

AI编译器系列

LLVM架构和原理



ZOMI



BUILDING A BETTER CONNECTED WORLD

Ascend & MindSpore

www.hiascend.com
www.mindspore.cn

Talk Overview

I. 传统编译器

- History of Compiler - 编译器的发展
- GCC process and principle – GCC 编译过程和原理
- LLVM/Clang process and principle – LLVM 架构和原理

2. AI编译器

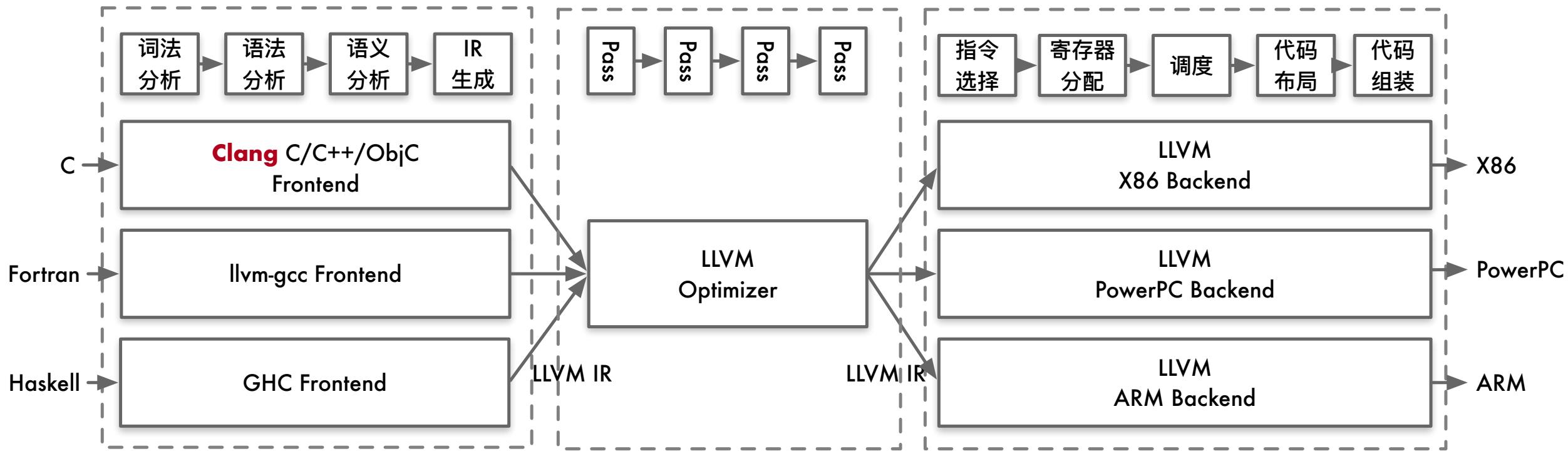
- History of AI Compiler – AI编译器的发展
- Base Common architecture – AI编译器的通用架构
- Different and challenge of the future – 与传统编译器的区别，未来的挑战与思考

Talk Overview

LLVM/Clang process and principle – LLVM 架构和原理

- LLVM 项目发展历史
- LLVM 基本设计原则和架构
- LLVM 中间表示 LLVM IR
- LLVM 前端过程
- LLVM 中间优化
- LLVM 后端生成
- 基于 LLVM 项目

