

Санкт-Петербургский государственный университет

Хечнев Семен Евгеньевич

Выпускная квалификационная работа

Валидация спецификации RISC-V ISA в LLVM относительно Sail

Уровень образования: бакалавриат

Направление *02.03.03 «Математическое обеспечение и администрирование
информационных систем»*

Основная образовательная программа *СВ.5162.2021 «Технологии программирования»*

Научный руководитель:
доцент кафедры системного программирования, к. ф.-м. н., Луцев Д. В.

Консультант:
инженер-исследователь, Лаборатория технологий программирования инфраструктурных
решений СПбГУ, Косарев Д. С.

Рецензент:
инженер-исследователь, Лаборатория технологий программирования инфраструктурных
решений СПбГУ, Кутуев В. А.

Санкт-Петербург
2025

Saint Petersburg State University

Semyon Khechnev

Bachelor's Thesis

Validation of the RISC-V ISA specification in LLVM against Sail

Education level: bachelor

Speciality *02.03.03 "Software and Administration of Information Systems"*

Programme *CB.5162.2021 "Programming Technologies"*

Scientific supervisor:
C.Sc., docent D.V. Luciv

Consultant:
Research Engineer at "Laboratory of Programming Technologies for Infrastructure Solutions at
St. Petersburg State University" D.S. Kosarev

Reviewer:
Research Engineer at "Laboratory of Programming Technologies for Infrastructure Solutions at
St. Petersburg State University" V.A. Kutuev

Saint Petersburg
2025

Оглавление

Введение	4
1. Постановка задачи	5
2. Обзор	6
2.1. RISC-V	6
2.2. Sail	6
2.3. LLVM TableGen	9
3. Особенности реализации	11
3.1. Архитектура	11
3.2. Специализация	12
3.3. Поиск входных и выходных операндов	15
3.4. Анализ обращения инструкции к памяти	16
3.5. Особенности расширения Р	17
3.6. Тесты и CI	20
4. Найденные расхождения	21
Заключение	22
Список литературы	23

Введение

Архитектура набора команд (ISA) RISC-V [8] на сегодняшний день является одной из наиболее активно развивающихся открытых архитектур. Её открытость, модульность и поддержка со стороны научного и индустриального сообщества способствуют её распространению в широком спектре систем — от встраиваемых платформ до высокопроизводительных процессоров.

LLVM [7] — это инфраструктура для компиляторов, которая поддерживает множество целевых платформ, в том числе и RISC-V. В LLVM спецификация ISA целевой платформы описывается декларативно, с помощью предметно-ориентированного языка TableGen [6]. Параллельно существует одобренный сообществом RISC-V International эмулятор архитектуры RISC-V, `sail-riscv`¹, написанный на языке Sail [10]. Данный эмулятор определяет строгую семантику исполнения инструкций и используется для верификации [4], моделирования и формального анализа [2]. Таким образом, одновременно существуют две ключевые спецификации RISC-V ISA, каждая из которых описывает одни и те же инструкции, но с разных позиций.

Несмотря на наличие двух спецификаций RISC-V ISA, в настоящее время не существует инструментов, позволяющих систематически сопоставлять эти спецификации и проверять их согласованность. Это создаёт риск накопления расхождений между спецификациями. Такие расхождения могут затрагивать ограничения на операнды, семантические свойства или поведение в особых ситуациях. Наличие подобных расхождений может приводить к некорректной генерации кода компилятором, расхождению поведения аппаратных реализаций и снижению общей надёжности экосистемы RISC-V.

Настоящая работа посвящена валидации спецификации RISC-V ISA в LLVM относительно её формальной модели на Sail. В работе предлагается разработать статический анализатор, который будет искать несогласованности спецификаций.

¹<https://github.com/riscv/sail-riscv> (дата обращения: 17 мая 2025 г.).

1. Постановка задачи

Целью работы является валидация спецификации RISC-V ISA в LLVM относительно Sail. Для её выполнения были поставлены следующие задачи.

