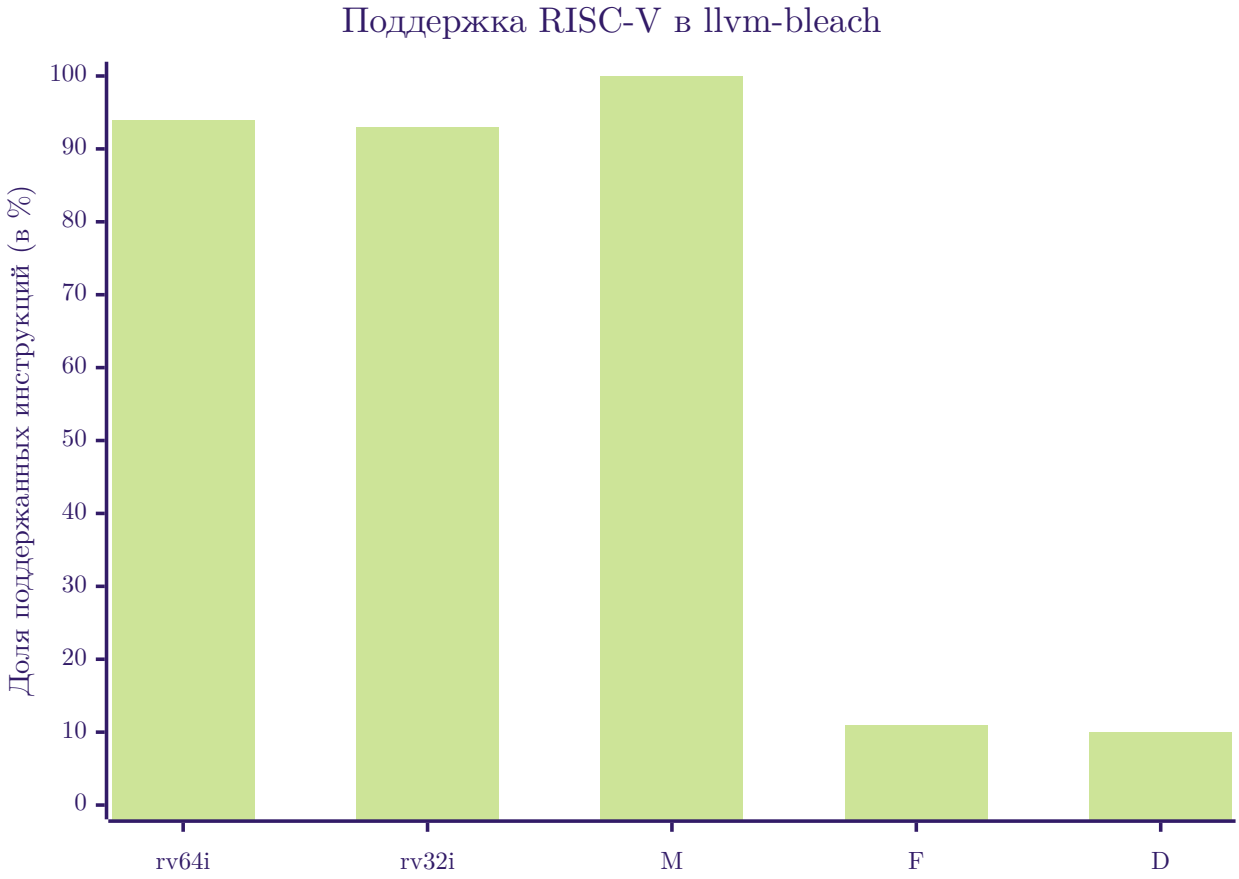

Рис. 2. Поддержка расширений RISC-V в разработанном инструменте llvm-bleach

6 Заключение

В данной работе была исследована задача трансляции машинного кода открытой и расширяемой архитектуры RISC-V на другие архитектуры и высокоуровневое представление LLVM IR. Исследование существующих решений показало, что инструменты трансляции и подъёма машинного кода в высокоуровневые представления либо вовсе не поддерживают архитектуру RISC-V, либо поддерживают малую её часть, а поддержка новых расширений в них затруднена.

В связи с этим был разработана универсальная модель инструкции RISC-V, основанная на семантическом переводе инструкций LLVM MIR в функции LLVM IR. В предложенном формате было описано необходимой подмножество инструкций этой архитектуры. На базе данной модели инструкций был разработан новый инструмент подъёма RISC-V кода в LLVM IR, названный `llvm-bleach`, способоный к статической бинарной трансляции входного машинного кода на любую поддержанную компилятором `clang` архитектуру.

