Санкт-Петербургский государственный университет

Кафедра системного программирования

Валидация спецификации RISC-V ISA в LLVM относительно Sail

Хечнев Семен Евгеньевич

Научный руководитель: доцент кафедры системного программирования, к. ф.-м. н., Луцив 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

Введение

- RISC-V — открытая, расширяемая ISA (Instruction Set Architecture)
- Sail[1] — язык для описания моделей процессорных архитектур. Позволяет задать семантику исполнения процессорных инструкций
  - Sail-riscv[2] — формальная, поведенческая спецификация RISC-V ISA
- LLVM[3] — инфраструктура для компиляторов
  - Спецификация целевой ISA описывается декларативно на TableGen
- Актуальна проблема валидации спецификаций относительно друг друга

[1] https://github.com/rems-project/sail/
[2] https://github.com/riscv/sail-riscv
[3] https://llvm.org

Постановка задачи

Целью работы является валидация спецификации RISC-V ISA в LLVM относительно Sail

Задачи:

- Разработать прототип статического анализатора, включающий следующие анализы
  - Поиск входных и выходных операндов инструкций
  - Определение, может ли инструкция писать или читать память
- Используя статический анализатор, провалидировать спецификации RISC-V ISA
  - В LLVM v19 против sail-riscv[4]
  - Расширения P (version 0.9.11-draft-20211209) в LLVM против sail-riscv-p[5]

[4] https://github.com/riscv/sail-riscv/tree/05b845c91d1c1db7b361fc8d06e815b54ca0b07a
[5] https://github.com/umcann123/sail-riscv-p

Обзор. Описание спецификации в LLVM

- TableGen[6] — декларативный язык, который используется в LLVM для описания предметно-ориентированной информации
- В частности для описания спецификации ISA целевых платформ
- Для каждой инструкции описываются следующие свойства
  - Мнемоника
  - Код инструкции
  - **Входные**, **выходные операнды**
  - Может ли **читать из памяти**, **писать в память**
  - Коммутативность
  - ...
- Файлы с TableGen кодом имеют расширение `.td`
- При помощи утилиты `llvm-tblgen` можно скомпилировать в `JSON`

[6] `https://llvm.org/docs/TableGen/`

Обзор. Sail-riscv

- Sail-riscv[7] — это модель архитектуры RISC-V на Sail
- Модель представляет из себя реализацию интерпретатора инструкций RISC-V ISA
  - AST
  - encdec, кодирование/декодирование кодов инструкций (opcode) в AST
  - assembly, отображение AST в мнемоники
  - execute, семантика исполнения инструкций

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

[7] https://github.com/riscv/sail-riscv

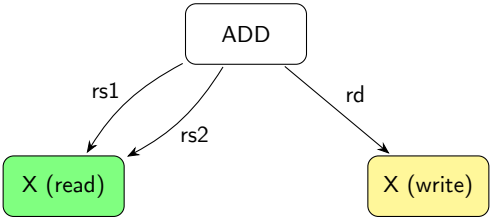
Реализация. Статический анализатор

- Язык реализации — OCaml
- **Sail**
  1. Входные данные — `sail` файлы модели
  2. Синтаксический анализ файлов при помощи библиотеки компилятора Sail
  3. Статический анализ
     - Определение мнемоник
     - Специализация `execute` функций
     - Анализ `execute` функций
- **LLVM TableGen**
  1. Входные данные — файлы спецификации на TableGen, скомпилированные[8] в `JSON`
  2. Из `JSON` извлекается
     - Входные и выходные операнды
     - Может ли инструкция читать/писать память
- Полученная информация сравнивается, и генерируется отчёт, содержащий расхождения

[8] при помощи `llvm-tblgen`

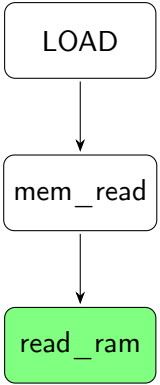
Реализация. Поиск входных и выходных операндов

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



Реализация. Обращение инструкции к памяти

- Функции **`read_ram`** и **`write_ram`** являются примитивными для работы с памятью
- Для каждой `execute` функции проверялась достижимость вершин **`read_ram`** и **`write_ram`** в графе вызовов функций