1. Разработать прототип статического анализатора, который будет включать следующие анализы:
 - поиск входных и выходных операндов инструкций;
 - определение, может ли инструкция читать или писать память.
2. С помощью статического анализатора провалидировать спецификации RISC-V ISA:
 - в LLVM 19 против sail-riscv²;
 - расширения P³ в LLVM против sail-riscv-p⁴.

²<https://github.com/riscv/sail-riscv/tree/05b845c91d1c1db7b361fc8d06e815b54ca0b07a> (дата обращения: 17 мая 2025 г.).

³Version 0.9.11-draft-20211209

⁴<https://github.com/umcann123/sail-riscv-p> (дата обращения: 17 мая 2025 г.).

2. Обзор

В настоящем разделе будет представлен обзор предметной области.

2.1. RISC-V

RISC-V — свободная и открытая архитектура процессоров, впервые представленная в 2010 году в университете Беркли, США [11]. Создана по концепции RISC (Reduced Instruction Set Computer) [5], то есть обладает довольно скромным набором инструкций, но с возможностью дополнения при помощи расширений.

2.2. Sail

Эмулятор архитектуры — это инструмент, который позволяет исполнять программу для одной архитектуры на процессоре другой (или той же) архитектуры. Sail — это язык для определения эмуляторов процессорных архитектур. На языке Sail реализованы модели множества архитектур, включая X86⁵, ARM⁶ [3], RISC-V⁷ [3] и другие. По моделям на данном языке Sail умеет генерировать эмулятор на OCaml или C. Кроме генерации эмуляторов по моделям, Sail также позволяет генерировать и исполнять автоматические тесты, исполнять тесты безопасности кода, а также верифицировать модели ISA.

2.2.1. Модель sail-riscv

Sail-riscv — это формальная модель RISC-V ISA, реализованная на языке Sail. Модель представляет из себя интерпретатор инструкций RISC-V ISA.

Во-первых, в модели для удобного внутреннего представления инструкций определяется дерево абстрактного синтаксиса (Abstract Syntax Tree). Причём один узел AST может представлять множество

⁵<https://github.com/rem-s-project/sail-x86-from-acl2> (дата обращения: 17 февраля 2025 г.).

⁶<https://github.com/rem-s-project/sail-arm> (дата обращения: 17 февраля 2025 г.).

⁷<https://github.com/riscv/sail-riscv> (дата обращения: 17 февраля 2025 г.).

инструкций. Например, на листинге 1 узел `ITYPE`, который параметризован 12-ти битным непосредственным (`immediate`) значением, двумя индексами регистров (два битовых вектора длины 5) и интерпретируемой операцией, представляет инструкции `addi`, `slti`, `sltiu`, `xori`, `ori` и `andi`.

Листинг 1: Пример определения узла `AST` в `sail-riscv`

```
enum iop = {RISCV_ADDI, RISCV_SLTI, RISCV_SLTIU,  
            RISCV_XORI, RISCV_ORI, RISCV_ANDI}  
  
union clause ast = ITYPE : (bits(12), regidx, regidx, iop)
```

Затем для каждого узла `AST` определяются `encdec` отображения, которые предназначены для декодирования 32-битного кода инструкции (`opcode`) в параметризованный узел `AST` модели и, наоборот, кодирования параметризованного узла `AST` в код инструкции.

Листинг 2: Пример определения функции кодирования и декодирования инструкций в `sail-riscv`

```
mapping encdec_iop : iop <=> bits(3) = {  
    RISCV_ADDI  <=> 0b000,  
    RISCV_SLTI  <=> 0b010,  
    RISCV_SLTIU <=> 0b011,  
    RISCV_ANDI  <=> 0b111,  
    RISCV_ORI   <=> 0b110,  
    RISCV_XORI  <=> 0b100  
}  
  
mapping clause encdec = ITYPE(imm, rs1, rd, op)  
    <=> imm @ rs1 @ encdec_iop(op) @ rd @ 0b0010011
```

На листинге 2 представлено определение такого отображения для узла `ITYPE`. Из определения этого отображения следует, что первые 12 бит кода инструкции кодируют непосредственное значение, следующие 5 бит кодируют входной регистр, следующие 3 бита кодируют целочис-

ленную операцию, которую будет представлять узел `ITYPE`, следующие 5 бит кодируют выходной регистр.

