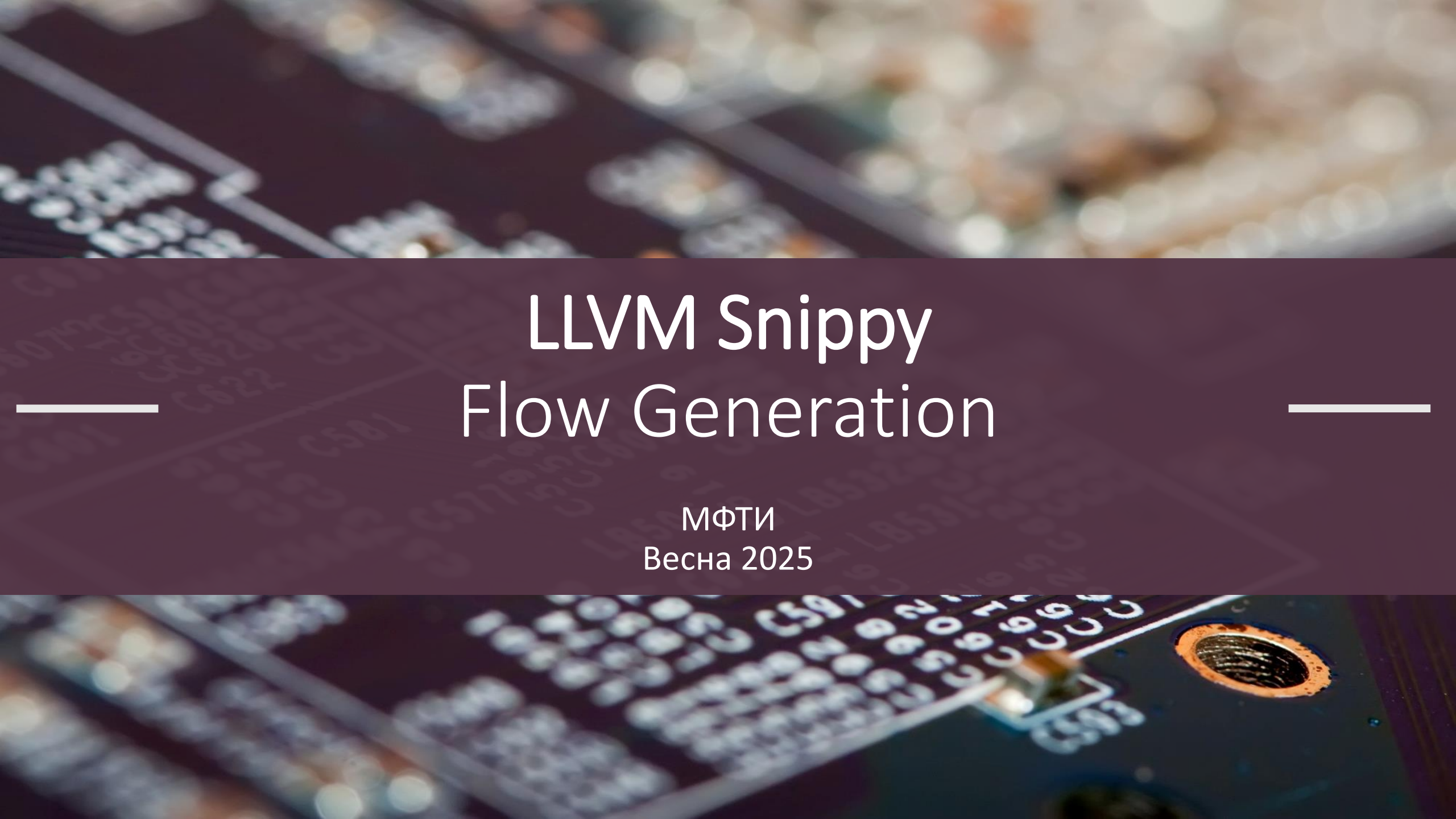


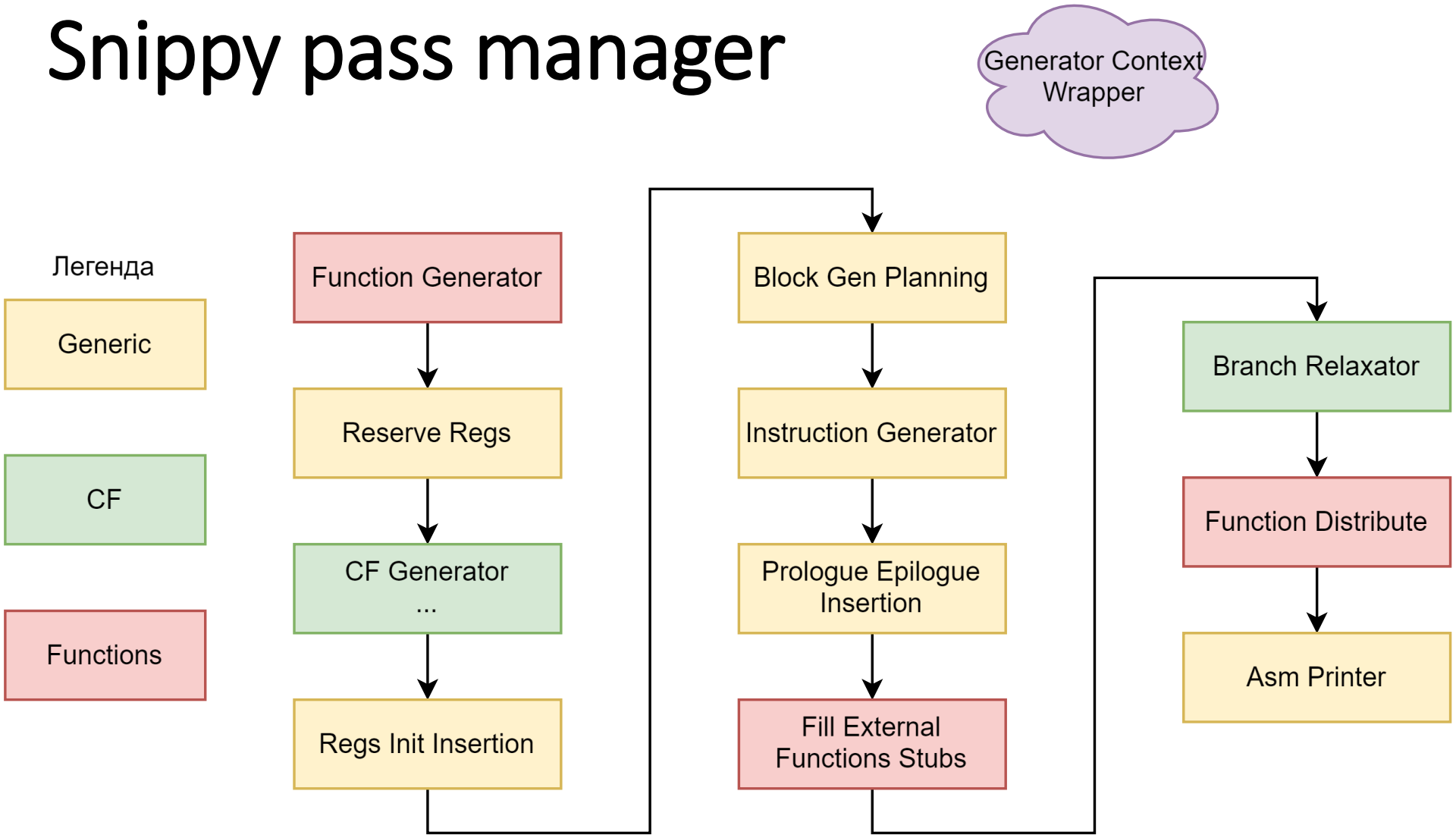
LLVM Snippy
Flow Generation

МФТИ
Весна 2025

Повторение: passes & pass manager

- Пасс – сущность работающая с кусочком IR
  - С модулем (единицей трансляции)
  - С функцией
  - С циклом
  - С базовым блоком
- Пассы или модифицируют, или анализируют IR
- Любой пасс может зависеть от любого анализа
- Pass manager управляет последовательностью трансформаций и подготавливает для них анализы

