LLVM-SNIPPY

Генерация случайных инструкций в инфраструктуре LLVM и верификация когерентности памяти.

К. Владимиров, Syntacore, 2024
mail-to: konstantin.vladimirov@gmail.com

Краткое введение в архитектуру RISC-V

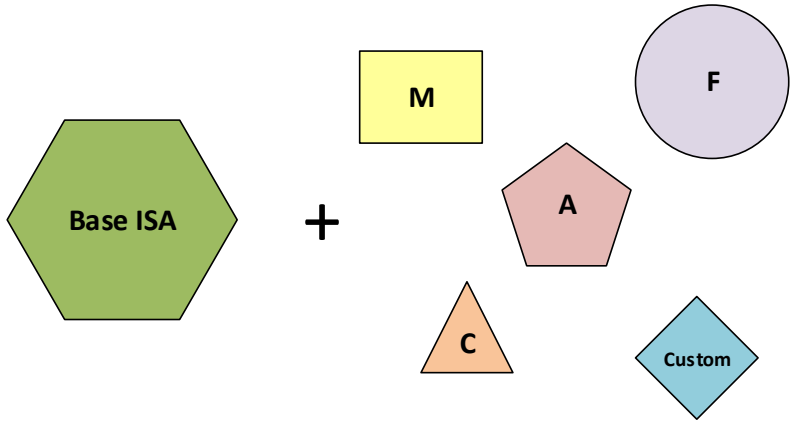
Модульная структура с поддержкой расширяемости и специализации.

- Простота базового набора.
- Широкий набор стандартных расширений.
- Разумное управление кодированием команд, существенное резервирование.

Стабильность.

- Базовый набор и стандартные расширения зафиксированы.
- Добавление функциональности через расширения, не выпуск новых версий.

Спецификации доступны для свободного и бесплатного использования.



Базовый целочисленный набор команд (на выбор)

- RV32I – 35 инструкций для работы с 32-битными целыми и 32-битными адресами.
- RV32E – вариация RV32I с урезанным количеством регистров общего назначения.
- RV64I, RV128I – наборы команд для поддержки расширенных целых и 64-битных адресов.

*** ISA это система команд процессора**

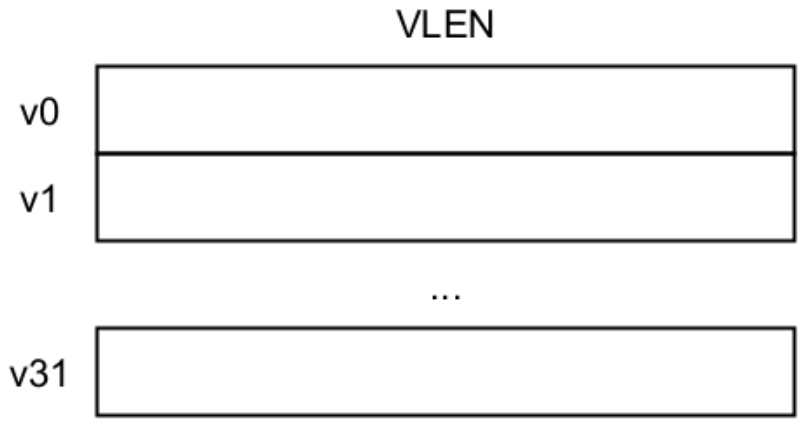
IMAFDQLCBJTPVN

- Стандартизованные расширения начинаются с буквы Z.
  - **zicsr** это инструкции для работы с CSR
  - **zifencei** это FENCE.I
  - **zam** это невыровненные атомики
  - **ztso** это total store ordering
- Расширения уровня супервизора начинаются с буквы S.
  - **sinval** это инвалидация TLB
- Нестандартные расширения начинаются с буквы X.

- **M** это умножение и деление.
- **A** это атомики.
- **F, D, Q** это различный FP.
- **C** это compressed.
- **V** это (scalable) vectorization.
- **B** это битовые манипуляции.
- **P** это packed SIMD.
- Буквы **Z**, **X**, **S** и **H** зарезервированы.

