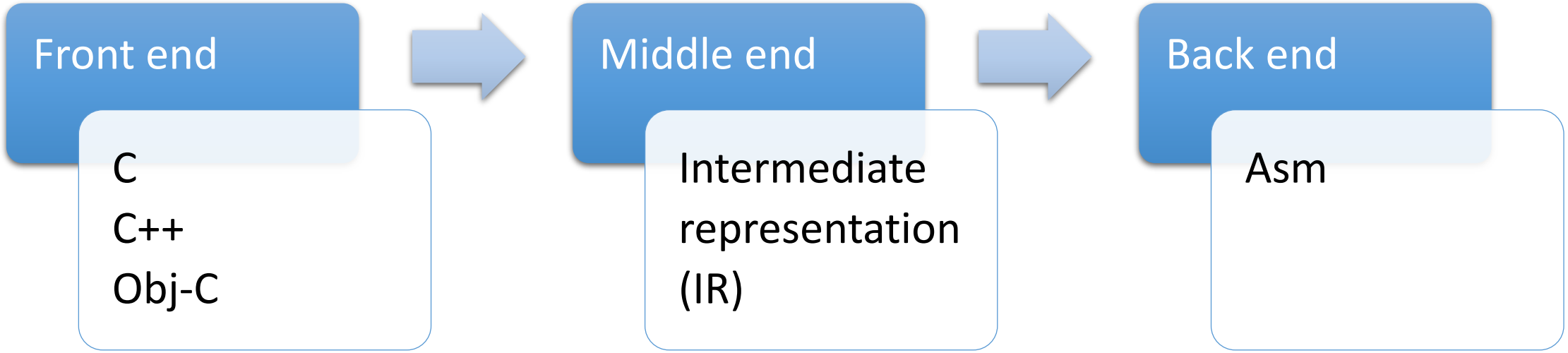
LLVM

МФТИ
Весна 2025

LLVM

- Дипломный проект Криса Латнера
- Впервые опубликован в 2003 году
- Изначально планировалось как виртуальная машина (Low Level Virtual Machine)
- Сейчас разрабатывает RISC-V backend в SiFive

Архитектура компиляторов



Обсуждение: пишем компилятор

Как бы вы реализовывали подобный проект?

Из каких библиотек он бы состоял?

Обсуждение: пишем Front end компилятора

- Front end

Обсуждение: пишем Front end компилятора

- Front end
  - Lexer
    - Выполняет лексический анализ – разбивает входной текст на поток лексем (идентификаторы, ключевые слова, литералы, операторы и разделители)

Обсуждение: пишем Front end компилятора

- Front end
  - Lexer
  - Parser
    - Выполняет синтаксический анализ потока лексем, конструирует AST

Обсуждение: пишем Front end компилятора

- Front end
  - Lexer
  - Parser
  - AST
    - Предоставляет структуру данных AST и методы работы с ней

Обсуждение: пишем Front end компилятора

- Front end
  - Lexer
  - Parser
  - AST
  - Semantic analyzer
    - Выполняет семантический разбор AST (разрешение имен, вывод типов, инстанцирование шаблонов)

Обсуждение: пишем Front end компилятора

- Front end
  - Lexer
  - Parser
  - AST
  - Semantic analyzer
  - AST to IR codegen
    - Строит IR из AST

Обсуждение: пишем Middle end компилятора

- Middle end

Обсуждение: пишем Middle end компилятора

- Middle end
  - IR (Intermediate representation)
    - Реализация IR как структуры данных + базовые операции с ним

Обсуждение: пишем Middle end компилятора

- Middle end
  - IR (Intermediate Representation)
  - IR analysis
    - Различные алгоритмы анализа IR

Обсуждение: пишем Middle end компилятора

- Middle end
  - IR (Intermediate Representation)
  - IR analysis
  - Optimizations over IR
    - Выполняет оптимизации и преобразования над IR

Обсуждение: пишем Back end компилятора

- Back end
  - Codegen
    - Генерирует ассемблер из IR. Довольно сложен, детально рассмотрим чуть поздже.

Обсуждение: пишем компилятор

- Front end
  - Lexer
  - Parser
  - AST
  - Semantic analyzer
  - AST to IR
- Middle end
  - IR
  - IR analysis
  - Optimizations over IR
- Back end
  - Codegen

Подобным образом реализованы и **GCC**, и **Clang/LLVM**

Но *разработчики* компиляторов больше любят **Clang** и **LLVM**

Почему?