Snippy pass manager



[llvm-snippy/lib/FlowGenerator.cpp]

Passes: GeneratorContextWrapper

- Immutable pass – никогда не изменяет IR, и не инвалидируется
- Нужен для хранения GeneratorContext
- В контексте генерации хранятся:
  - Конфигурация генерации
  - Структуры данных для хранения информации между пассами
- Является анализом, используется всеми пассами snippy

llvm-snippy/lib/GeneratorContextPass.h

Passes: ReserveRegs

- В snippy реализован механизм резервации регистров:
  - Для всей генерации
  - Для выбранной функции
  - Для выбранного базового блока

- ReserveRegs резервирует sp, если включена генерация со стеком
- Один из самых ранних пассов

llvm-snippy/lib/ReserveRegsPass.cpp

Passes: RegsInitInsertion

- Генерирует начальное состояние регистров
- Вшивает инициализацию регистров в сниппет
- Использует знание LLVM о том как эффективнее всего записать значение в регистр

llvm-snippy/lib/RegsInitInsertionPass.cpp

Сложность генерации

- Snippy поддерживает разные модальные режимы генерации:
  - Случайная генерация
  - Генерация burst-групп
  - Позже могут быть добавлены другие
- Snippy поддерживает разные режимы ограничения генерации:
  - Количество инструкций
  - Размер инструкций
- Иногда эти режимы могут сложно переплетаться
- Отслеживание выполнения всех условий в момент генерации переусложняет логику

Passes: BlockGenPlanning

- Решение: планирование генерации
- Функция состоит из базовых блоков
- Базовые блоки состоят из групп инструкций
- Группа инструкций генерируется в едином режиме с фиксированным видом ограничения

llvm-snippy/lib/BlockGenPlanningPass.cpp

Passes: InstructionGenerator

- Как видно из названия – главный пасс Snippy
- Реализует обобщенную логику генерирования случайных инструкций
- Единственный пасс, использующий модель при генерации
- Генерирует по GenPlan’у

llvm-snippy/lib/FlowGenerator.cpp

Passes: InstructionGenerator

Обсуждение:

Какие вы видите проблемы в обобщенной генерации?

Passes: InstructionGenerator

Проблемы обобщенной генерации:

- Выбор инструкции для загрузки значения в память
- Генерация возврата из функции разная для разных архитектур
- Некоторые функции должны быть сгенерированы по-особенному
- Инициализация регистров в разных архитектурах должна происходить по-разному
- И прочие проблемы вызванные не обобщаемыми отличиями разных архитектур

Passes: InstructionGenerator

Проблемы обобщенной генерации:

- Выбор инструкции для загрузки значения в память
- Генерация возврата из функции разная для разных архитектур
- Некоторые функции должны быть сгенерированы по-особенному
- Инициализация регистров в разных архитектурах должна происходить по-разному
- И прочие проблемы вызванные не обобщаемыми отличиями разных архитектур

Кто виноват и что делать?