Сложность верификации железа: VPU

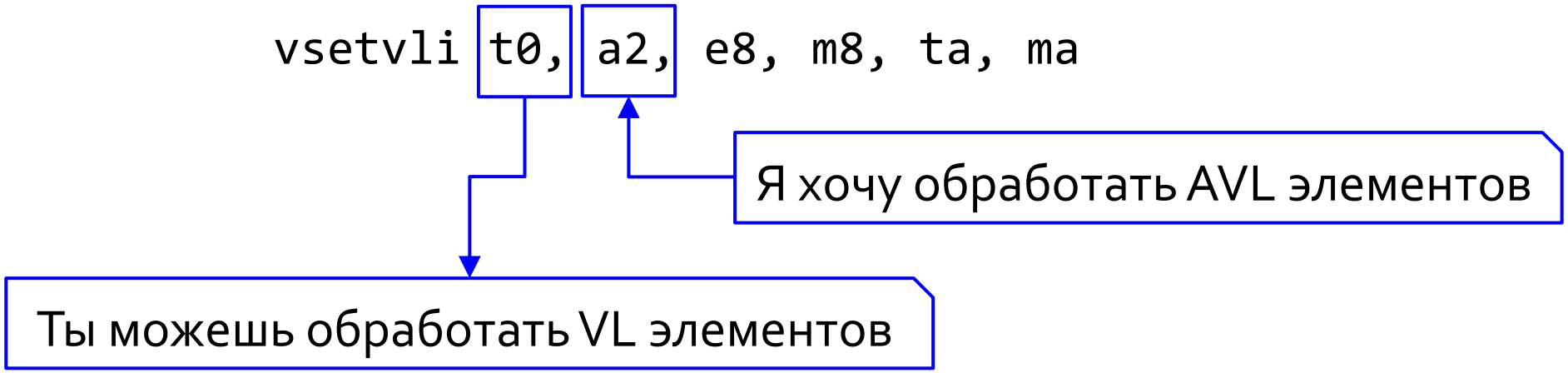
- Более 500 различных опкодов
- 4 (или более) варианта SEW
- 7 вариантов LMUL
- mask agnostic / undisturbed
- tail agnostic / undisturbed
- fixed-point rounding and saturation
- И это не считая вариантов аргументов и масок.
- 2.5 миллиона точек покрытия.



https://github.com/riscv/riscv-v-spec/blob/master/v-spec.adoc

Идея инструкций vset*, AVL и VL

```
vsetvli rd, rs1, vtype1    # AVL = x[rs1], x[rd] = vl
vsetivli rd, uimm, vtype1  # AVL = uimm, x[rd] = vl
vsetvl rd, rs1, rs2        # AVL = x[rs1], VT = x[rs2], x[rd] = vl

vset* rd, x0, ...          # AVL = VLMAX
```



Пример векторного кода: memcpy

```
.global memcpy
    # void *memcpy(void* dest, const void* src, size_t n)
    # a0=dest, a1=src, a2=n
memcpy:
    mv a3, a0                           // a3 = dest
loop:
    vsetvli t0, a2, e8, m8, ta, ma      // t0 = vsize up to a2
    vle8.v v0, (a1)                     // [v0..vN] = *a1;
    add a1, a1, t0                      // a1 += t0;
    sub a2, a2, t0                      // a2 += t0;
    vse8.v v0, (a3)                     // *a3 = [v0..vN];
    add a3, a3, t0                      // a3 += t0;
    bnez a2, loop                       // if (a2 != 0) goto loop;
    ret                                 // return a0;
```


memcpy example

Судьба ad-hoc тестового генератора

Пишем поддержку для простой базовой RISC-V ISA

- Пишем описания инструкций и их кодировку.
- Пишем модель с семантикой инструкций.
- До этой точки дошли: risc-v ctg, risc-v torture, riscv-dv, aapg, DiffuzRTL, etc...

Смело берёмся за расширения.

- Доходим до scalable vector extension (RVV) и начинаем страдать.
- В этой точке находятся: force risc-v, MicroTESK
- Дальше вспоминаем, что есть bitmanip, vector crypto, packed SIMD и кастомные расширения и сдаёмся.

Пример труда и страданий: force risc-v

```
<I name="VLSSEG7E16.V" isa="RISCV" class="LoadStoreInstruction" group="Vector">
  <O name="const_bits" type="Constant" bits="31-26,14-12,6-0"
value="1100101010000111"/>
  <O name="vd" type="VECREG" bits="11-7" access="Write" choices="Vector registers"
layout-type="FixedElementSize" reg-count="7" elem-width="16"/>
  <O name="vm" type="Choices" bits="25-25" choices="Vector mask" differ="vd"
class="VectorMaskOperand"/>
  <O name="LoadStore-rs1-rs2" type="LoadStore"
class="VectorStridedLoadStoreOperandRISCV" alignment="2" base="rs1" data-size="14"
element-size="2" index="rs2" mem-access="Read">
  <O name="rs1" type="GPR" bits="19-15" choices="Nonzero GPRs"/>
  <O name="rs2" type="GPR" bits="24-20" choices="Nonzero GPRs"/>
  </O>
  <asm format="VLSSEG7E16.V %s, %s, %s, %s" op1="vd" op2="rs1" op3="rs2" op4="vm"/>
</I>

<I name="VLSSEG7E32.V" isa="RISCV" class="LoadStoreInstruction" group="Vector">
  ....
```


v_instructions.xml

Судьба любого тестового генератора

Пишем поддержку для простой базовой RISC-V ISA

- (оптимизм и надежды)

Смело берёмся за расширения.