Реализация. Особенности sail-riscv-p (1)

- RISC-V — модульная архитектура, позволяющая расширять ISA при помощи расширений
- Расширение P предназначено для ускорения целочисленных и fixed point вычислений
- Sail-riscv-p[9] — это форк sail-riscv с реализацией расширения P
- Для 32-ух битного RISC-V операнды некоторых инструкций кодируют пару регистров

```
function clause execute(PEXT_64BITS_ADD_SUB(rs2, rs1, rd, op))
...
  rs1_val = X(rs1 | 0b00001) @ X(rs1 & 0b11110)
...
```

[9] https://github.com/umcann123/sail-riscv-p

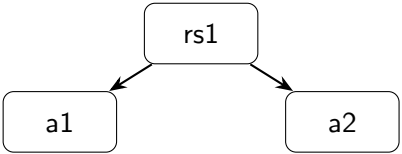
Реализация. Особенности sail-riscv-p (2)

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

1. Представление программы в ANF, A-Normal Form — все аргументы вызовов функций константы или переменные

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

2. Определение входных и выходных операндов по графу вызовов
3. Построение и анализ data flow графа



Реализация. CI и тесты

1. Для статического анализатора написаны модульные и интеграционные тесты
2. Тестовое покрытие составляет 81.76%
3. `CI` для проверки успешности сборки проекта и прохождения тестов

Найденные расхождения

- Расширение C
  - У 10 инструкций в sail-riscv регистр SP (stack pointer) не является явным аргументом
    - `c.addi16sp`, `c.addi4spn`, `c.fldsp`, `c.flwsp`, `c.fsdsp`, `c.fswsp`, `c.ldsp`, `c.lwsp`, `c.sdsp`, `c.swsp`
- Расширение V
  - У 430 инструкций в sail-riscv операнд `vd` является входным операндом
  - Расхождения возникают, потому что эти инструкции не используются напрямую в LLVM при генерации кода
  - В LLVM генерируются искуственные псевдоинструкции, которые содержат дополнительные свойства
- Расширение P
  - В LLVM 22 инструкции читают регистр `vxsat` (используется как флаг переполнения)
  - В sail-riscv-p 116 инструкций пишут в регистр `vxsat`

Результаты

Таким образом, в рамках ВКР были достигнуты следующие результаты[10]

- 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 v19 против sail-riscv
  - Расширения `P` (version 0.9.11-draft-20211209) в LLVM против sail-riscv-p

[10] Код — `https://github.com/Kakadu/llvm_vs_sail`. Имя пользователя: s-khechnev

Дополнительный слайд. TableGen

Листинг 1: Пример описания инструкции add на декларативном языке TableGen

```
let hasSideEffects = 0, mayLoad = 0, mayStore = 0 in
class ALU_rr<bits<7> funct7, bits<3> funct3, string opcodestr,
             bit Commutable = 0>
    : RVInstR<funct7, funct3, OPC_OP, (outs GPR:$rd),
                (ins GPR:$rs1, GPR:$rs2),
                opcodestr, "$rd, $rs1, $rs2"> {
  let isCommutable = Commutable;
}

def ADD : ALU_rr<0b0000000, 0b000, "add", Commutable=1>,
            ...
```

Дополнительный слайд. Инструкции RVV в LLVM TableGen

```
"VAADD_VV": {
  "AsmString": "vaadd.vv\t$vd, $vs2, $vs1, $vm",
  "InOperandList": { "args": [ [ { "def": "VR" }, "vs2" ],
                               [ { "def": "VR" }, "vs1" ],
                               [ { "def": "VMaskOp" }, "vm" ] ] },
  "OutOperandList": { "args": [[ { "def": "VR" }, "vd" ]]}
},

"PseudoVAADD_VV_MF8": {
  "AsmString": "", // empty
  "BaseInstr": { "def": "VAADD_VV" },
  "InOperandList": { "args": [[ { "def": "VR" }, "rd" ],
                               [ { "def": "VR" }, "rs2" ],
                               [ { "def": "VR" }, "rs1" ],
                               [ { "def": "ixlenimm" }, "rm" ],
                               [ { "def": "AVL" }, "vl" ],
                               [ { "def": "ixlenimm" }, "sew" ],
                               [ { "def": "ixlenimm" }, "policy" ] ] },
  "OutOperandList": { "args": [[ { "def": "VR" }, "rd" ]]},
}
```

