



Syntacore™
Custom cores and tools

Компиляторные технологии в верификации аппаратного обеспечения

Константин Владимиров, 2025

info@syntacore.com

RISC-V International



RISC-V это свободная и открытая RISC ISA

- Является результатом общемирового сотрудничества

RISC-V Foundation / International

- Основан в **2015** индустриальными лидерами и стартапами
- **4000 +** членов из **70 +** стран
- Продвигает исследования и инновации

Российский Альянс RISC-V

- Создан для развития и популяризации архитектуры **RISC-V** в нашей стране, представления и защиты общих интересов участников
- **Основная цель** – создание открытого сообщества разработчиков программного и аппаратного обеспечения



Преимущества RISC-V

Простота

- Базовый набор команд существенно меньше, чем другие коммерческие ISA

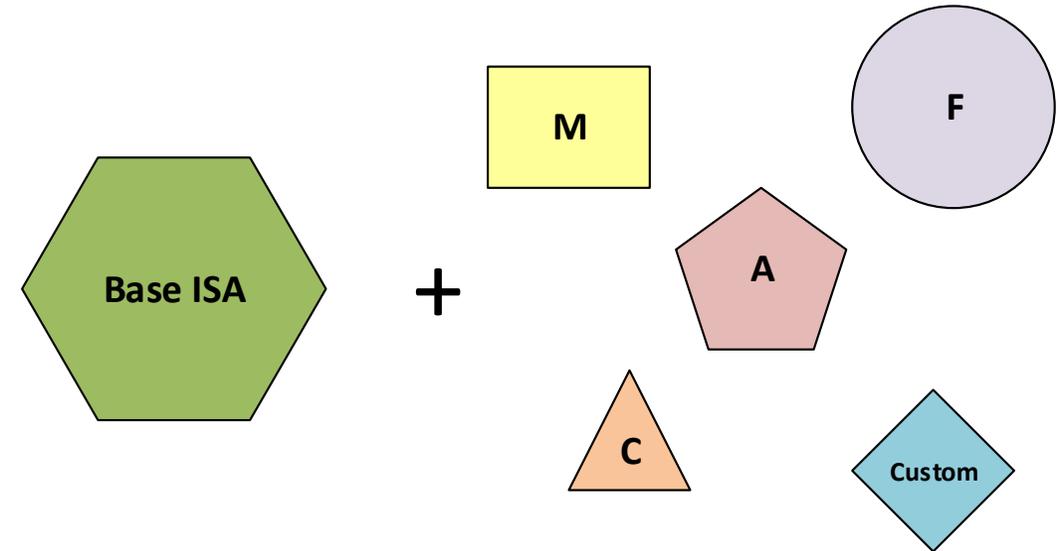
Модульная структура с поддержкой расширяемости и специализации

- 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)

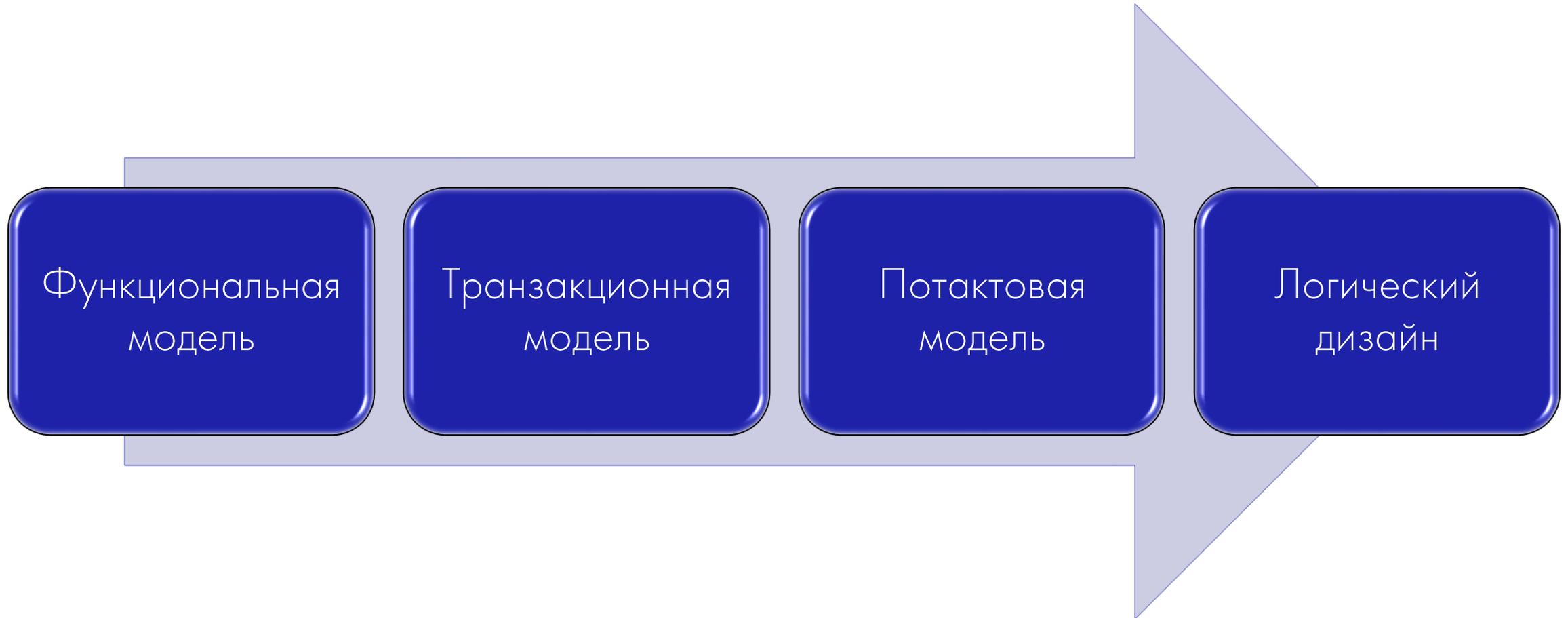
- RV32I – для работы с 32-битными целыми и адресами
- RV32E – вариация RV32I с урезанным количеством регистров общего назначения
- RV64I, RV128I – наборы команд для поддержки расширенных целых и 64-битных адресов



Расширяемость RISC-V: IMAFDQLCBJTPVN

- Стандартизированные 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
- Расширения уровня гипервизора начинаются с буквы H.
- 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.

Моделирование RISC-V на разных уровнях

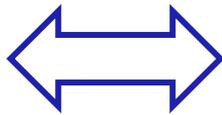




Верификация модели

- Задача верификации это задача проверки соответствия спецификации (например спецификации RISC-V)
- Как решать эту задачу в общем случае?