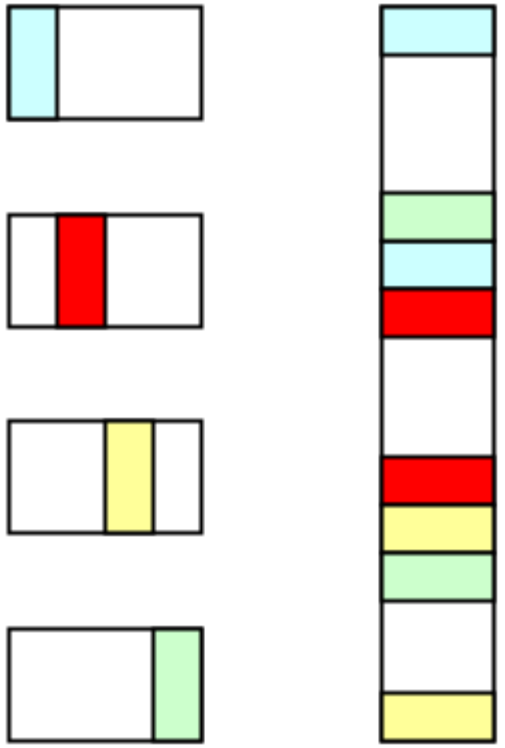
- (боль и страдания)

А ведь поддержка расширений это только вершина айсберга.

- Для верификации железа необходимо поддержать (иногда очень сложную) логику генерации.

Пример сложной логики

- Сложные и нерегулярные схемы доступа.
- При этом в случае векторных операций эти схемы должны быть наложены на доступ страйдами.
- Сложные цикловые паттерны и вызовы функций.
- Представьте также ситуации когда нужно совмещать работу с памятью и нетривиальную передачу управления.