Код, который отвечает за определение семантики исполнения инструкции, содержится в `execute` функциях.

Листинг 3: Пример определения семантики исполнения инструкций `addi`, `slti`, `sltiu`, `xori`, `ori` и `andi` в `sail-riscv`

```
function clause execute (ITYPE (imm, rs1, rd, op)) = {
  let rs1_val = X(rs1);           // read the rs1
  let immext : xlenbits = EXTS(imm); // sign-extend imm
  let result : xlenbits = match op { // compute the result
    RISCV_ADDI => rs1_val + immext, // for addi, do addition
    RISCV_SLTI => EXTZ(rs1_val <_s immext), // ...etc
    RISCV_SLTIU => EXTZ(rs1_val <_u immext),
    RISCV_ANDI => rs1_val & immext,
    RISCV_ORI  => rs1_val | immext,
    RISCV_XORI => rs1_val ^ immext
  };
  X(rd) = result;                // write result to rd
  RETIRE_SUCCESS                  // successful termination
}
```

На листинге 3 определена функция, которая задаёт семантику исполнения для узла `ITYPE`. Семантика следующая, во-первых, считывается значение из входного регистра `rs1`, затем знаково расширяется непосредственное значение до ширины равной `XLEN` (текущей разрядности RISC-V), далее в зависимости от параметра `op`, представляющего целочисленную операцию, вычисляется результат, в конце вычисленный результат записывается в выходной регистр `rd`.

Наконец, для каждого узла `AST` определяются функции `assembly`, отображающие параметризованные узлы `AST` в мнемоники. На листинге 4 определена такая функция для узла `ITYPE`. Из определения видно, что узел `ITYPE` отображается в зависимости от параметра `op` в конкретную целочисленную инструкцию с непосредственным операндом.

Листинг 4: Пример определения функции отображения параметризованного узла AST в мнемонику в sail-riscv

```
mapping itype_mnemonic : iop <=> string = {
  RISCV_ADDI  <=> "addi",
  RISCV_SLTI  <=> "slti",
  RISCV_SLTIU <=> "sltiu",
  RISCV_XORI  <=> "xori",
  RISCV_ORI   <=> "ori",
  RISCV_ANDI  <=> "andi"
}

mapping clause assembly = ITYPE(imm, rs1, rd, op)
  <=> itype_mnemonic(op) ^ spc() ^ reg_name(rd) ^ sep()
    ^ reg_name(rs1) ^ sep() ^ hex_bits_signed_12(imm)
```

2.3. LLVM TableGen

LLVM [7] — это инфраструктура для разработки компиляторов, созданная в 2003 году в Иллинойском университете, США. LLVM поддерживает широкий спектр целевых платформ, включая RISC-V. Важную роль для бэкендов LLVM играет TableGen.

TableGen — это декларативный язык, который используется в LLVM для описания предметно-ориентированной информации. В частности, TableGen используется для описания спецификации ISA целевой платформы. Информация о спецификации ISA целевой платформы из TableGen файлов используется в бэкендах LLVM. Файлы TableGen имеют расширение .td. При помощи утилиты `llvm-tblgen` можно скомпилировать .td файлы в JSON.

Листинг 5: Описание инструкции addi при помощи TableGen

```
let hasSideEffects = 0, mayLoad = 0, mayStore = 0 in
class ALU_ri<bits<3> funct3, string opcodestr>
  : RVInstI<funct3, OPC_OP_IMM, (outs GPR:$rd),
```

```
(ins GPR:$rs1, simm12:$imm12),  
opcodestr, "$rd, $rs1, $imm12">,  
Sched<[WriteIALU, ReadIALU]>;  
  
let isReMaterializable = 1, isAsCheapAsAMove = 1 in  
def ADDI : ALU_ri<0b000, "addi">;
```

В описании инструкции на TableGen постулируются следующие свойства:

- входные и выходные операнды, т. е. операнды, из которых считываются и в которые записываются значения;
- может ли инструкция читать или писать память;
- коммутативность;
- сложность;
- и множество других свойств.