Почему LLVM любят
Первая причина – это IR

LLVM IR – это не просто набор классов для IR, это полноценный самоописывающийся, строго типизированный язык, единый* на весь middle end

Gimple (GCC IR)**

```
int square (int num)
{
  int D.2744;
  int _2;

  <bb 2> :
  _2 = num_1(D) * num_1(D);

  <bb 3> :
<L0>:
  return _2;
}
```

LLVM IR

```
define i32 @square(i32 %num) {
entry:
  %num.addr = alloca i32
  store i32 %num, ptr %num.addr
  %0 = load i32, ptr %num.addr
  %1 = load i32, ptr %num.addr
  %mul = mul nsw i32 %0, %1
  ret i32 %mul
}
```

* GCC имеет несколько IR в middle end

** Строго говоря, на поздних стадиях оптимизаций некоторые конструкции становятся запрещенными, что создает неявное разделение внутри одного IR

Почему LLVM любят
Вторая причина – это лицензия

GCC распространяется под лицензией GNU GPLv3

GNU GPL требует распространения с бинарными файлами (в том числе неизменными) исходного кода или письменного обязательства его предоставить

LLVM распространяется под лицензией Apache 2.0

Apache 2.0 позволяет комбинировать открытый и закрытый код без обязательства его предоставления

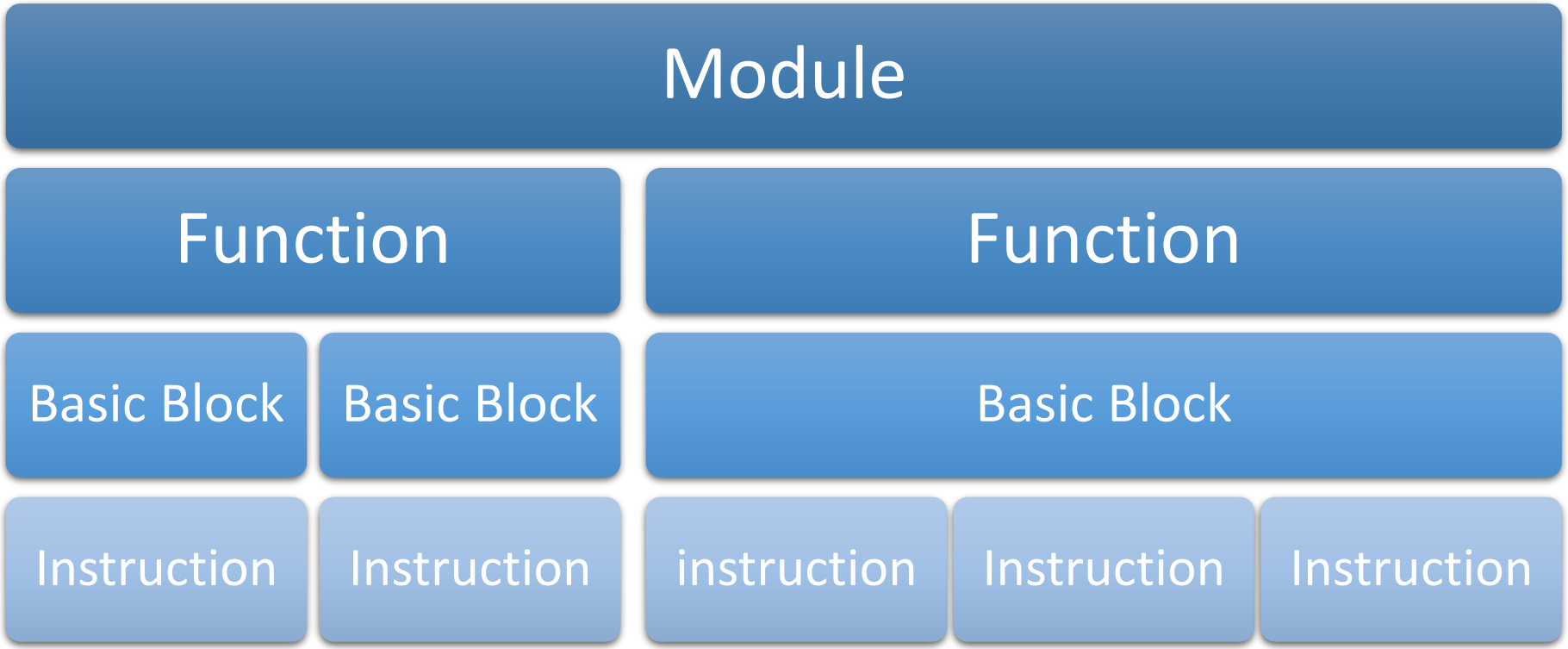
Почему LLVM любят
Третья причина – это модульность

- Языковой front end clang переиспользуется в отладчике lldb
- Описание target’a из backend’a переиспользуется в линкере lld, отладчике lldb и других проектах
- Множество clang-based тулов – clang-format, clang-tidy, clangd и т.д.