Компиляторы спешат на помощь

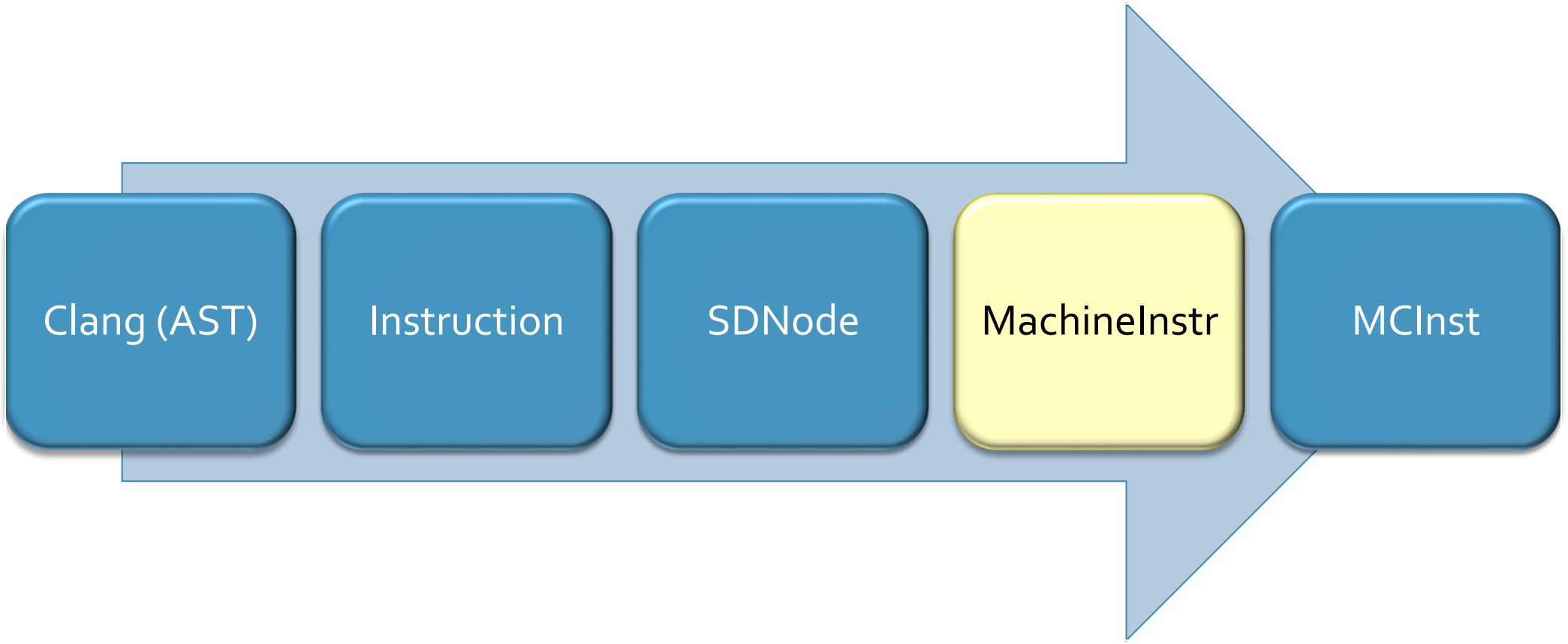
Преимущества инфраструктуры LLVM

- Уже есть описания всех инструкций и легко добавлять новые.
- Удобное промежуточное представление: базовые блоки, возможность делать любой control flow.
- Из коробки поддержаны вызовы функций и многое другое.
- Просто и удобно писать логику.

```
// unit-stride segment load vd, (rs1), vm
class VUnitStrideSegmentLoad<bits<3> nf,
  RISCVWidth width, string opcodestr> :
    RVInstVLU<nf, width.Value{3},
      LUMOPUnitStride, width.Value{2-0},
      (outs VR:$vd),
      (ins GPRMemZeroOffset:$rs1, VMaskOp:$vm),
      opcodestr, "$vd, ${rs1}$vm">;

// segment fault-only-first load vd, (rs1), vm
class VUnitStrideSegmentLoadFF<bits<3> nf,
  RISCVWidth width, string opcodestr> :
    RVInstVLU<nf, width.Value{3},
      LUMOPUnitStrideFF, width.Value{2-0},
      (outs VR:$vd),
      (ins GPRMemZeroOffset:$rs1, VMaskOp:$vm),
      opcodestr, "$vd, ${rs1}$vm">;
....
```

RISCVInstrInfoV.td

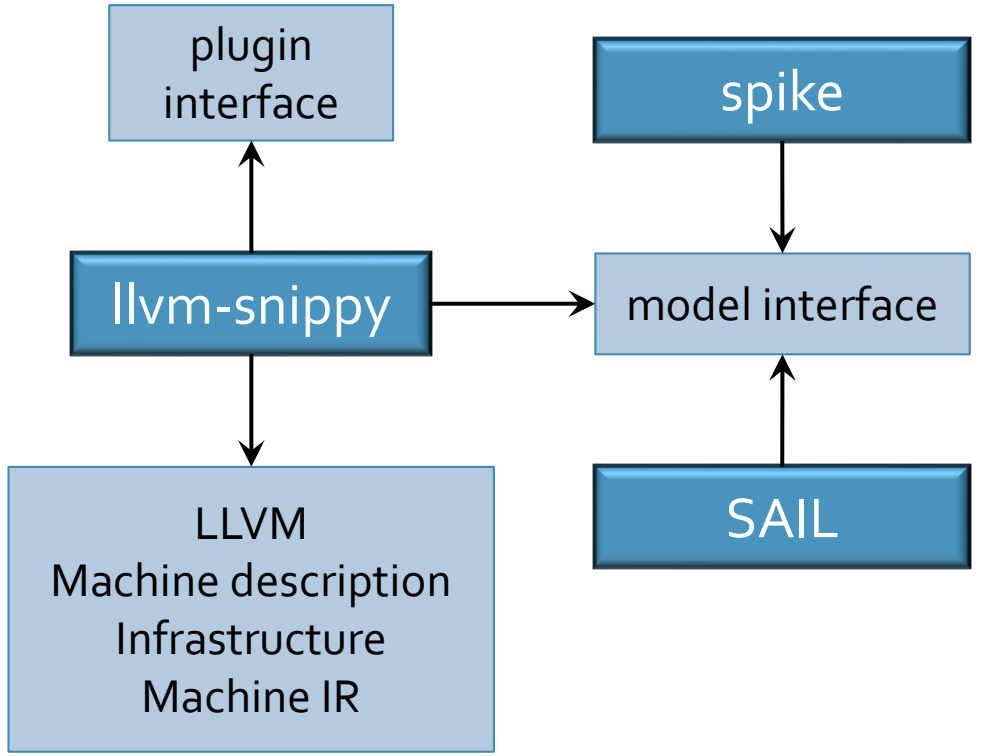
Кратко про инструкции в LLVM



[life-of-an-instruction-in-llvm]

Архитектура LLVM-snippy

- Генератор кросс-платформенный. Пока есть только RISC-V backend.
- Никакого ручного описания инструкций и их кодировки, мы используем LLVM.
- Никакого* описания семантики инструкций, мы используем внешние модели.
- В результате мы в генераторе можем сосредоточиться на логике генерации.
  - Паттерны доступа к памяти.
  - Циклы, условные переходы, функции.



* есть нюансы

Интерфейс модели

- Базовый интерфейс: что llvm-snippy может попросить от модели (rvm stands for RISC-V model interface).

```
rvm_modelCreate
rvm_modelDestroy
rvm_executeInstr
rvm_readMem / rvm_writeMem
rvm_readPC  / rvm_setPC
```

```
rvm_readXReg    / rvm_setXReg
rvm_readFReg    / rvm_setFReg
rvm_readVReg    / rvm_setVReg
rvm_readCSRReg / rvm_setCSRReg
rvm_logMessage
```

- Увы, необходимость поддерживать разные классы регистров делает модель бэкенд-специфичной. Для ARM или x86 интерфейс модели будет другим.