Например, на листинге 5 представлено описание инструкции `addi`. В описании инструкции говорится, например, что первый операнд (`rd` — register destination) является выходным, а два остальных (`rs1` — register source и `imm` — непосредственное значение) входными, инструкция не читает (`mayLoad = 0`) и не пишет (`mayStore = 0`) память, инструкция не имеет побочных эффектов.

3. Особенности реализации

В настоящем разделе будут описаны ключевые особенности реализации статического анализатора.

3.1. Архитектура

Статический анализатор разделён на два модуля. Первый модуль отвечает за анализ декларативной спецификации в LLVM, второй ответственен за анализ модели на Sail.

- Модуль, ответственный за анализ LLVM, получает на вход скомпилированные в единый JSON .td файлы, содержащие описание спецификации RISC-V ISA. Затем из входного JSON файла для каждой инструкции извлекаются следующие свойства:
 - мнемоника (извлекается по ключу `AsmString`);
 - входные операнды (по ключу `InOperandList`);
 - выходные операнды (по ключу `OutOperandList`);
 - неявные входные операнды (по ключу `Uses`);
 - неявные выходные операнды (по ключу `Defs`);
 - может ли инструкция читать память (по ключу `mayLoad`);
 - может ли инструкция писать в память (по ключу `mayStore`);
 - для какой разрядности (RV32 или RV64, или для обеих) определена инструкция (по ключу `Predicates`).

Наконец, анализатор генерирует модуль, который содержит всю собранную информацию о свойствах каждой инструкции и предоставляет интерфейс для доступа к информации. Генерация отдельного модуля со всей информацией реализована для того, чтобы при каждом запуске анализатора не анализировать один и тот же JSON, тем самым экономя драгоценное время.

- Второй модуль, отвечающий за статический анализ модели на Sail, получает на вход `.sail` файлы, которые содержат определение модели. Далее при помощи библиотеки компилятора языка Sail производится синтаксический анализ и вывод типов, на выходе получая типизированное AST. Затем на полученном ранее типизированном AST запускаются анализы, о которых подробнее далее.
- В итоге полученная из модулей информация сравнивается, и генерируется отчёт, которые содержит все найденные расхождения.

3.2. Специализация

Как было сказано ранее, эмулятор `sail-riscv` представляет из себя интерпретатор некоторого AST, каждый узел которого параметризован некоторыми параметрами. Один узел AST модели может представлять сразу несколько инструкций, но специализация узла представляет из себя конкретную инструкцию.

Например, узел AST из листинга 1 специализированный по последнему параметру

```
ITYPE (imm, rs1, rd, RISCV_ADDI)
```

представляет инструкцию

```
addi rd, rs1, imm
```

3.2.1. Сопоставление узлов AST и мнемоник

Во-первых, необходимо было определить за какие инструкции отвечает каждый узел AST модели `sail-riscv`. Для этого производился анализ специализаций функций `assembly`, которые как раз отвечают за отображение узлов AST в мнемоники инструкций.

3.2.2. Специализация `execute` функций

Узел `ITYPE`, определенный на листинге 1, параметризован четырьмя параметрами: непосредственным значением ширины 12 бит, двумя

индексами регистров и целочисленной операцией, которая представлена типом перечисления (enum). В зависимости от значения последнего параметра по-разному вычисляется результат в `execute` функции, задающей семантику исполнения узла `ITYPE`, на листинге 3. При этом в общем случае в зависимости от параметров, семантика исполнения может кардинально изменяться, например, при одном значении параметра может происходить чтение какого-то регистра, а при другом не происходить; при одном значении может читаться память, а при другом происходить запись в память. Поэтому важно специализировать.