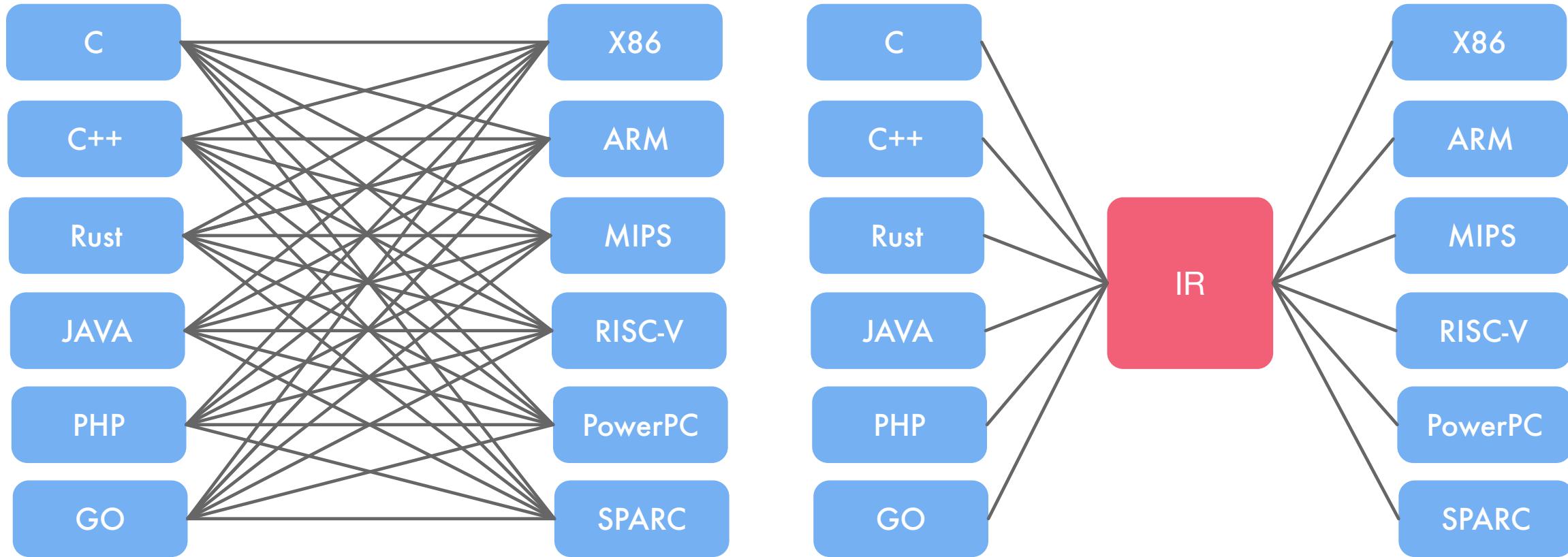
LLVM Architecture



LLVM IR

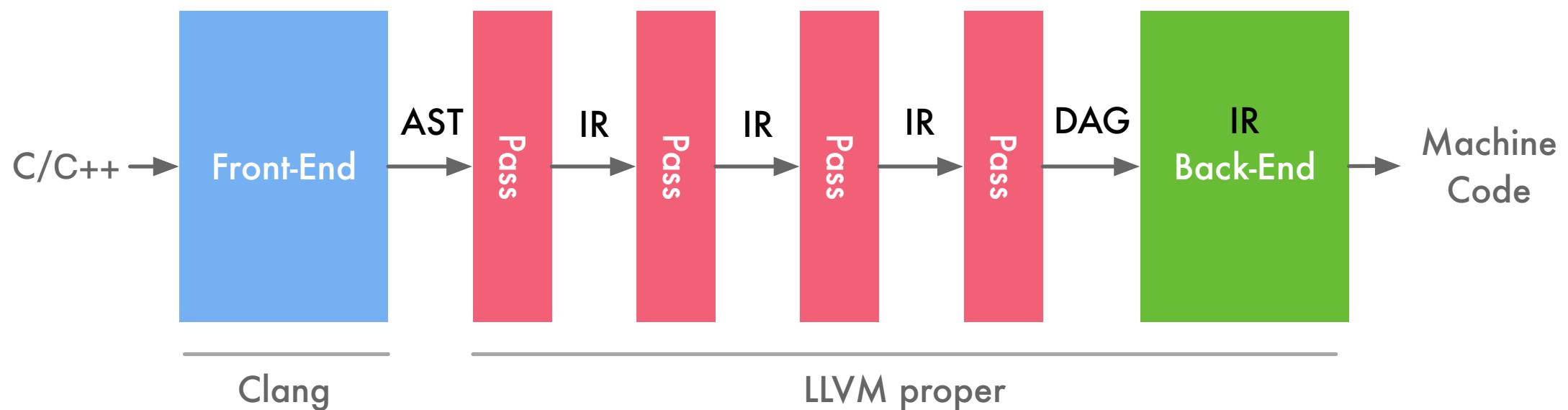
中间表达

LLVM IR



LLVM IR

这并不意味着LLVM使用单一 IR 表示方式， 在编译不同阶段会采用不同的数据结构：



LLVM IR 表示

```
1 void test(int a, int b){  
2     | int c = a * b + 100;  
3 }
```