Passes: InstructionGenerator

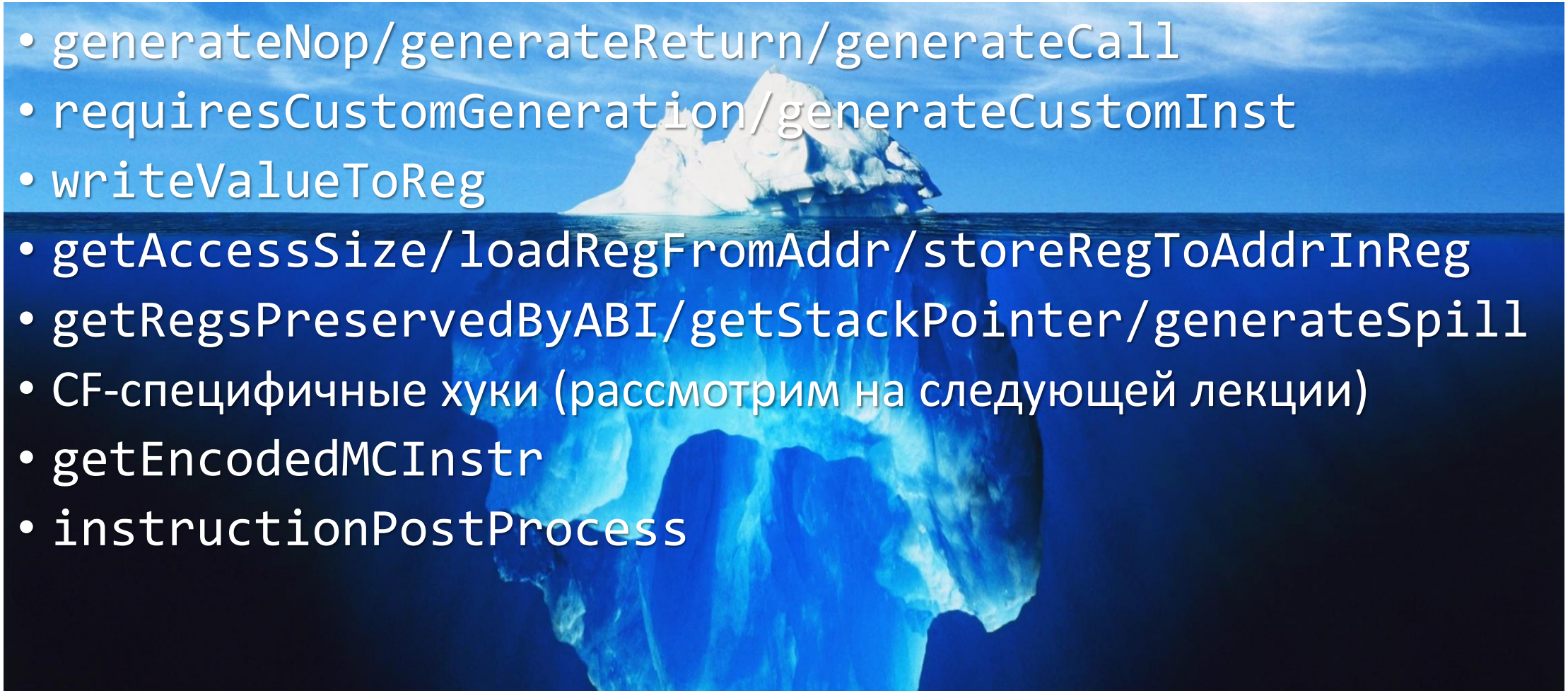
Проблемы обобщенной генерации:

- Выбор инструкции для загрузки значения в память
- Генерация возврата из функции разная для разных архитектур
- Некоторые функции должны быть сгенерированы по-особенному
- Инициализация регистров в разных архитектурах должна происходить по-разному
- И прочие проблемы вызванные не обобщаемыми отличиями разных архитектур

Решение: класс с виртуальными хуками для выполнения операций специфичных для целевой архитектуры

Snippy Target

Класс с виртуальными хуками:



- `generateNop/generateReturn/generateCall`
- `requiresCustomGeneration/generateCustomInst`
- `writeValueToReg`
- `getAccessSize/loadRegFromAddr/storeRegToAddrInReg`
- `getRegsPreservedByABI/getStackPointer/generateSpill`
- CF-специфичные хуки (рассмотрим на следующей лекции)
- `getEncodedMCInstr`
- `instructionPostProcess`