Дополнительный слайд. Sail-riscv (1)

Листинг 2: Пример узла AST, представляющего инструкции addi, slti, sltiu, andi, ori и xori

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

Листинг 3: Функция кодирования/декодирования

```
mapping encdec_iop : iop <-> bits(3) = {
  RISCV_ADDI  <-> 0b000,
  RISCV_SLTI  <-> 0b010,
  ...
}
mapping clause encdec = ITYPE(imm, rs1, rd, op)
  <-> imm @ rs1 @ encdec_iop(op) @ rd @ 0b0010011
```

Дополнительный слайд. Sail-riscv (2)

Листинг 4: Пример функции assembly (отображает узлы AST в мнемоники) для узла с листинга 2

```
mapping itype_mnemonic : iop <-> string = {
  RISCV_ADDI  <-> "addi",
  RISCV_SLTI  <-> "slti",
  ...
}

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)
```

По assembly функциям определяются мнемоники инструкций, которые представляет узел AST

Дополнительный слайд. Sail-riscv (3)

Листинг 5: Пример определения семантики исполнения инструкций addi, slti, sltiu, andi, ori и xori. Анализатор специализирует тело функции по параметру op

```
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
    ...
  };
  X(rd) = result;                        // write result to rd
  RETIRE_SUCCESS                         // successful termination
}
```

Дополнительный слайд. Sail-riscv (4)

Листинг 6: Пример execute функции, которая вызывает другой execute

```
function clause execute (C_ADDI4SPN(rdc, nzimm)) = {
  let imm : bits(12) = (0b00 @ nzimm @ 0b00);
  let rd = creg2reg_idx(rdc);
  execute(ITYPE(imm, sp, rd, RISCV_ADDI))
}
```

Добавляет ненулевое непосредственное значение, расширенное нулями и умноженное на 4, к регистру SP (stack pointer) и записывает результат в rdc.

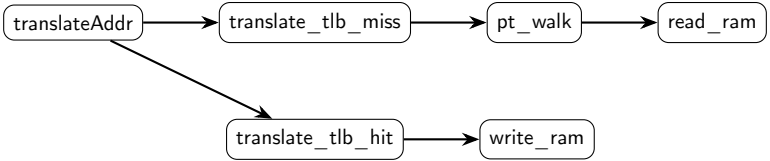
Дополнительный слайд. Sail-riscv (5)

Листинг 7: Пример необходимости удаления неиспользуемых выражений после специализации функции

```
function clause execute (PEXT_32_16_MUL_ADD(rs2, rs1, rd, op)) = {
  let rs1_val = X(rs1); let rs2_val = X(rs2); let rd_val = X(rd);
  result : xlenbits = zeros();
  foreach (i from 0 to (sizeof(xlen) - 32) by 32) {
    let rs1_word = slice(rs1_val, i, 32);
    let rs2_word = slice(rs2_val, i, 32);
    let rd_word = slice(rd_val, i, 32);
    result[(i+31)..i] = match op {
      SMMWB  => smmw(rs1_word, slice(rs2_word, 0, 16)),
      KMMAWB => kmmaw(rs1_word, slice(rs2_word, 0, 16), rd_word),
      ...
```

Дополнительный слайд. Обращение инструкции к памяти в sail-riscv

- При трансляции виртуального адреса в физический может происходить чтение/запись памяти[11]
- Чтение
  - Если TLB miss, тогда чтение таблицы страниц (page table) из памяти
- Запись
  - Если необходимо обновить бит A (accessed) или D (dirty) у записи таблицы страниц



[11] https://github.com/riscv/sail-riscv/blob/74c245ee2e501fd46dd1009c33a28b0a1992a9a7/doc/notes_Virtual_Memory.adoc#simplified-call-graph