Model
under test



The RISC-V Instruction Set Manual
Volume I: Unprivileged ISA
Document Version 20191213

Editors: Andrew Waterman¹, Krste Asanović^{1,2}
¹SiFive Inc.,

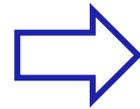
²CS Division, EECS Department, University of California, Berkeley
andrew@sifive.com, krste@berkeley.edu
December 13, 2019



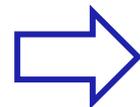
Идея ISG и ко-симуляция

- Генератором последовательности инструкций (ISG) называется программа, которая способна выдавать одну за другой инструкции с заданными ограничениями.

```
...  
addiw a2,a2,276  
lw s7,340(a2)  
sra gp,gp,ra  
addi s3,a0,495  
sra a4,a1,a2  
sra a3,gp,s5  
sub s9,a2,s4  
....
```



Model under test



Golden model



Применение ISG для широкого класса моделей

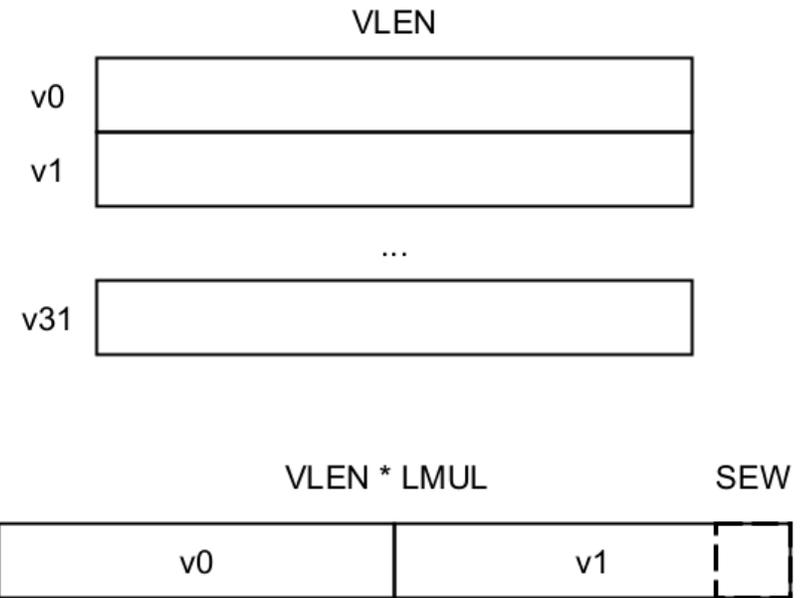
- Тестирование cycle-approximate модели. Тот же поток инструкций, который обрабатывается вашим дизайном, должен обрабатываться и его высокоуровневой моделью.
 - Можно найти точки максимального расхождения между предсказаниями модели и логическим дизайном. Они могут служить к улучшению модели или к улучшению дизайна.
- Тестирование функциональных симуляторов.
 - Симуляторы, такие, как spike, SAIL и QEMU фактически работают над потоком инструкций. Известны open-source проекты симуляторов, в которых llvm-snipru используется для косимуляции против spike.
 - Частный, но важный случай это тестирование valgrind. Инжектирование инструкций идущих мимо схемы памяти и правильная отработка таких ошибок в valgrind.
- Тестирование отладчика: поток случайных инструкций может быть аннотирован дебаг-информацией (например про LoC).



Сложность верификации на примере RVV

```
vsetvli rd, rs1, vtype      # AVL = x[rs1], x[rd] = v1
vsetivli rd, uimm, vtype   # AVL = uimm, x[rd] = v1
vsetvl rd, rs1, rs2        # AVL = x[rs1], VT = x[rs2], x[rd] = v1
vset* rd, x0, ...         # AVL = VLMAX
```

- Несколько сотен различных опкодов.
- 4 (или более) варианта SEW, 7 вариантов LMUL.
- Agnostic и undisturbed режимы для tail и mask.
- Fixed-point rounding and saturation.
- И это не считая вариантов аргументов и масок даёт более чем 2.5 миллиона точек покрытия.





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">
  . . . .
```



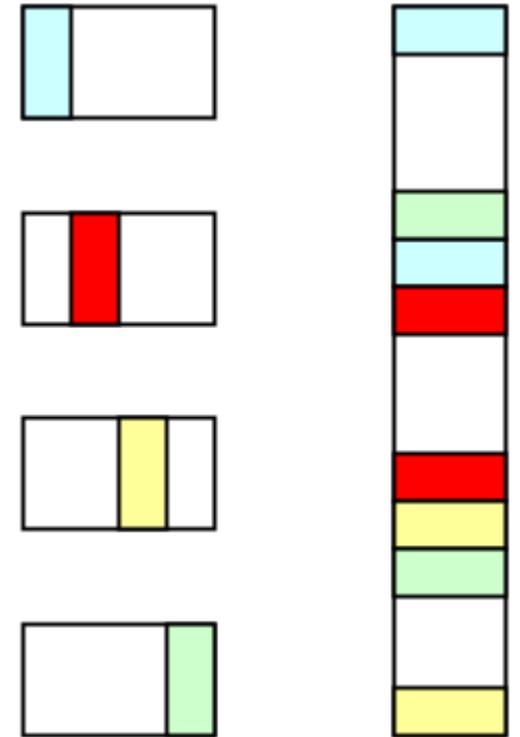
Судьба ad-hoc тестового генератора

- 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.
- А ведь поддержка расширений это только вершина айсберга.
 - Для верификации железа необходимо поддерживать (иногда очень сложную) логику генерации.

Примеры сложной логики



- Инициализация регистров и памяти.
 - На FPGA выгодно вызывать специальную функцию для инициализации
 - Для симулятора наоборот выгодно чтобы все секции были уже зашиты в бинарный файл.
- Сложные и нерегулярные схемы доступа.
 - Возможность игнорировать какие-то адреса.
 - В случае векторных операций эти схемы должны быть наложены на доступ страйдами.
 - Возможность обращаться по адресам, которые в свою очередь загружены из памяти.
- Цикловые паттерны и вызовы функций.
 - Резервация регистров под индуктивные переменные.
 - Последовательные циклы.





Компиляторы спешат на помощь: LLVM

- LLVM предоставляет развитую открытую инфраструктуру для написания инструментов, основанных на компиляторе.
- Уже есть описания всех инструкций, которые может породить компилятор и легко добавлять новые.
- Любое расширение в любом случае сначала появляется в компиляторе.
- Из коробки поддерживаны различные варианты ABI, существует поддержка сброса в разные объектные файлы и ассемблер.
- LLVM это превосходный источник информации об архитектуре.