Passes: InstructionGenerator

Общий алгоритм генерации:

- Выбирается базовый блок для генерации
- Если инструкция должна быть сгенерирована особым образом, она генерируется особым образом
- Иначе инструкция может быть сгенерирована обобщенным алгоритмом:
  - Создается инструкция
  - Генерируются операнды
  - Постобрабатываются операнды
- Собирается статистика о сгенерированных инструкциях
- По собранной статистике принимается решение об остановке генерации

Passes: InstructionGenerator

Случаи для особой генерации:

- Специфичная для целевой архитектуры генерация (пока только RVV)
- Burst

Сложность генерации RVV инструкций

- В отличие от AVX, то как интерпретируется векторный регистр не закодировано в инструкции – режим нужно задавать заранее
- Случайно нужно не только генерировать RVV инструкции, но и переключать режим VPU (Vector Processing Unit)
- Для конфигурирования генерации нужен особый отдельный target-специфичный конфиг

Обсуждение: как бы вы поддержали возможность добавления target-специфичного конфига?

Сложность генерации RVV инструкций

- По умолчанию считается, что в начале каждого блока RVV режим не выставлен
- Вез RVV режима RVV инструкции не генерируются (фильтруются гистограмме)
- RVV режим выставляется специальной инструкцией (vset*)
  - Может генерироваться по гистограмме
  - Может генерироваться как служебная инструкция, если задана вероятность переключения режима
- После первого vset* в базовом блоке RVV опкоды «включаются» и генерируются по гистограмме

Burst

- **Burst группа** – группа инструкций, следующих друг за другом, которые могут «выжечь» тестируемый модуль
- Используется для инструкций обращения к памяти

Алгоритм генерации:

- Генерируется список опкодов для генерируемой пачки
- Для всех опкодов подготавливаются регистры
- Генерируются инструкции с предвыбранными операндами

Passes: PrologueEpilogueInsertion

Пролог – код в начале функции, подготавливающий стек для использования функцией

Эпилог – код высвобождающий стек в конце функции

PrologueEpilogutInsertion вставляет загрузку изменяемых функцией регистров на стек в начале этой функции, и восстанавливает их со стека в конце функции

Обсуждение: какие могут быть подводные?