clang -S -emit-llvm test.c

LLVM IR 表示

```
1 ; Function Attrs: noinline nounwind optnone ssp uwtable
2 define void @test(i32, i32) #2 { ; 有个全局函数@test (a,b)
3   %3 = alloca i32, align 4 ; 局部变量 c
4   %4 = alloca i32, align 4 ; 局部变量 d
5   %5 = alloca i32, align 4 ; 局部变量 e
6   store i32 %0, i32* %3, align 4 ; %0 赋值给%3 c = a
7   store i32 %1, i32* %4, align 4 ; %1 赋值给%4 d = b
8   %6 = load i32, i32* %3, align 4 ; 读取%3,赋值给%6 就是函数参数a
9   %7 = load i32, i32* %4, align 4 ; 读取%4,赋值给%7 就是函数参数b
10  %8 = mul nsw i32 %6, %7 ; a * b
11  %9 = add nsw i32 %8, 100 ; a * b + 100
12  store i32 %9, i32* %5, align 4 ; 参数 %9 赋值给 %5 e ===> 就是转换前函数写的int c变量
13  ret void
14 }
```

LLVM IR基本语法

1. 注释以;开头
2. 全局表示以@开头，局部变量以%开头
3. alloca在函数栈帧中分配内存
4. i32 32位 4个字节的意思
5. align 字节对齐
6. store写入
7. load读取

LLVM IR

LLVM IR 作为一种编译器 IR，它的两个基本原则指导着核心库的开发：

- SSA 表示，代码组织为三地址指令序列和无限寄存器让优化能够快速执行。
- 整个程序的 IR 存储到磁盘让链接时优化易于实现。

LLVM IR

LLVM IR 采用静态单赋值形式（ Static single assignment , SSA ），具有两个重要特征：

- 代码组织为三地址指令序列
- 寄存器数量无限制

What is SSA(Static Single Assignment) 静态单赋值

当程序中的每个变量都有且只有一个赋值语句时，称一个程序是 SSA 形式的。LLVM IR 中，每个变量都在使用前都必须先定义，且每个变量只能被赋值一次。以 $1 * 2 + 3$ 为例：

```
1 %0 = mul i32 1, 2
2 %0 = add i32 %0, 3
3 ret i32 %0
```

```
1 %0 = mul i32 1, 2
2 %1 = add i32 %0, 3
3 ret i32 %1
```

每个值只有单一赋值定义了它。每次使用一个值，可以立刻向后追溯到给出其定义的唯一的指令。极大简化优化，因为SSA形式建立了平凡的use-def链，也就是一个值到达使用之处的定义的列表。

LLVM IR基本语法

- LLVM IR 是类似于精简指令集（ RISC ）的底层虚拟指令集；
- 和真实精简指令集一样，支持简单指令的线性序列，例如添加、相减、比较和分支；
- 指令都是三地址形式，它们接受一定数量的输入然后在不同的寄存器中存储计算结果；
- 与大多数精简指令集不同， LLVM 使用强类型的简单类型系统，并剥离了机器差异；
- LLVM IR 不使用固定的命名寄存器，它使用以 % 字符命名的临时寄存器；

三地址码

三地址码指令 Three-address code - Wikipedia

每个三地址码指令，都可以被分解为一个四元组（4-tuple）的形式：（运算符，操作数1，操作数2，结果），由于每个陈述都包含了三个变量，即每条指令最多有三个操作数，所以它被称为三地址码。

| 指令类型 | 指令形式 | 四元组表示 |
|------|---------------------------------------|---------------|
| 赋值指令 | $z = x \text{ op } y$ ($z = x + y$) | (op, x, y, z) |

LLVM IR 表示形式

LLVM IR 具有三种表示形式，这三种中间格式是完全等价的：

- 在内存中的编译中间语言（无法通过文件的形式得到的指令类等）
- 在硬盘上存储的二进制中间语言（格式为 .bc ）
- 人类可读的代码语言（格式为 .ll ）

LLVM IR 内存模型

- LLVM IR 文件的基本单位称为 module
- 一个 module 中可以拥有多个顶层实体，比如 function 和 global variavle
- 一个 function define 中至少有一个 basicblock
- 每个 basicblock 中有若干 instruction，并且都以 terminator instruction 结尾