```
def ADDIW : RVInstI<0b000, OPC_OP_IMM_32,
  (outs GPR:$rd), (ins GPR:$rs1, simm12:$imm12),
  "addiw", "$rd, $rs1, $imm12">,
  Sched<[WriteIALU32, ReadIALU32]>;

def SLLIW : ShiftW_ri<0b0000000, 0b001, "slliw">;
def SRLIW : ShiftW_ri<0b0000000, 0b101, "srliw">;
def SRAIW : ShiftW_ri<0b0100000, 0b101, "sraiw">;

def ADDW   : ALUW_rr<0b0000000, 0b000, "addw", Commutable=1>,
  Sched<[WriteIALU32, ReadIALU32, ReadIALU32]>;

def SUBW   : ALUW_rr<0b0100000, 0b000, "subw">,
  Sched<[WriteIALU32, ReadIALU32, ReadIALU32]>;

def SLLW   : ALUW_rr<0b0000000, 0b001, "sllw">,
  Sched<[WriteShiftReg32, ReadShiftReg32]>;

def SRLW   : ALUW_rr<0b0000000, 0b101, "srlw">,
  Sched<[WriteShiftReg32, ReadShiftReg32]>;

def SRAW   : ALUW_rr<0b0100000, 0b101, "sraw">,
  Sched<[WriteShiftReg32, ReadShiftReg32]>;
```



Компиляторы спешат на помощь: LLVM

- LLVM предоставляет развитую открытую инфраструктуру для написания инструментов, основанных на компиляторе.
- Уже есть описания всех инструкций, которые может породить компилятор и легко добавлять новые.
- Любое расширение в любом случае сначала появляется в компиляторе.
- Из коробки поддерживаны различные варианты ABI, существует поддержка сброса в разные объектные файлы и ассемблер.
- LLVM это превосходный источник информации об архитектуре.

```
// unit-stride segment load vd, (rs1), vm
class VUnitStrideSegmentLoad<bits<3> nf,
  RISCWidth 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">;

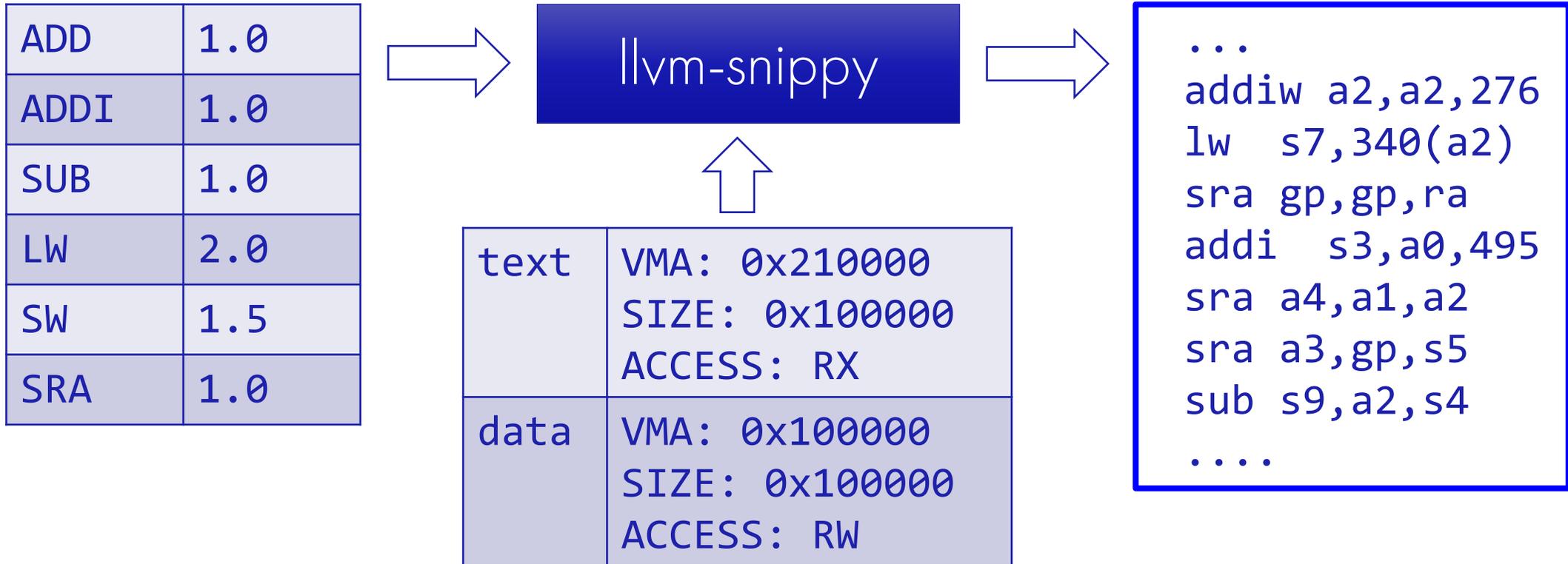
// segment fault-only-first load vd, (rs1), vm
class VUnitStrideSegmentLoadFF<bits<3> nf,
  RISCWidth width, string opcodestr> :
  RVInstVLU<nf, width.Value{3},
  LUMOPUnitStrideFF, width.Value{2-0},
  (outs VR:$vd),
  (ins GPRMemZeroOffset:$rs1, VMaskOp:$vm),
  opcodestr, "$vd, ${rs1}$vm">;

....
```



Первый взгляд

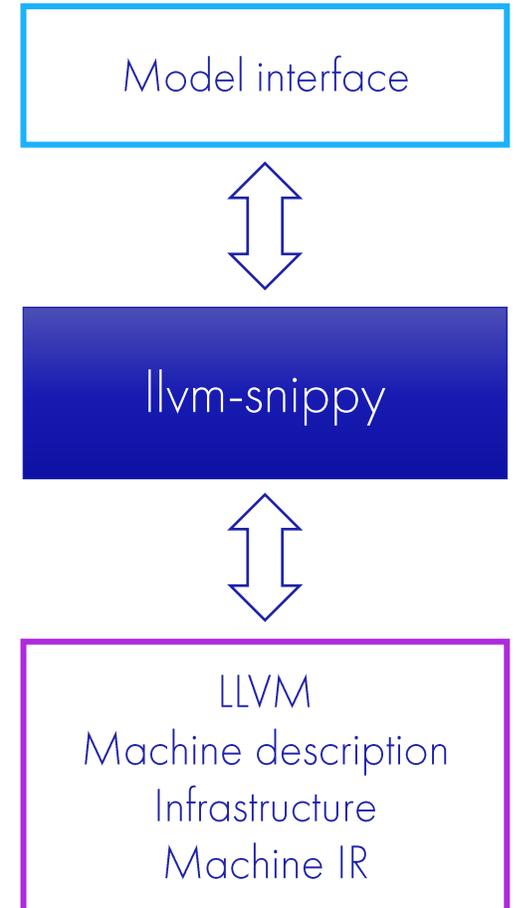
- LLVM-snippy это потенциально кросс-платформенный генератор последовательностей машинных инструкций.





Архитектура генератора

- Генератор кросс-платформенный. Пока что есть только RISC-V backend. Мы рады всем, кто принесёт нам свой бэкенд.
- 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**.
- Как вы думаете сколько в генераторе только верхнеуровневых групп настроек?



* есть нюансы



Настройки генератора (верхний уровень)

sections

histogram

imm-hist

fpu config

memory
schemes

burst

branches

call graph

extensions

target
specific



Анатомия простейшего snippets

sections:

```
- name:      code
  VMA:      0x210000
  SIZE:     0x100000
  LMA:      0x210000
  ACCESS:   rx
- name:      data
  VMA:      0x100000
  SIZE:     0x100000
  LMA:      0x100000
  ACCESS:   rw
```

histogram:

```
- [ADD, 1.0]
- [SUB, 1.0]
- [AND, 1.0]
- [OR, 1.0]
- [ORI, 1.0]
- [XOR, 1.0]
- [XORI, 1.0]
- [LW, 10.0]
- [SW, 10.0]
```

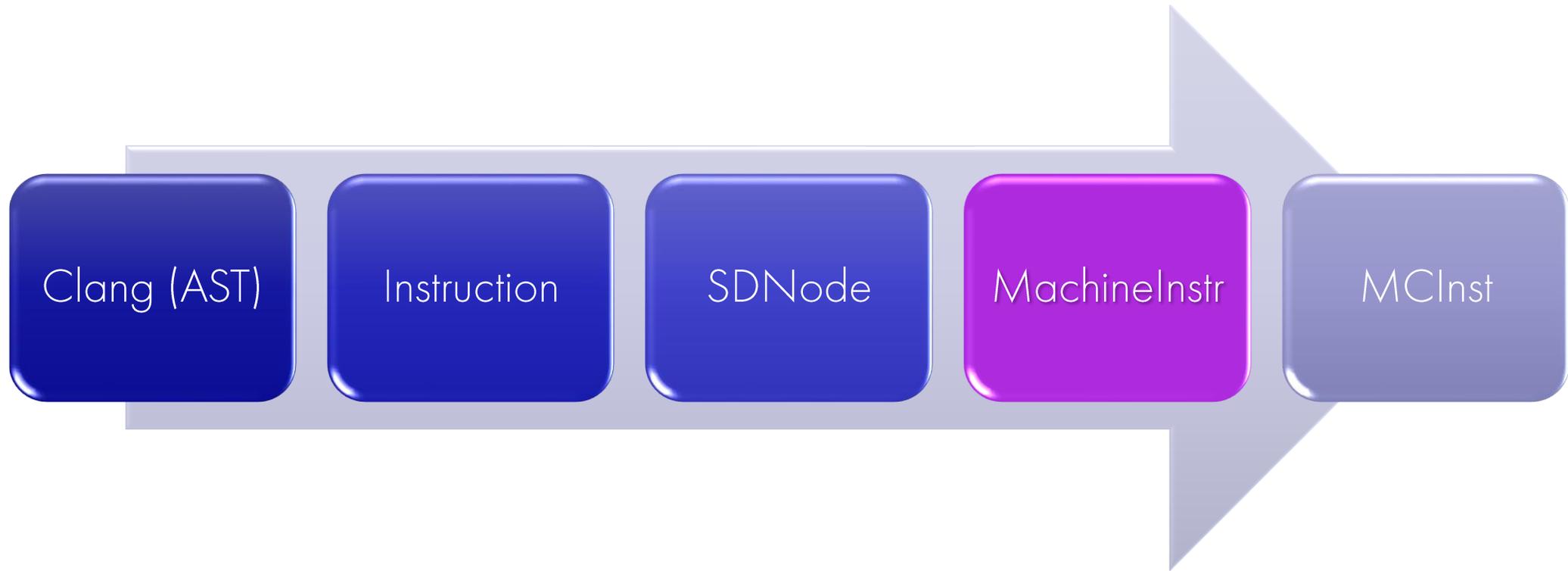
```
$ ./snippy --list-opcode-names -march=riscv64-linux-gnu
```

```
$ ./snippy -march=riscv64-linux-gnu ./yml/layout.yaml -model-plugin=None
```



Компиляторы спешат на помощь: machine IR

- LLVM предоставляет несколько уровней промежуточного представления.
- LLVM-snippy использует Machine IR.





Чем заканчивается сниппет?

- Вы можете управлять чем является последняя (не входящая в гистограмму) инструкция сниппета.
- По умолчанию это **EBREAK**.
- Можно выставить `--last-instr=OPCODE` где **OPCODE** это любой поддерживаемый опкод.
- Также популярны:
 - `--last-instr=RET` но тогда вы сами отвечаете за то, куда ему возвращаться (см. далее).
 - `--last-instr=` задаёт ситуацию, когда сниппет ничем не заканчивается.
- Все настройки `llvm-snippy` это **YAML файлы**. Порядок их подачи не важен, но вы не имеете права разрывать один и тот же верхнеуровневый ключ между несколькими файлами.



Пример настройки: выбор используемых immediates

- Мы можем управлять вероятностями того какие immediate values появятся в операциях.

```
$ ./llvm-snippy -march=riscv64-linux-gnu ./yaml/layout.yaml -seed=0  
--trace-log=trace.log ./yaml/imm-hist.yaml
```

- Слева -- что встречается, справа -- вероятность.

```
imm-hist:  
  opcodes:  
    - '.+': uniform
```

- Можно ставить более интересные варианты.
- В том числе с поддержкой регулярных выражений.

```
imm-hist:  
  opcodes:  
    - "ADDI":  
      - [1, 1.0]  
      - [2, 1.0]  
    - "L[DWH]":  
      - [1, 1.0]
```



Сложность в генерации сниппетов с плавающей точкой

- Наивная попытка сделать сниппет очень быстро вырождает почти все его значения регистров в NaN.

`./llvm-snippy`

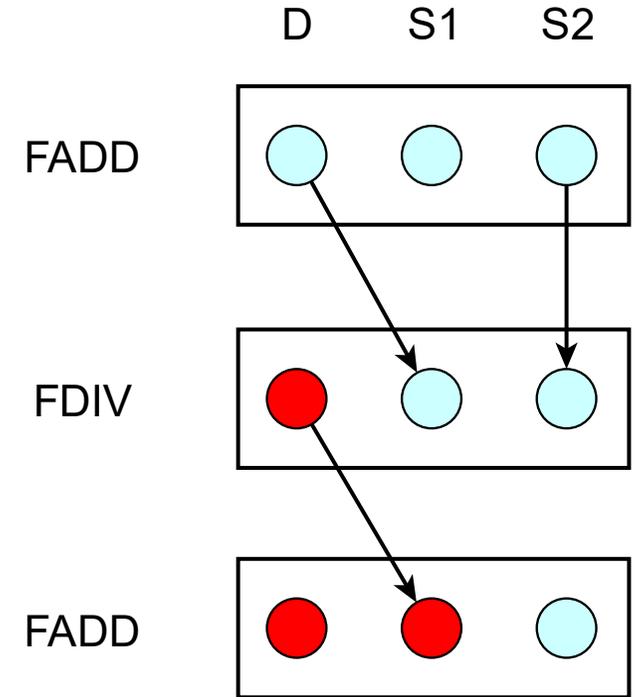
`./yaml/layout-fd.yaml`

`./yaml/memory.yaml`

`./yaml/fpu-config.yaml`

- Здесь дополнительная конфигурация указывает когда мы ловим и переписываем.
- То, чем мы переписываем, это тоже valuegram.