llvm-snippy/lib/PrologueEpilogueInsertionPass.cpp

Passes: PrologueEpilogueInsertion

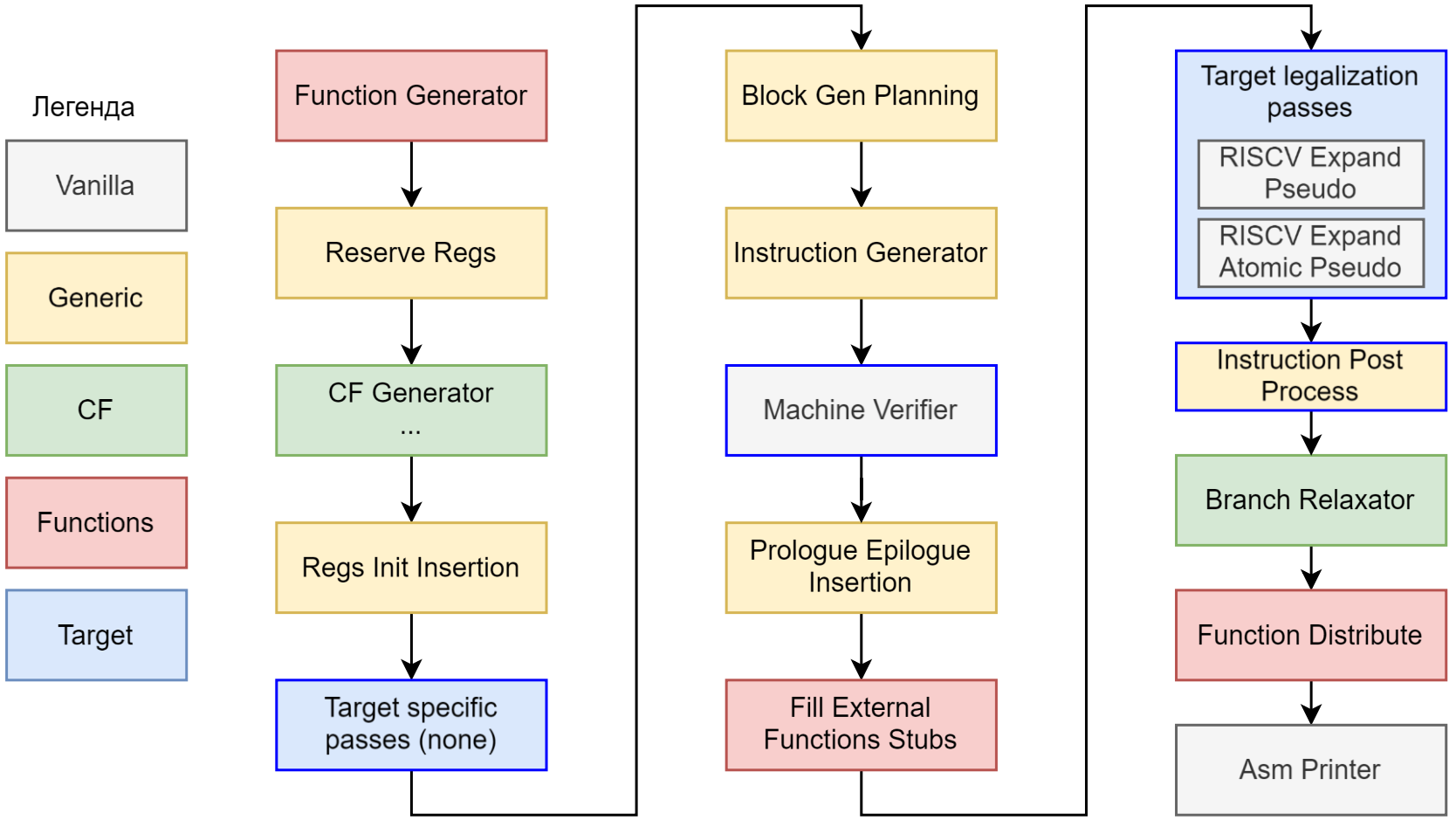
У snippy может быть два режима работы со стеком:

- Внешний стек
  - Используется стек вызывающей функции
  - Не может быть использовано с самопроверкой
  - Может быть сразу использован
- Свой стек
  - Используется отдельная указанная секция из конфига
  - Перед использование должен быть инициализирован – должен быть сохранен указатель на стек вызывающей функции, а в sp регистр записан слот на следующий свободный слот используемого стека
- В обоих случаях после инициализации стека необходимо собрать информацию об используемых в функции регистрах и сгенерировать для них spill/reload

Passes: AsmPrinter

- Ванильный пасс LLVM
- «Печатает» ассемблер
  - Может генерировать сразу объектный файл (snippy именно так и поступает)
- Конвертирует MIR сначала в MCInst (Machine Code Instruction)
  - MIR является промежуточным представлением, *похожим* на ассемблер
  - В MIR есть функции, базовые блоки, глобальные объекты
  - MCInst является удобным внутренним представлением для уже *почти* транслированного ассемблера
  - В MCInst нет функций, базовых блоков и глобальных объектов, вместо этого используются секции, метки и т.д.
- После конвертирует поток MCInst в объектный файл

Остались за кадром



+ пассы предназначенные только для дампов

To be continued …

На следующем занятии

- Узнаем как сводить не только татуировки, но и графы
- Узнаем что такое отношение доминации и как доминировать граф
- Какое отношение это вообще имеет к компиляторам
- Научимся с помощью доминации находить циклы
- Узнаем зачем все эти знания нужны в snippy для генерации потока управления