LLVM IR 内存模型

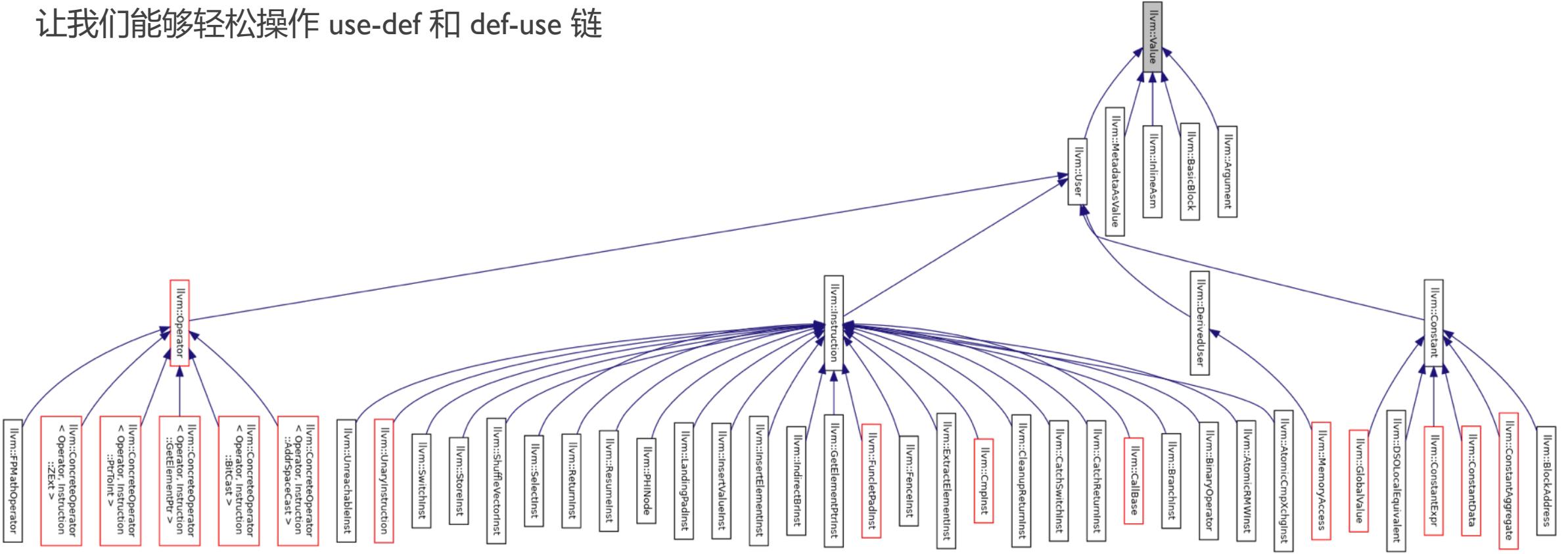
```
1 ; Function Attrs: noinline nounwind optnone ssp uwtable
2 define void @test(i32, i32) #2 { ; 有个全局函数@test (a,b)
3   %3 = alloca i32, align 4 ; 局部变量 c
4   %4 = alloca i32, align 4 ; 局部变量 d
5   %5 = alloca i32, align 4 ; 局部变量 e
6   store i32 %0, i32* %3, align 4 ; %0 赋值给%3 c = a
7   store i32 %1, i32* %4, align 4 ; %1 赋值给%4 d = b
8   %6 = load i32, i32* %3, align 4 ; 读取%3,赋值给%6 就是函数参数a
9   %7 = load i32, i32* %4, align 4 ; 读取%4,赋值给%7 就是函数参数b
10  %8 = mul nsw i32 %6, %7 ; a * b
11  %9 = add nsw i32 %8, 100 ; a * b + 100
12  store i32 %9, i32* %5, align 4 ; 参数 %9 赋值给 %5 e ===> 就是转换前函数写的int c变量
13  ret void
14 }
```

LLVM IR 内存模型

| | |
|--------------------|---|
| Module | Module类聚合了整个翻译单元用到的所有数据，它是LLVM术语中的“module”的同义词。它声明了Module::iterator typedef，作为遍历这个模块中的函数的简便方法。你可以用begin()和end()方法获取这些迭代器。 |
| Function | Function类包含有关函数定义和声明的所有对象。对于声明来说（用isDeclaration()检查它是否为声明），它仅包含函数原型。无论定义或者声明，它都包含函数参数的列表，可通过getArgumentList()方法或者arg_begin()和arg_end()这对方法访问它。你可以通过Function::arg_iterator typedef遍历它们。如果Function对象代表函数定义，你可以通过这样的语句遍历它的内容：for (Function::iterator i = function.begin(), e = function.end(); i != e; ++i)，你将遍历它的基本块。 |
| BasicBlock | BasicBlock类封装了LLVM指令序列，可通过begin()/end()访问它们。你可以利用getTerminator()方法直接访问它的最后一条指令，你还可以用一些辅助函数遍历CFG，例如通过getSinglePredecessor()访问前驱基本块，当一个基本块有单一前驱时。然而，如果它有多个前驱基本块，就需要自己遍历前驱列表，这也不难，你只要逐个遍历基本块，查看它们的终结指令的目标基本块。 |
| Instruction | Instruction类表示LLVM IR的运算原子，一个单一的指令。利用一些方法可获得高层级的断言，例如isAssociative()，isCommutative()，isIdempotent()，和isTerminator()，但是它的精确的功能可通过getOpcode()获知，它返回llvm::Instruction枚举的一个成员，代表了LLVM IR opcode。可通过op_begin()和op_end()这对方法访问它的操作数，它从User超类继承得到。 |