```
fpu-config:
  overwrite:
    mode: if-all-operands
    range:
      min: -100
      max: 100
      weight: 1.0
    rounding-mode: rup
  ieee-single:
    valuegram:
      - [0x7f800000, 1.0]
      - [0x80000000, 1.0]
      - type: bitrange
        min: 0x7f800000
        max: 0x80000000
        weight: 1.0
```





Сложность верификации моделей на примере кешей

Проблемы когерентности кешей.

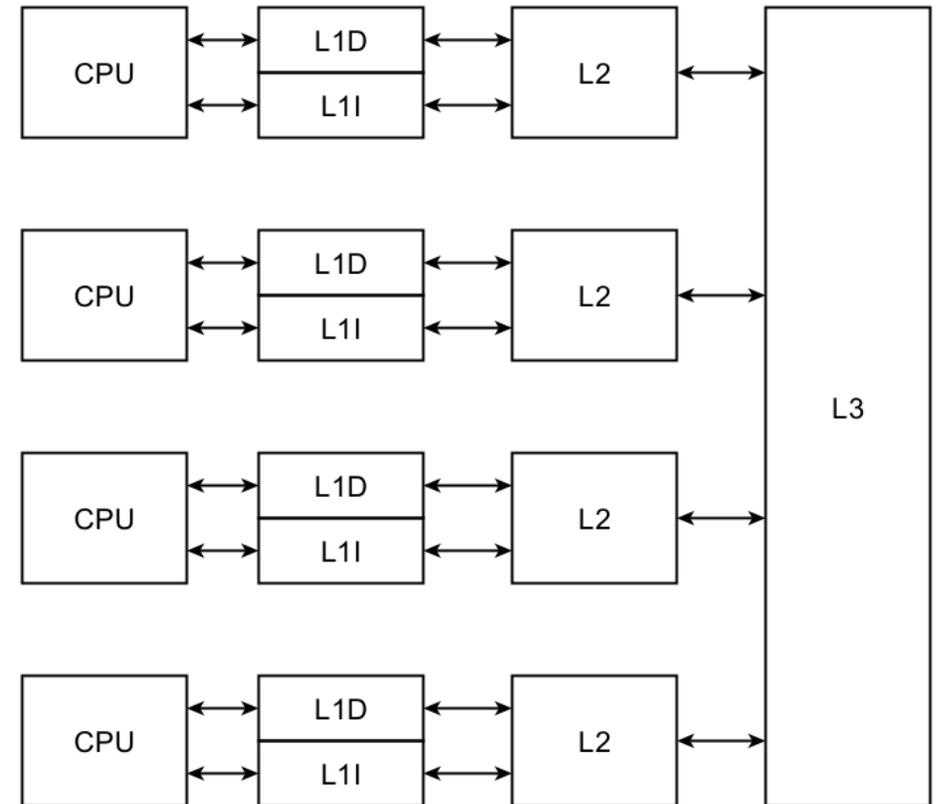
- Тесты на false sharing
- Тесты на вытеснения из кешей
- Тесты на сочетания кешей инструкций и данных
- Тесты на работу кешей и префетчера

Проблемы сочетания кешей с расширениями.

- То же самое но с векторными loads/stores
- То же самое но вместе с FPU

Проблемы сочетания instruction cache с control flow.

- Нетривиальная передача управления плюс память?
- То же самое плюс вызовы функций?



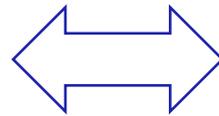


Схемы доступа и секции доступа

- Схема доступа может идти мимо секций.
- Это нормально, так как секции задаются для хранения данных для `sniprru`, а доступ может быть и по внешним адресам.
- Если схема доступа не задана, схема доступа по умолчанию это каждый байт какой-то RW секции.

sections:

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



access-ranges:

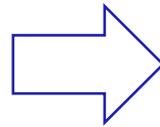
```
- start: 0x102000
  size: 0x1000
  stride: 16
  first-offset: 1
  last-offset: 2
```



Компромисс: ancillary instructions

- Чтобы соответствовать схеме памяти, нужно потратить несколько вспомогательных (ancillary) инструкций на формирование адреса.

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



```
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
```

- В гистограмме задаётся распределение вероятностей только **для основных** инструкций.
- Вспомогательных инструкций с точки зрения генератора не существует.



Пройтись в другом порядке по тем же адресам?

- Вы можете сдампить те адреса, которые были использованы

```
./llvm-snippy -march=riscv64-unknown-elf -mcpu=scr9 \  
  ./yaml/layout.yaml ./yaml/eviction.yaml -model-plugin=spike \  
  -num-instrs=1000 -seed=0 --trace-log=log \  
  --dump-memory-accesses=acc.yaml -o layout.elf
```

- И далее использовать ровно эти адреса

```
./llvm-snippy -march=riscv64-unknown-elf -mcpu=scr9 \  
  ./yaml/layout.yaml ./acc.yaml -model-plugin=spike \  
  -num-instrs=100 -seed=0 -o layout.elf
```

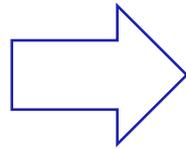
- В схеме `access-addresses` адреса могут быть `ordered` (тогда по ним снippet ходит в том же порядке) или `unordered` (тогда в случайном).



Режим более существенной нагрузки (burst mode)

- Иногда хочется создавать существенную нагрузку когда много операций с памятью идут одна за другой.