Анализатор специализирует `execute` функции только по параметрам типов перечисления (enum) и логических типов (bool).

Например, для специализации `ITYPE (imm, rs1, rd, RISCV_ADDI)` (специализирован только последний параметр) тело, соответствующей `execute` функции, определённой на листинге 3, после специализации выглядело следующим образом.

Листинг 6: Пример специализированной по параметру `execute` функции из листинга 3

```
function clause execute (ITYPE (imm, rs1, rd, RISCV_ADDI)) = {
  let rs1_val = X(rs1);           // read the rs1
  let immext : xlenbits = EXTS(imm); // sign-extend imm
  let result : xlenbits = rs1_val + immext;
  X(rd) = result;                 // write result to rd
  RETIRE_SUCCESS                  // successful termination
}
```

Однако после описанной выше специализации в теле функции могут остаться выражения, результат которых нигде не используется. При этом такие выражения могут негативно влиять на анализ. Например.

Листинг 7: Пример функции, после специализации которой могут остаться неиспользуемые выражения, негативно влияющие на дальнейший анализ

```
1 function clause execute (PEXT_32_16_MUL_ADD(rs2, rs1, rd, op)) = {
2   let rs1_val = X(rs1); let rs2_val = X(rs2); let rd_val = X(rd);
```

```

3  result : xlenbits = zeros();
4  foreach (i from 0 to (sizeof(xlen) - 32) by 32) {
5      let rs1_word = slice(rs1_val, i, 32);
6      let rs2_word = slice(rs2_val, i, 32);
7      let rd_word = slice(rd_val, i, 32);
8      result[(i+31)..i] = match op {
9          SMMWB => smmw(rs1_word, slice(rs2_word, 0, 16)),
10         KMMAWB => kmmaw(rs1_word, slice(rs2_word, 0, 16), rd_word ),
11         ...
12     };
13 };
14 X(rd) = result;
15 RETIRE_SUCCESS
16 }

```

Листинг 8: Вспомогательные выражения

```

1  let rd_val = X(rd);
2  result[(i+31)..i] = smmw(rs1_word, slice(rs2_word, 0, 16));

```

Тело функции из листинга 7 сокращено для краткости. Будем считать, что после 10 строчки ни один идентификатор, определённый ранее, не используется.

В функции листинга 7 параметр `op` имеет тип перечисления со значениями `SMMWB`, `KMMAWB` ... Если специализировать тело этой функции по параметру `op` со значением `SMMWB`, то выражение на 8 строчке сократится до выражения, представленного на строчке 2 листинга 8. Но после такой специализации связка (binding) `rd_word`, определенная на строчке 7, нигде больше не используется. Поэтому её определение можно удалить из специализированной версии функции. При этом после такого удаления становится не используемой связка `rd_val`, поэтому выражение (в котором происходит чтение регистра!) определения этой связки (строчка 1 листинга 8) можно удалить из 2 строчки листинга 7.

Таким образом, если не производить подобный проход, то для ин-

струкции SMMWB и многих других было бы найдено ненужное чтение из регистра `rd`.

3.3. Поиск входных и выходных операндов

В модели `sail-riscv` для взаимодействия с регистрами используются следующие функции:

- `X` для чтения или записи регистров общего назначения (General-Purpose Registers);
- `F` для чтения или записи вещественных регистров;
- `V` для чтения или записи векторных регистров.

Причем если вызов одной из выше определенных функций происходит справа от присваивания, то выбирается перегрузка для чтения регистра, если слева, то для записи.

Рассмотрим следующую функцию.