`llvm-bleach` был использован для тестирования корректности оптимизаций в компиляторе `clang`. Данный эксперимент позволил обнаружить ошибку в виде некорректной работы инструкций выставления режима округления при оптимизации кода для архитектуры RISC-V.

Дальнейшими улучшениями могут являться:

- Использование предложенного подхода для динамической бинарной трансляции
- Генерация описания инструкций по предложенной модели из формального описания спецификаци.
- Использование поднятого LLVM IR кода для анализа программ без доступа к исходному коду.

Список Литературы

[Wat+13] A. Waterman, Y. Lee, R. Avizienis, H. Cook, D. Patterson, и K. Asanovic, «The RISC-V instruction set», 2013 г. doi: 10.1109/HOTCHIPS.2013.7478332.

[CLZ86] R. Cytron, A. Lowry, и F. K. Zadeck, «"Code motion of control structures in high-level languages" in Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages
», 1986 г. doi: 10.1145/512644.512651.

[RWZ88] B. K. Rosen, M. N. Wegman, и F. K. Zadeck, «"Global value numbers and redundant computations" in Proceedings of the 15th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages
», 1988 г. doi: 10.1145/73560.73562.

[LA04] C. Lattner и V. Adve, «"LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation" in Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization
», 2004 г. doi: 10.5555/977395.977673.

[BA24] К. Владимиров и И. Андреев, «Эффективное построение переупорядочиваний множества операций с памятью в многопоточной программе», 2024 г., *Международный научный журнал "Современные информационные технологии и ИТ-образование"*. doi: 10.25559/SITITO.020.202401.149-156.

[Roh19] R. Rohleder, «Hands-On Ghidra - A Tutorial about the Software Reverse Engineering Framework. In Proceedings of the 3rd ACM Workshop on Software Protection
», 2019 г., *Association for Computing Machinery, London, UK.* doi: 10.1145/3338503.3357725.

[Ros25] «Rosetta». Просмотрено: 14 июнь 2025 г. [Онлайн]. Доступно на: https://web.archive.org/web/20060113055505/http://www.apple.com/rosetta/

[Bar+03] L. Baraz *и др.*, «IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium®-based systems. In Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture
», 2003 г.

[Ber25] «Berberis». Просмотрено: 14 июнь 2025 г. [Онлайн]. Доступно на: https://android.googlesource.com/platform/frameworks/libs/binary_translation/

[Bel05] F. Bellard, «QEMU, a fast and portable dynamic translator. In Proceedings of the USENIX Annual Technical Conference», 2005 г. doi: 10.5555/1247360.1247401.

[YS19] S. B. Yadavalli и A. Smith, «Raising binaries to LLVM IR with MCTOLL (WIP paper). In proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems», 2019 г. doi: 10.1145/3316482.3326354.

[Fra25] «Framework for lifting x86, amd64, aarch64, sparc32, and sparc64 program binaries to LLVM bitcode». Просмотрено: 15 июнь 2025 г. [Онлайн]. Доступно на: https://github.com/lifting-bits/mcsema

[ES20] A. Engelke и M. Schulz, «Instrew: Leveraging LLVM for High Performance Dynamic Binary Instrumentation. In proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments
», март 2020 г. doi: 10.1145/3381052.3381319.

[Che+25] C. Chen, S. Sugita, Y. Nada, H. Irie, S. Sakai, и R. Shioya, «Biotite: A High-Performance Static Binary Translator using Source-Level Information. In proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction
», 2025 г.

[PB25] А. Романов и К. Владимиров, *Гибкий подход к подъёму LLVM MIR кода в SSA форму LLVM IR.* "Труды 67-й Всероссийской научной конференции МФТИ, Радиотехника и компьютерные технологии.", 2025, сс. 70–71.

[LLV25] «LLVM Language Reference Manual». Просмотрено: 18 июнь 2025 г. [Онлайн]. Доступно на: https://llvm.org/docs/LangRef.html

[llv25] «llvm-bleach». Просмотрено: 17 июнь 2025 г. [Онлайн]. Доступно на: https://github.com/ajlekcahdp4/llvm-bleach

[YAM25] «YAML Ain't Markup Language». Просмотрено: 17 июнь 2025 г. [Онлайн]. Доступно на: https://yaml.org/spec/1.2.2/

[Spi25] «Spike RISC-V ISA Simulator». Просмотрено: 18 июнь 2025 г. [Онлайн]. Доступно на: https://github.com/riscv-software-src/riscv-isa-sim

[Mac25] «MachineLICM incorrectly hoists fsrmi out of loop». Просмотрено: 18 июнь 2025 г. [Онлайн]. Доступно на: https://github.com/llvm/llvm-project/issues/135172