llvm/tools/llvm-snippy/Model/RISCVModel/RVM.h

Более сложный интерфейс

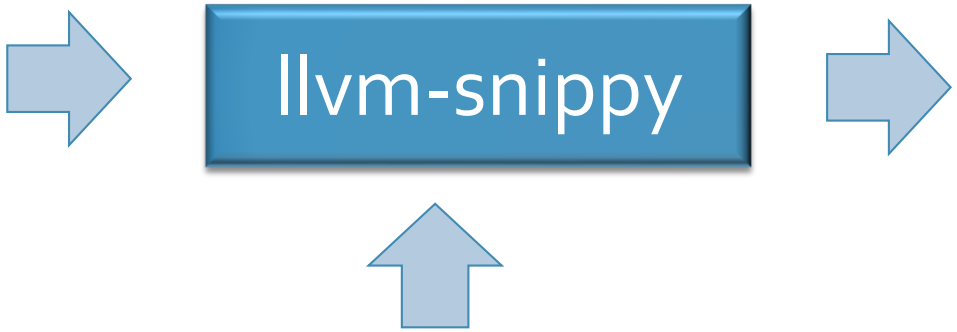
- Модель может быть предварительно сконфигурирована.
- Где у неё располагается память?
- Какие выставлены расширения: Z, X, Misa?
- Возможен ли misaligned access?
- Во что выставлен VLEN?
- Кроме того, модель может предоставлять callbacks.
- Всё это входит в **конфигурацию модели**. В интерфейсе это функция rvm_getModelConfig.

RVDash example model

Функционирование LLVM-snippy

ADD	1.5
SUB	1.5
OR	1.0
XOR	1.0
AND	1.0
SRA	2.0

TOTAL	8.0



text	VMA: 0x210000 SIZE: 0x100000 ACCESS: RX
data	VMA: 0x100000 SIZE: 0x100000 ACCESS: RW

```
...
add a2,a2,a0
or  s7,a2,a3
sra gp,gp,ra
xor s3,a0,a4
and a4,a1,a2
sra a3,gp,s5
sub s9,a2,s4
....
```

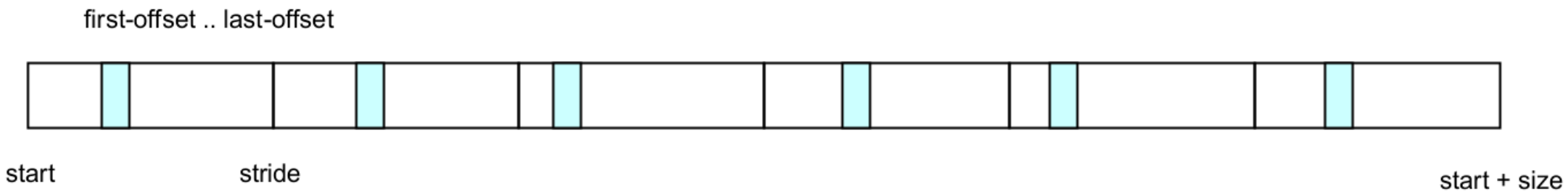
+ linker script

Генерация работы с памятью

- Для загрузки и сохранения в память нужен валидный адрес.
- При обычной генерации сниппета инструкция за инструкцией, валидный адрес сам по себе не обязательно образуется в регистре.
- Для качественного тестирования хорошо бы иметь возможность задавать и поддерживать регулярную схему доступа.

ADD	1.5
SUB	1.5
OR	1.0
XOR	1.0
AND	1.0
SRA	2.0
LW	2.0
SW	1.5

Схема доступа к памяти страйдами



$$k = rand(size/stride)$$

$$off\_len = last\_offset - first\_offset$$

$$off = rand(off\_len) + first\_offset$$

$$addr = start + k * stride + off$$

```
access-ranges:
  - start: 0x100000
    size: 0x10000
    stride: 16
    first-offset: 2
    last-offset: 8
```

Схема с фиксированными битами


```
- mask:  0xF8000000
  fixed: 0x0000C81
```

`0800C81, 1800C81, 1000C81, ….`

- Генератор поддерживает важный сценарий когда часть бит адреса зафиксирована, а часть изменяется.

Сочетания схем в настройках snippy

```
access-ranges:
    - weight: 10
      start: 0x100000
      size: 0x10000
      stride: 16
      first-offset: 0
      last-offset: 2

access-evictions:
    - weight: 5
      mask: 0x00FFF0
      fixed: 0x130000
```

```
sections:
  - name:       text
    VMA:        0x210000
    SIZE:       0x100000
    LMA:        0x210000
    ACCESS:     rx

  - name:       data
    VMA:        0x100000
    SIZE:       0x100000
    LMA:        0x100000
    ACCESS:     rw
```

Вспомогательные инструкции

- Чтобы соответствовать схеме памяти, нужно потратить несколько инструкций на формирование адреса.

```
sra s8,gp,s1
lui   t5,0x1c1
addiw t5,t5,378
sw    s6,206(t5)
lui   a1,0x1fa
addiw a1,a1,1054
sw    zero,-1438(a1)
sub   s8,a5,s7
```

ADD	1.0
ADDI	1.0
SUB	1.0
LW	2.0
SW	1.5
SRA	1.0

- В гистограме задаётся распределение вероятностей для основных инструкций.