LLVM IR 内存模型最重要概念：Value, Use, User

让我们能够轻松操作 use-def 和 def-use 链

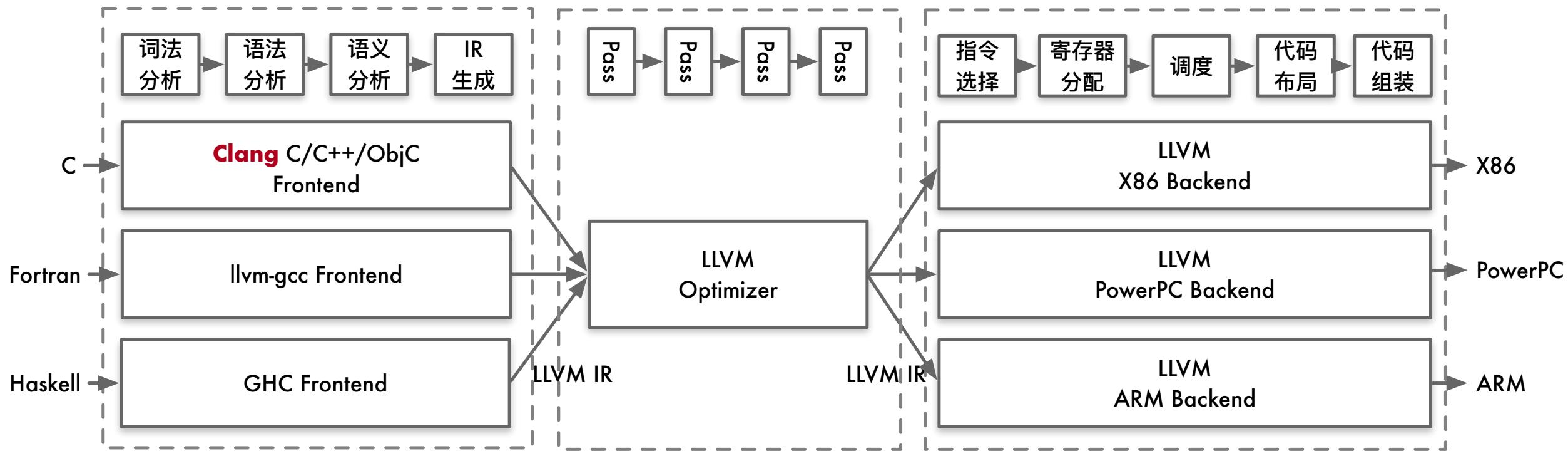


https://buaa-se-compiling.github.io/miniSysY-tutorial/pre/design_hints.html

LLVM 前端

LLVM Architecture

编译器前端将源代码变换为编译器的中间表示 LLVM IR，它处于代码生成之前，后者是针对具体目标的。



Lexical analysis 词法分析

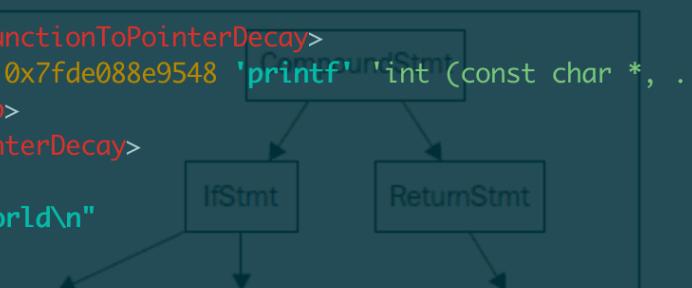
- 前端的第一个步骤处理源代码的文本输入，将语言结构分解为一组单词和标记，去除注释、空白、制表符等。每个单词或者标记必须属于语言子集，语言的保留字被变换为编译器内部表示。

```
int 'int'          [StartOfLine] Loc=<hello.c:5:1>
identifier 'main'   [LeadingSpace] Loc=<hello.c:5:5>
l_paren '('        Loc=<hello.c:5:9>
void 'void'         Loc=<hello.c:5:10>
r_paren ')'        Loc=<hello.c:5:14>
l_brace '{'        Loc=<hello.c:5:15>
identifier 'printf' [StartOfLine] [LeadingSpace] Loc=<hello.c:6:5>
l_paren '('        Loc=<hello.c:6:11>
l_paren '('        Loc=<hello.c:6:12 <Spelling=hello.c:3:19>>
string_literal '"hello world\n"' Loc=<hello.c:6:12 <Spelling=hello.c:3:20>>
r_paren ')'        Loc=<hello.c:6:12 <Spelling=hello.c:3:35>>
r_paren ')'        Loc=<hello.c:6:21>
semi ';'           Loc=<hello.c:6:22>
return 'return'    [StartOfLine] [LeadingSpace] Loc=<hello.c:7:5>
numeric_constant '0' [LeadingSpace] Loc=<hello.c:7:12>
semi ';'           Loc=<hello.c:7:13>
r_brace '}'        [StartOfLine] Loc=<hello.c:8:1>
eof '所有标签'     Loc=<hello.c:8:2>
```

Syntactic analysis 语法分析

- 分组标记以形成表达式、语句、函数体等。检查一组标记是否有意义，考虑代码物理布局，未分析代码的意思，就像英语中的语法分析，不关心你说了什么，只考虑句子是否正确，并输出语法树（AST）。

```
| -ParmVarDecl 0x7fde088fb320 <col:52> col:58 'size_t':>'unsigned long', col:11> a 'int'
| -ParmVarDecl 0x7fde088fb3a0 <line:62:7, col:18> col:30 'const char,* restrict' 'int'