```
x6 = LUI 262529
x6 = SLLI x6, 1
x6 = ADDI x6, 451
x31 = LW x6, 173
x14 = LUI 65643
x14 = SLLI x14, 3
x14 = ADDI x14, -276
x23 = LW x14, 1948
x16 = ADDI x0, 257
x16 = SLLI x16, 23
x16 = ADDI x16, 1632
SD x9, x16, 496
```

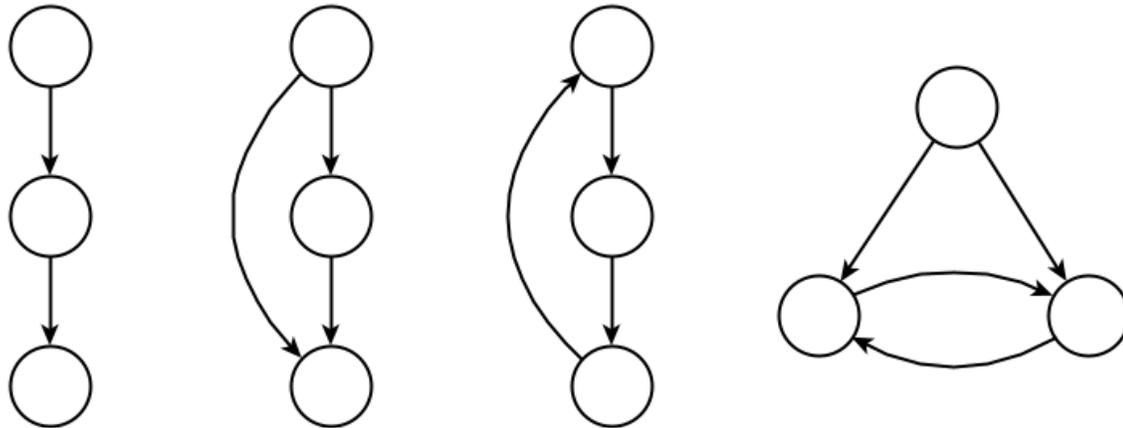


```
x6 = ADDI x6, 451
x14 = LUI 65643
x14 = SLLI x14, 3
x14 = ADDI x14, -276
x16 = ADDI x0, 257
x16 = SLLI x16, 23
x16 = ADDI x16, 1632
x22 = LBU x29, -1156
SW x11, x15, -1257
x31 = LW x6, 173
x23 = LW x14, 1948
SD x9, x16, 496
```



Поток управления (control flow)

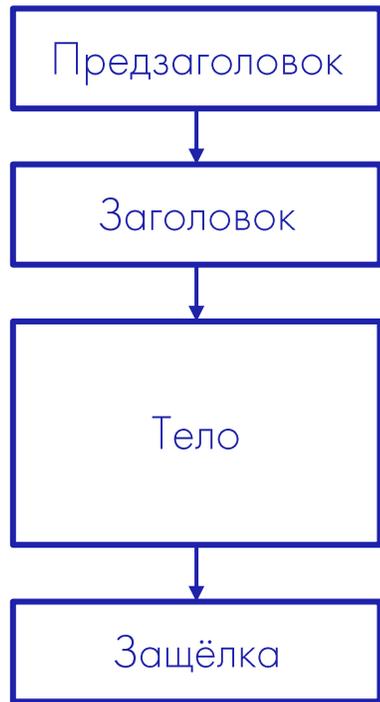
- Для llvm-snirpy критично умение сгенерировать любую инструкцию, поддерживаемую в LLVM.
- Это вызывает определённые проблемы, если это бранч.



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



Компиляторы спешат на помощь: циклы в LLVM



Устанавливаем индуктивные переменные (из числа зарезервированных регистров).

В заголовок можно попасть только из защёлки и предзаголовка

В теле и в заголовке не используются зарезервированные регистры.

Инкрементируем и проверяем значения против условия выхода.

- Мы резервируем два регистра на каждый уровень вложенности цикла.
- Это может приводить к проблемам нехватки регистров (особенно для compressed инструкций).



С точки зрения пользователя: бранчеграмма

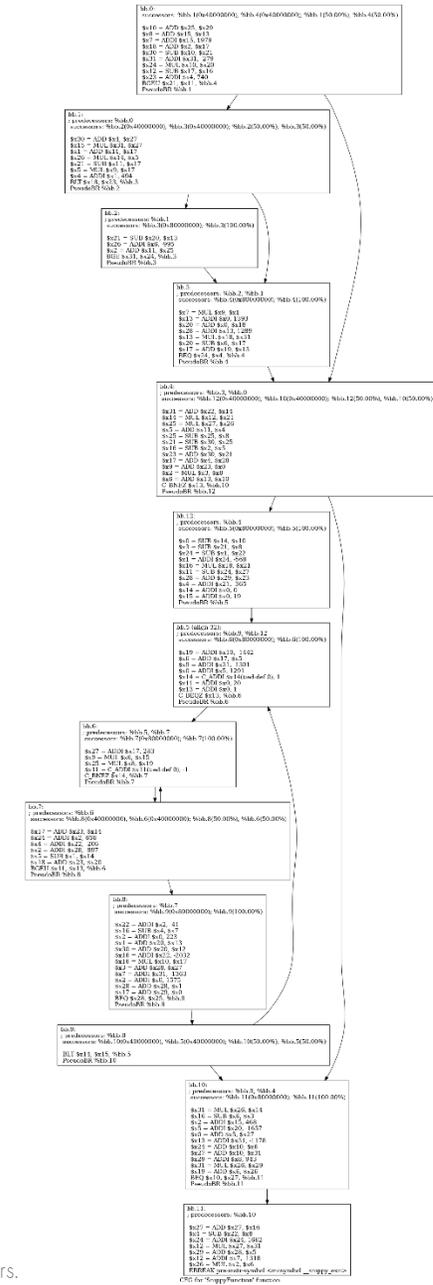
```
./llvm-snippy -march=riscv64-linux-gnu
./yaml/layout-branches.yaml -num-instrs=100
./yaml/memory-aligned.yaml -model-plugin=None
```

- Для более подробной информации о CFG