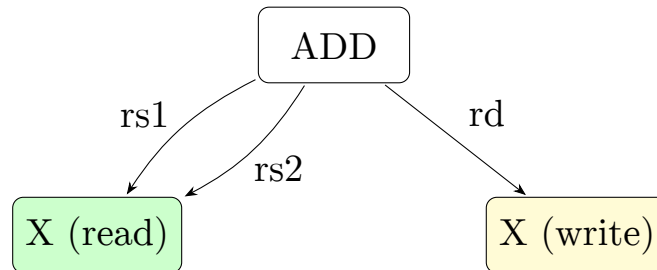
Листинг 9: Пример определения функции, задающей семантику исполнения инструкции `add`

```
function clause execute ( ADD(rs2, rs1, rd) ) = {  
    let rs1_val = X(rs1) ;  
    let rs2_val = X(rs2) ;  
    let result : xlenbits = rs1_val + rs2_val;  
    X(rd) = result  
    RETIRE_SUCCESS  
}
```

Функция на листинге 9 определяет семантику исполнения для инструкции `add`. Семантика следующая. Во-первых, при помощи функции `X` считываются значение входных регистров `rs1` и `rs2`. Затем вычисляется результат, и результат записывается в регистр `rd`, используя перегрузку функции `X` для записи.

Для поиска входных и выходных регистров строился граф вызовов функций [9]. Затем производился обход графа вызовов и анализ того, какие функции, с какими аргументами вызывали функции для взаимодействия с регистрами.

Рис. 1: Граф вызовов для функции из листинга 9



Из графа на рисунке 1 видно, что функция, задающая семантику для исполнения инструкции `add`, вызывает `X` для чтения регистров `rs1` и `rs2`, а также вызывает `X` для записи значения в регистр `rd`. Таким образом, для инструкции `add` регистры `rs1` и `rs2` являются входными, а регистр `rd` выходным.

3.4. Анализ обращения инструкции к памяти

Для работы с памятью в `sail-riscv` используются функции `read_ram` и `write_ram`. Для того чтобы определить читает или пишет инструкция память, для каждой специализации функции `execute` проверялась достижимость вершин `read_ram` и `write_ram` в ранее определённом графе вызовов функций.

Однако любая инструкция, которая работает с адресами, при трансляции виртуального адреса в физический может как писать, так и читать память. Это происходит из-за взаимодействия с таблицей страниц (`page table`).

- Чтение из памяти может потребоваться, когда при трансляции виртуального адреса в физический в TLB (Translation Lookaside Buffer — специализированный кэш для ускорения трансляции адресов, хранящий информацию о недавних трансляциях) отсут-

ствуется необходимая запись. В этом случае требуется обход таблицы страниц, которая хранится в памяти.

- Запись в память может потребоваться, когда при трансляции виртуального адреса в физический необходимо обновить бит доступа (A — accessed) или модификации (D — dirty) у записи таблицы страниц.

Обращения к памяти при трансляции виртуальных адресов в физические игнорировались. Для этого при решении задачи достижимости вершин `read_ream` и `write_ram` в графе вызовов игнорировался вызов функции `translateAddr`, которая ответственна за трансляцию виртуального адреса в физический.

3.5. Особенности расширения P

RISC-V — это модульная ISA, которую можно расширять при помощи расширений. Расширение P (packed SIMD) — это расширение, предназначенное для ускорения целочисленных вычислений и вычислений с фиксированной точкой.

Проект `sail-riscv-p`⁸ — это форк `sail-riscv` с реализованным расширением P. В этом расширении существуют инструкции, у которых один операнд может представлять сразу пару регистров.

Листинг 10: Пример `execute` функции, задающей семантику исполнения инструкциям, которые работают с парами регистров

```
function clause execute(PEXT_64BITS_ADD_SUB(rs2, rs1, rd, op)) =
{
  results : bits(64) = zeros();
  rs1_val : bits(64) = zeros();
  rs2_val : bits(64) = zeros();
  if sizeof(xlen) == 32
  then {
```

⁸<https://github.com/umcann123/sail-riscv-p> (дата обращения: 17 февраля 2025 г.).

```

    rs1_val = X(rs1 | 0b00001) @ X(rs1 & 0b11110);
    rs2_val = X(rs2 | 0b00001) @ X(rs2 & 0b11110);
}
else { ... };
results = match op {
    ADD64 => rs1_val + rs2_val,
    ...
};
if sizeof(xlen) == 32
then {
    X(rd | 0b00001) = results[63..32];
    X(rd & 0b11110) = results[31..0];
}
else { ... };
RETIRE_SUCCESS
}

```

Нам интересно определять тип операндов (представляют ли они пару или нет), потому что он потенциально может быть неправильно определен в спецификации в LLVM.