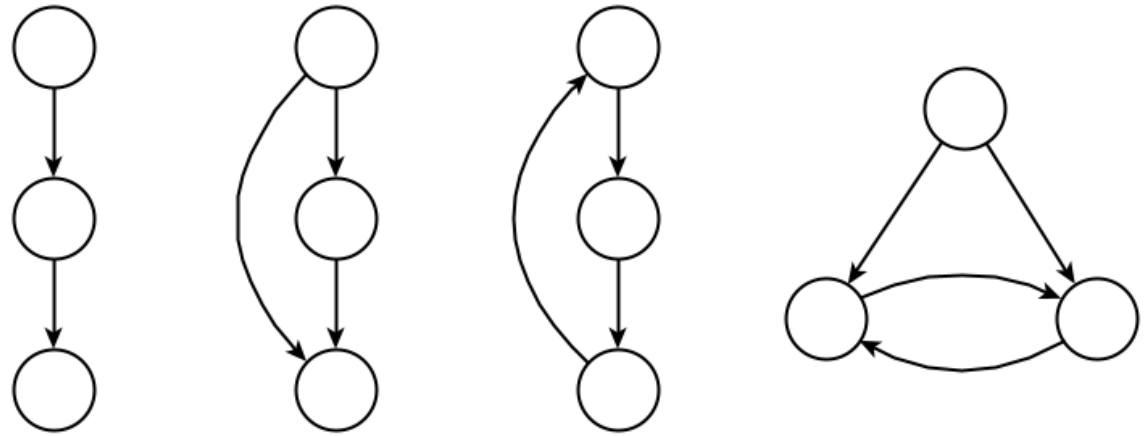
Поддержка для VPU config

```
riscv-vector-unit:
  mode-distribution:

    VM:
      - [all_ones, 2.0]
    VL:
      - [vlmax, 2.0]
      - [any_legal, 1.0]
    VXRM:
      rnu: 1.0
      rne: 1.0
      rdn: 1.0
    VTA:
      tu: 1.0
      ta: 1.0

VTYPE:
  SEW:
    sew_8: 1.0
    sew_16: 1.0
    sew_32: 1.0
  LMUL:
    m1: 1.0
    m2: 1.0
    m4: 1.0
VMA:
  mu: 1.0
  ma: 1.0
```

```
...
vsetvli s10,a0,e32,m1,ta,ma
vmxnor.mm v0,v0,v0
vmerge.vim  v5,v1,-14,v0
vadd.vi v4,v27,7,v0.t
vmsleu.vi v29,v10,-15
vmsleu.vi v26,v15,8,v0.t
vslidedown.vi v15,v26,5,v0.t
...
addi  a3,zero,64
vsetvli s3,a3,e8,m4,tu,mu
vmxnor.mm v0,v0,v0
vssrl.vi  v20,v16,7
vsra.vi v16,v4,2
...
```

Генерация потока управления

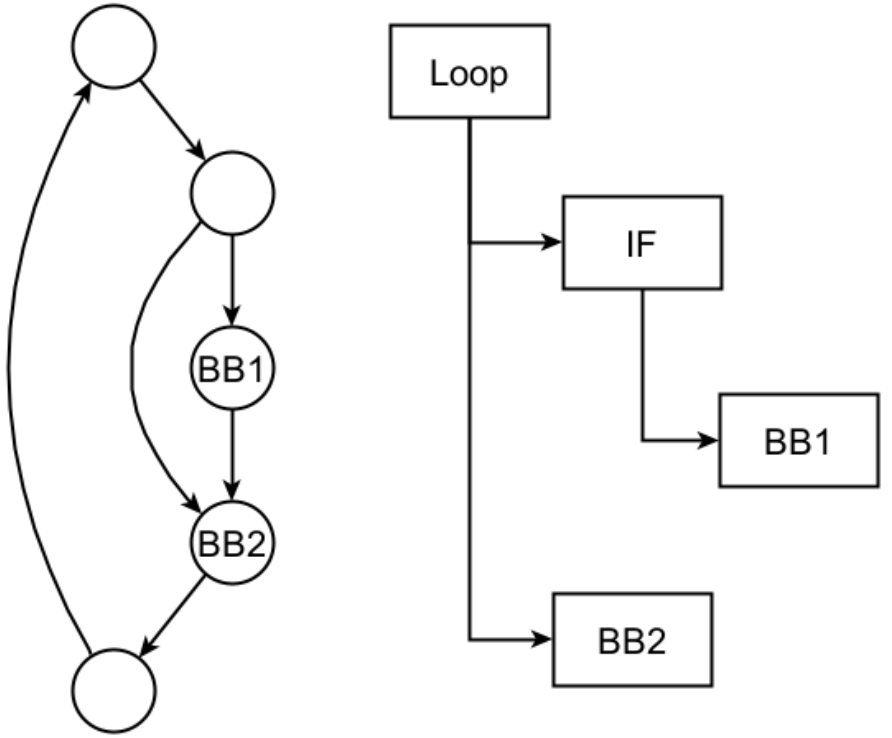
- Для llvm-snippy критично умение сгенерировать любую инструкцию поддержанную в LLVM.
- Это вызывает определённые проблемы, если это бранч.