```
--dump-cfg --cfg-basename=branches
```

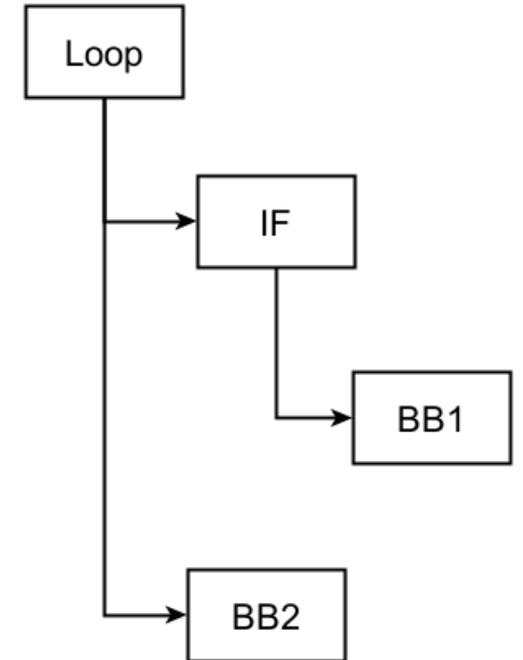
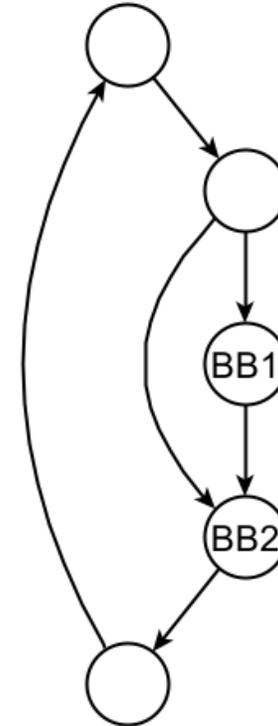
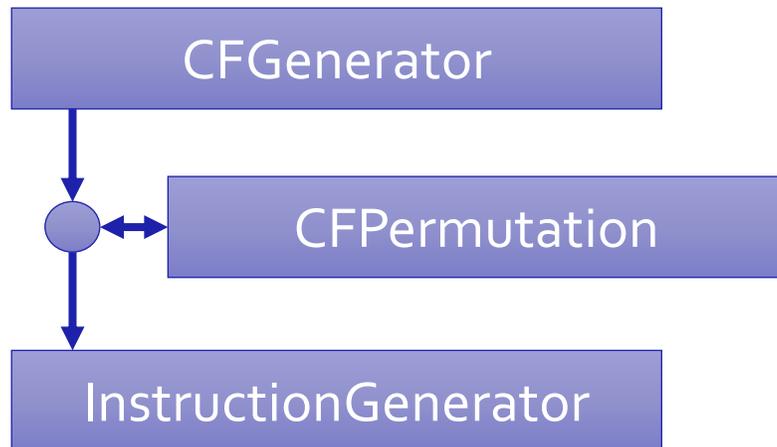
branches:

- permutation: on # enable CF permutation, on by default
- alignment: 32 # in bytes, 1 by default
- loop-ratio: 0.5 # loop/all branches probability, 0.5 by default
- number-of-loop-iterations:
 - min: 2 # 4 by default
 - max: 32 # 4 by default
- max-depth:
 - if: 500 # unlimited by default
 - loop: 4 # 4 by default
- distance:
 - blocks:
 - min: 1 # 0 by default
 - max: 20 # by default calculated, depending on max branch distance



Компиляторные идеи: пасс менеджер

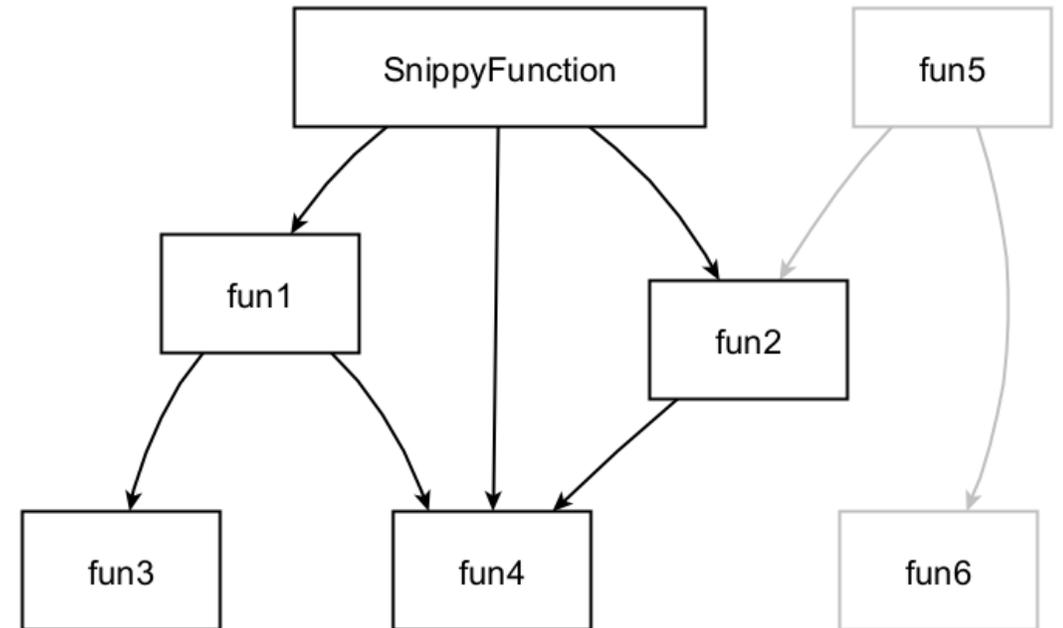
- Сначала генерируется control-flow, выдерживая инварианты **структурности**.
- При этом мы берём примерно столько бранчей, сколько было по вероятности.
- Далее генератор идёт по базовым блокам и **один за другим** наполняет их остальными инструкциями.





Граф вызовов функций (call graph)

- У llvm-snipru есть возможность формировать случайный call-graph
- Строго иерархический call graph без циклов.
- Возможность наличия недостижимых функций.
- Единственная точка уважения к ABI это внешняя функция snippets.
- Изначально у нас не было никаких требований к внутреннему ABI. Функции **свободно портят любые регистры**.
- Это вызвало определённые проблемы и в snipru была введена возможность резервации регистров.





Списки резервации и спилла

- Список `--spilled-regs-list` задаёт все регистры, которые должны быть сохранены на стек в начале snippets и восстановлены в конце.

`--spilled-regs-list=X5,X6,X7,X28`

- Список `--reserved-reg-list` задаёт регистры, которые запрещены к использованию **основными инструкциями** snippets.

`--reserved-regs-list=X1,X3,X4,X8,X9,X10,X11,X12,X13,X14,X15,X16,X17`

- Мы не запретили использование таких регистров ancillary инструкциями, потому что если заблокированы 30 из 32 регистров, может не хватить регистров на формирование адреса.
- Мы стараемся такое использование минимизировать, но сейчас весь reserve это soft-reserve.
- Фактически hard-reserve в коде генератора поддержан, **но как опция недоступен**.
- В частности по умолчанию зарезервирован (hard-резервирован) SP.



Переопределение внутреннего stack pointer

- Решение о hard-резервации SP влияло на покрытие регистров в сниппетах.
 - Было принято решение дать возможность его переопределять.
- redefine-sp=[any, SP, any-not-SP, reg::R] где SP это стек поинтер, указанный в ABI.
- Мы будем обозначать переопределённый SP как RedefSP.
 - Тогда правила следующие:
 - RedefSP = any означает любой регистр, который не резервирован, не spilled и не X0, X1, X6 для RISC-V.
 - RedefSP = any-not-SP означает то же самое и не SP.
 - X0 это hard-wired zero, X1 это адрес возврата.
 - X6 не может быть Redef-SP используется для tail-calls в chained-RX.



Режим honor-target-abi