Листинг 11: Пример выражения, в котором происходит чтение и конкатенация значений из парных регистров

`X(rs1 | 0b00001) @ X(rs1 & 0b11110)`

Для анализа типа операнда использовался эвристический алгоритм. Рассмотрим алгоритм на примере выражения из листинга 11.

1. Представление программы в ANF [12] (A-Normal Form — форма представления программы, при которой все аргументы функций тривиальны, т. е. константы или переменные).

Листинг 12: ANF выражения из листинга 11

```

let a1 = rs1 | 0b00001 in
let a2 = rs1 & 0b11110 in
X(a1) @ X(a2)

```

2. Определение входных и выходных регистров по алгоритму описанному в 3.3. Для выражения из листинга 12 регистры `a1` и `a2` являются входными.
3. Построение и анализ data flow графа [1]

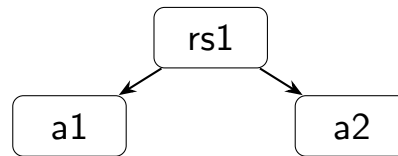


Рис. 2: Data flow граф для выражения из листинга 12

Из графа определяем, что от регистра `rs1` зависит два входных регистра `a1` и `a2`. Делаем вывод, что регистр `rs1` представляет пару и является входным.

3.5.1. Анализ чтений и записей контрольно-статусных регистров

В Sail для определения регистра существует ключевое слово `register`. Регистры представляют из себя глобальные переменные некоторого типа, с которыми можно работать как с обычной переменной.

Листинг 13: Пример определения регистра в sail-riscv

```
register marchid : xlenbits
```

Для расширения `P` необходимо было дополнительно определить для каждой инструкции какие CSRs (Control/Status Registers) она может читать и писать.

Для того чтобы определить обращается ли какая-то специализация `execute` функции (другими словами инструкция) к контрольно-статусным регистрам, производился рекурсивный обход всех вызываемых ею функций. В процессе обхода отслеживались присваивания переменным. Если переменная представляла контрольно-статусный регистр, значит происходила запись в регистр, и такое обращение запоминалось. Аналогично, отслеживались использования переменных в выражениях.

Если используемая переменная — контрольно-статусный регистр, значит происходило чтение этого регистра, и такое обращение запомнилось.

3.6. Тесты и CI

Для статического анализатора были написаны модульные и интеграционные тесты. Тестовое покрытие анализатора составляет около 81%.

Для непрерывной интеграции при помощи `GitHub Actions` был реализован CI, в котором происходит проверка сборки проекта, выполнения тестов и генерация отчёта о покрытии кода тестами.

Кроме этого, для того чтобы ускорить выполнение CI, был собран `docker` образ⁹, в котором предустановлены все зависимости для сборки проекта и запуска тестов.

⁹<https://hub.docker.com/r/skhechnev/ocaml-5.1.0-sail> (дата обращения: 17 мая 2025 г.).

4. Найденные расхождения

Расхождения в LLVM 19 и sail-riscv¹⁰.

1. Регистр SP (stack pointer) не является явным аргументом в sail-riscv для инструкций: `c.addi16sp`, `c.addi4spn`, `c.fldsp`, `c.flwsp`, `c.fsdsp`, `c.fswsp`, `c.ldsp`, `c.lwsp`, `c.sdsp`, `c.swsp`.

Инструкции имеют разный синтаксис.

2. Для расширения векторных инструкций было найдено два типа расхождений:
 - для 28 инструкций в LLVM не указывается, что регистр `vd` является входным;
 - для 255 инструкций в LLVM не указывается, что регистры `v0` и `vd` являются входными.