ADD	1.0
ADDI	1.0
SUB	1.0
LW	2.0
SW	1.5
SRA	1.0
BEQ	1.0
BNE	1.0

Основные вопросы

- Как выбрать целевой базовый блок для бранча?
- Как избежать бесконечного исполнения сгенерированной программы?
- Как согласовать наличие потока управления со схемами памяти, burst и всем остальным?



Основные решения для control flow

- Генерация поддерживает структурный control flow без несводимых конструкций.
- На каждый цикл резервируется регистр для индуктивной переменной.
- При генерации оценивается глубина вложенности и задаётся вероятность того чем будет очередная конструкция.

```
branches:
  - permutation: on
    loop-ratio: 0.5
    number-of-loop-iterations:
      - min: 2
        max: 32
    max-depth:
      - if: 500
      - loop: 4
```

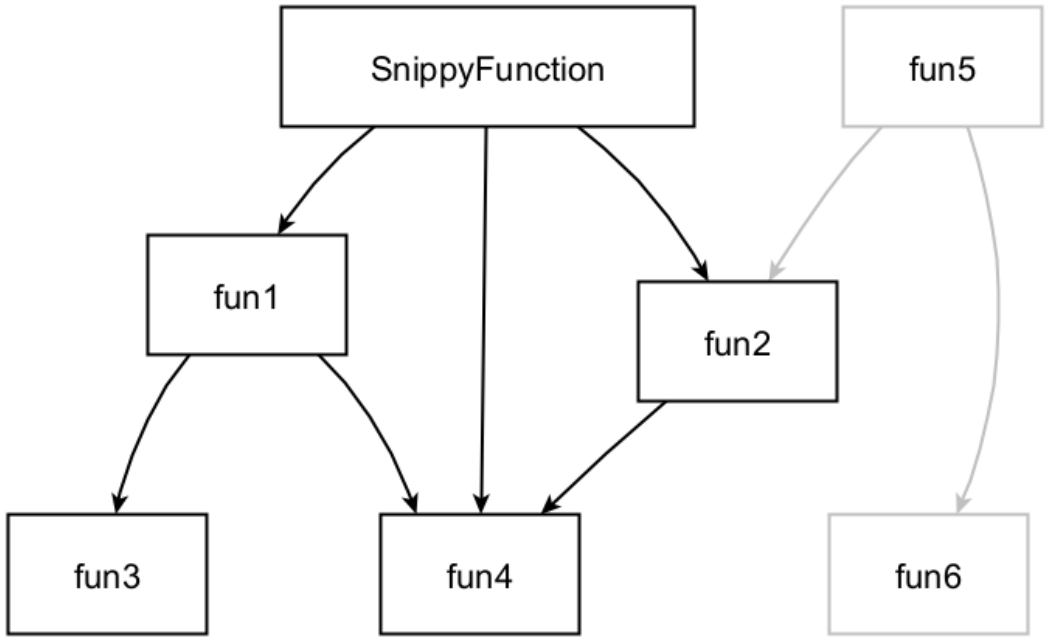
Вызовы функций

- Для llvm-snippy критично умение сгенерировать любую инструкцию поддержанную в LLVM.
- Это вызывает определённые проблемы, если это JAL или JALR.
- Как избежать бесконечной обобщённой рекурсии?
- Какое ABI использовать для сгенерированных функций?
- Как их хотя бы назвать?

ADD	1.0
ADDI	1.0
SUB	1.0
LW	2.0
SW	1.5
SRA	1.0
JAL	1.0
JALR	1.0

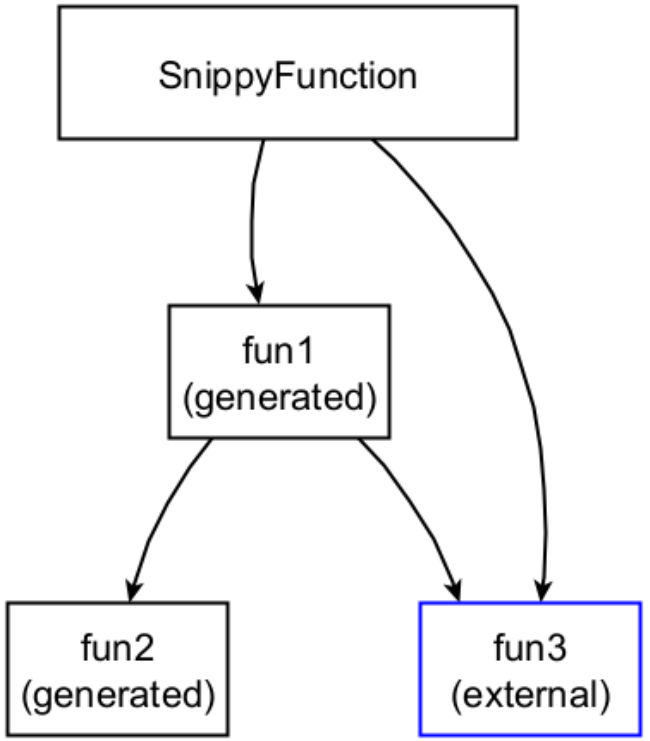
Основные решения для function call

- Строго иерархический call graph без циклов.
- Возможность наличия недостижимых функций.
- Никаких требований к внутреннему ABI. Функции свободно портят любые регистры.
- Единственная **точка уважения** к ABI это внешняя функция сниппета.