```
./llvm-snippy -march=riscv64-unknown-elf yml/layout.yaml  
-num-instrs=1000 yml/memory-aligned.yaml -model-plugin=sail  
-entry-point=test_0 -stack-size=1024 -honor-target-abi  
-last-instr=RET
```

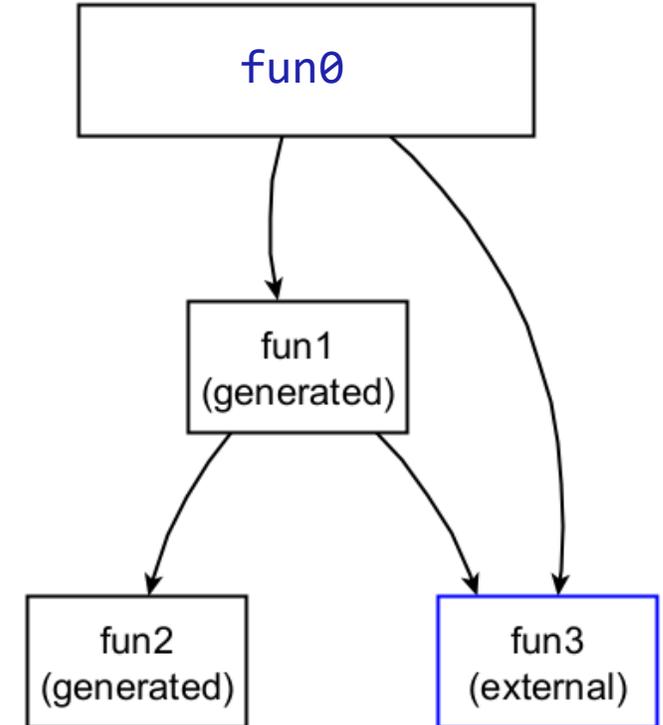
- Удобный способ определить spilled-list в соответствии с текущим ABI в LLVM.
- Здесь показано использование honor-target-abi с внутренним стеком, но вообще такой режим предполагает встраивание snippets и чаще используется с --external-stack
- При работе с honor-target=abi, опция --spilled-regs-list игнорируется.
- При других опциях (кроме honor-target-abi) мы не гарантируем, что эта функция может быть вызвана из C-кода.
- Поэтому с этим режимом не работает также redefine-SP.



Внешний call graph

- Пользователь имеет возможность не только попросить сгенерировать граф вызовов.
- Можно самостоятельно написать функцию и попросить вставить её в граф вызовов для snippets а потом слинковаться с кодом, написанным вручную.
- У функции fun3 будет C ABI.
- Если это должно работать на модели, fun3 должна быть стабилизирована.

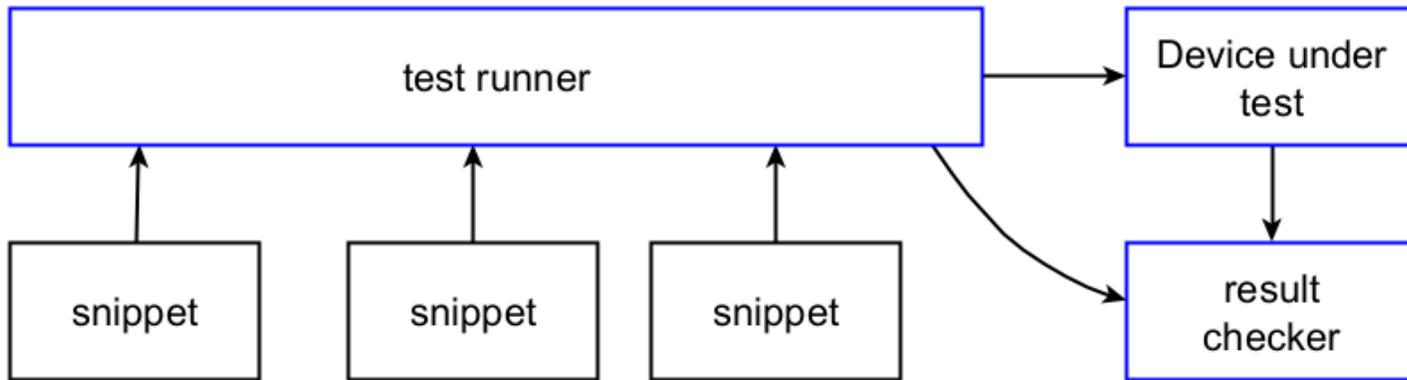
```
entry-point: fun0
function-list:
- name: fun0
  callees:
  - fun1
  - fun3
- name: fun1
  callees:
  - fun2
- name: fun3
  external: true
```





Использование llvm-snippy для учебных проектов

- Никакой ISG не является готовым решением для вашего проекта со всей его спецификой.



- В open-сорс начинают появляться готовые решения по прикручиванию llvm-snippy к системному тестбенчу (в основном интерес идёт от энтузиастов и их частных проектов).



Dmitriy Belimov • 1st
Senior Digital Circuits Engineer | Autonomous Robotics
2mo • Edited •

Good point of my pet project with own RISC-V core. Designed simple software testbed for evaluation work and data flow. How to it works:

1. The fuzzing program generator LLVM-Snippy generates testcases.
2. Spike RISC-V golden model executes the testcases with a trace file generation.
3. My core simulates in Icarus Verilog and executes the same testcases with a trace file generation.
4. Special python program compares the traces.
5. Goto step 1.

All information are stored into DB and can be viewed over simple web interface. Also I did simple Telegram bot to see its and control (start/stop) the testbed. As I see the way is very usefull for me.

I want to do 10000 tests rounds. Analyze and build result of testing. May be write a paper about the approach.

In the image you may see simple web interface of statistics and control panel of the Telegram bot.



https://github.com/pitman75/test_snippy



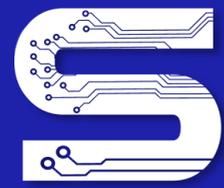
А где ещё компиляторы спешат на помощь?

- Представьте, что вам нужно:
 - Проанализировать код до его выполнения.
 - Это задача статического анализа (clang-tidy).
 - Оценить throughput / latency инструкций для платы, на которую у вас нет описания.
 - Это задача бенчмаркинга.
 - Сегодня будет доклад Анастасии Черниковой про llvm-exegesis и его применения.
 - Смоделировать конвеер исходя из модели конвейера в компиляторе (llvm-mca)

Призыв к действию: давайте создавать генератор вместе



<https://github.com/syntacore/snippy>



Syntacore™
Custom cores and tools

ВСЕМ СПАСИБО, А ТЕПЕРЬ Q&A