В sail-riscv регистр `vd` является входным, значение этого регистра используется для составления маски, от которой зависит результат инструкции.

Расхождения в расширении P в LLVM и sail-riscv-p¹¹.

1. В sail-riscv-p не хватает инструкции `rev`, в LLVM — `pktt32`, `clo32`, `clo8`, `clo16`, `pkbb32`.
2. В LLVM указано для 22 инструкций, что они читают контрольно-статусный регистр `vxsat` (используется как флаг переполнения). В sail-riscv-p 124 инструкции только пишут в регистр `vxsat`. Других обращений к контрольно-статусным регистрам не найдено.

¹⁰<https://github.com/riscv/sail-riscv/commits/05b845c91d1c1db7b361fc8d06e815b54ca0b07a> (дата обращения: 17 мая 2025 г.).

¹¹<https://github.com/umcann123/sail-riscv-p> (дата обращения: 17 апреля 2025 г.).

Полный список расхождений — https://github.com/Kakadu/llvm_vs_sail/blob/master/Results.md (дата обращения: 25 апреля 2025 г.).

Заключение

Таким образом, в рамках выпускной квалификационной работы были достигнуты следующие результаты.

- Разработан прототип статического анализатора, который реализует следующие анализы;
 - поиск входных и выходных операндов инструкций;
 - определение, может ли инструкция читать или писать память.
- С помощью статического анализатора провалидированы спецификации RISC-V ISA:
 - в LLVM 19 против sail-riscv;
 - расширения P в LLVM против sail-riscv-p.

Исходный код анализатора: https://github.com/Kakadu/llvm_vs_sail.

Имя пользователя: s-khechnev.

Список литературы

- [1] Allen Frances E., Cocke John. A program data flow analysis procedure // Communications of the ACM. — 1976. — Vol. 19, no. 3. — P. 137.
- [2] Detailed models of instruction set architectures: From pseudocode to formal semantics / Alasdair Armstrong, Thomas Bauereiss, Brian Campbell et al. // Proceedings of the 25th Automated Reasoning Workshop: Bridging the Gap between Theory and Practice: ARW. — 2018. — P. 23–24.
- [3] ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS / Alasdair Armstrong, Thomas Bauereiss, Brian Campbell et al. // Proceedings of the ACM on Programming Languages. — 2019. — Vol. 3, no. POPL. — P. 1–31.
- [4] Islaris: verification of machine code against authoritative ISA semantics / Michael Sammler, Angus Hammond, Rodolphe Lepigre et al. // Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. — 2022. — P. 825–840.
- [5] Jamil Tariq. Risc versus cisc // Ieee Potentials. — 1995. — Vol. 14, no. 3. — P. 13–16.
- [6] LLVM TableGen. — URL: <https://llvm.org/docs/TableGen/> (дата обращения: 16 декабря 2024 г.).
- [7] Lattner Chris, Adve Vikram. LLVM: A compilation framework for life-long program analysis & transformation // International symposium on code generation and optimization, 2004. CGO 2004. / IEEE. — 2004. — P. 75–86.
- [8] The RISC-V instruction set manual, volume I: User-level ISA, version 2.0 / Andrew Waterman, Yunsup Lee, David A Patterson,

Krste Asanovic // EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-54. — 2014. — P. 4.

- [9] Ryder Barbara G. Constructing the call graph of a program // IEEE Transactions on Software Engineering. — 1979. — no. 3. — P. 216–226.
- [10] The Sail instruction-set semantics specification language : Rep. / Technical report published by Cambridge University ; Executor: Kathryn E Gray, Peter Sewell, Christopher Pulte et al. : 2017.
- [11] Waterman Andrew Shell. Design of the RISC-V instruction set architecture. — University of California, Berkeley, 2016.
- [12] The essence of compiling with continuations / Cormac Flanagan, Amr Sabry, Bruce F Duba, Matthias Felleisen // ACM Sigplan Notices. — 1993. — Vol. 28, no. 6. — P. 237–247.