Внешний call-graph

- Пользователь имеет возможность не только попросить сгенерировать граф вызовов.
- Можно самостоятельно написать функцию и попросить вставить её в граф вызовов для сниппета а потом слинковаться с кодом, написанным вручную.
- Таким образом не только сниппет благодаря внешнему следованию ABI может быть помещён к вам, но и ваш код может быть помещён в сниппет.



Некоторые проблемы

- Что такое вероятность для инструкции в условиях нетривиального control-flow?
- Что делать генератору если в процессе генерации выпадает инструкция, которая требует специальной семантики?
- Кажется это разрушает идею последовательной генерации инструкций.
- В компиляторном мире сначала определяется call graph, дальше control-flow graph а дальше уже генерируются остальные инструкции.

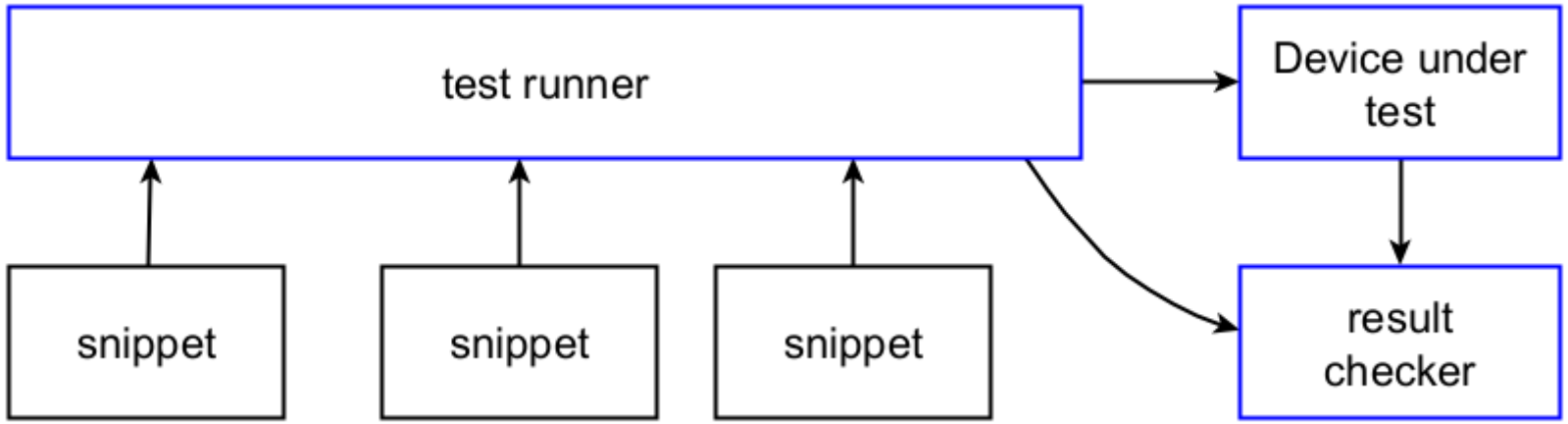
ADD	1.0
ADDI	1.0
SUB	1.0
LW	2.0
SW	1.5
SRA	1.0
JAL	1.0
BEQ	1.0

Структура пассов в генераторе

- Обычно компилятор построен вокруг пасс-менеджера.
- Пасс или фаза оптимизации работает с предусловием и постусловием валидного MachineIR.
- Для llvm-snippy у нас есть развитый пасс-менеджер, состоящий из фаз:
  - Генерации или загрузки control-flow.
  - Генерации и (опционально) рандомизации call-graph.
  - Разметки call-graph под модальные режимы, такие как burst.
  - Собственно генерирование потока инструкций с учётом вероятностей и разметки.
  - И много более того!
- Вы можете расширять его своими пассами, делающими то, что вам нужно.

Использование генератора

- Поскольку на выходе только сниппет и линкер скрипт, всю обвязку для тестирования пользователь предоставляет самостоятельно.



- Чтобы это функционировало каждый сниппет может обеспечить загрузку и дамп состояния регистров и дамп памяти после корректной работы.

Общие выводы

- Мы переиспользуем LLVM как в части описания инструкций так и в части инфраструктуры.
- Это позволяет концентрироваться на логике генерации.
- Генератор может быть эффективно использован для:
  - Верификации отдельных расширений (включая довольно сложные векторные).
  - Верификации подсистемы памяти.
  - Верификации бранч-предиктора и в целом фронтенда.
- В проект можно и нужно контрибьютить!

Q & A

Всем спасибо, ждём ваших коммитов и комментариев.

https://github.com/syntacore/snippy/releases/download/snippy-1.0/snippy_guide.pdf