Самоописываемость IR позволяет создавать отдельные средства для работы над IR отдельно от компилятора

Почему LLVM IR так удобен

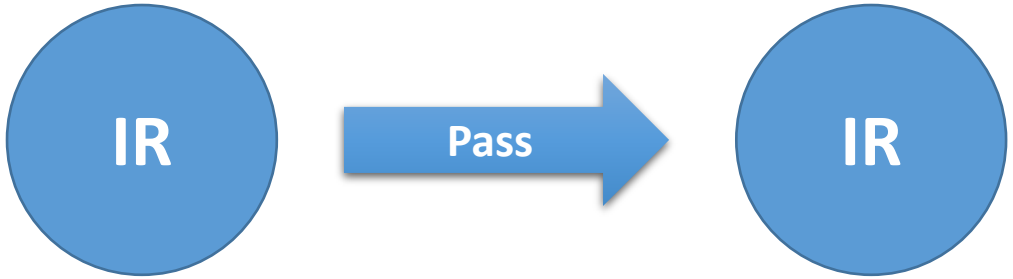
1. Четкая иерархия объектов и владения



Почему LLVM IR так удобен

2. Четкий механизм работы с IR – Passes

- Принимает на вход IR
- *Возможно* модифицирует IR
- Имеет состояние
- Может зависеть от других пассов (анализов)



Запуск всех пассов контролируется с помощью Pass Manager

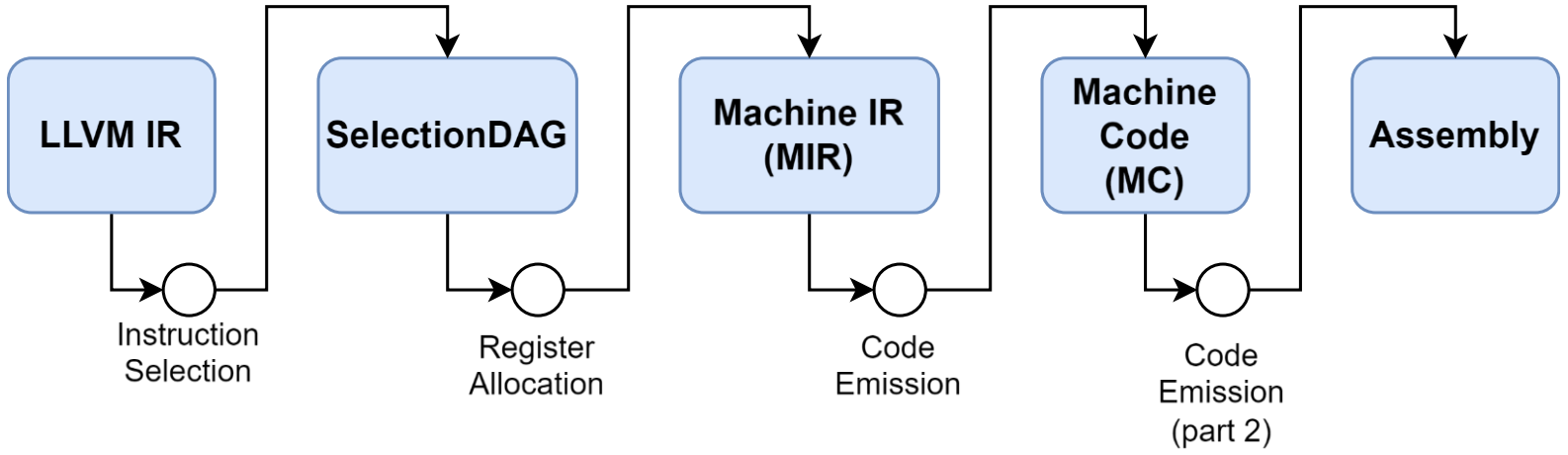
Что происходит в Back end?

До этого было вскользь упомянуто, что backend состоит из кодогенерации – процесса преобразования платформонезависимого IR в ассемблерный/объектный код целевой архитектуры

Что же там происходит?

Что происходит в Back end?

- В LLVM реализован алгоритм кодогенерации, независимый от целевой архитектуры
- Кодогенерация разделяется на несколько этапов, позволяя абстрагировать различия поддерживаемых целевых архитектур
- Благодаря чему такой подход к кодогенерации становится возможен?



Низкоуровневый IR в LLVM

Низкоуровневый IR в LLVM называется **Machine IR** (**MIR**)

➢Очень похож на ассемблер

Machine Instruction состоит из

- Опкода (target specific)
- Набора операндов
  - Регистры (как виртуальные так и физические)
  - Флаги
  - Immediate
  - Метки

Низкоуровневый IR в LLVM

Для работы с MIR есть специальная иерархия пассов:

- Module Pass
- Machine Function Pass
- Machine Basic Block Pass

To be continued …

На следующем занятии

- Рассмотрим практические аспекты проектирования LLVM как C++ фреймворка