`-ParmVarDecl 0x7fde088fb418 <col:32> col:39 'struct __va_list_tag *':>'struct __va_list_tag *'
-FunctionDecl 0x7fde088cb270 <line:70:1, line:71:40> line:70:12 __vsnprintf_chk 'int (char *restrict, size_t, int, size_t, const ch
第5章 LLVM中间表示
|-ParmVarDecl 0x7fde088fb650 <col:29, col:34> col:46 'char *restrict'
|-ParmVarDecl 0x7fde088fb6c8 <col:48> col:54 'size_t':>'unsigned long'
|-ParmVarDecl 0x7fde088fb748 <col:56> col:59 'int'
|-ParmVarDecl 0x7fde088cb000 <col:61> col:67 'size_t':>'unsigned long'
|-CompoundStmt 声明包含了其它的语句和表达式。下图是AST的图形视图, 可用下面的命令得到:
|-ParmVarDecl 0x7fde088cb080 <line:71:8, col:19> col:31 'const char *restrict'
`-ParmVarDecl 0x7fde088cb0f8 <col:33> col:40 'struct __va_list_tag *':>'struct __va_list_tag *'
-FunctionDecl 0x7fde088cb3e0 <hello.c:5:1, line:8:1> line:5:5 main 'int (void)'
第6章 语义分析和优化
CompoundStmt 0x7fde088cb608 <col:15, line:8:1>
|-CallExpr 0x7fde088cb580 <line:6:5, col:21> 'int'
| |-ImplicitCastExpr 0x7fde088cb568 <col:5> 'int (*) (const char *, ...)' <FunctionToPointerDecay>
| | `DeclRefExpr 0x7fde088cb480 <col:5> 'int (const char *, ...)' Function 0x7fde088e9548 printf 'int (const char *, ...)'
| |-ImplicitCastExpr 0x7fde088cb5c0 <line:3:19, col:35> 'const char *' <NoOp>
| `ImplicitCastExpr 0x7fde088cb5a8 <col:19, col:35> 'char *' <ArrayToPointerDecay>
| `ParensExpr 0x7fde088cb500 <col:19, col:35> 'char [13]' lvalue
| `StringLiteral 0x7fde088cb4d8 <col:20> 'char [13]' lvalue "hello world\n"
`-ReturnStmt 0x7fde088cb5f8 <line:7:5, col:12>
`-IntegerLiteral 0x7fde088cb5d8 <col:12> 'int' 0
```



Semantic analysis 语义分析

- 借助符号表检验代码没有违背语言类型系统。符号表存储标识符和其各自的类型之间的映射，以及其它内容。类型检查的一种直觉的方法是，在解析之后，遍历AST的同时从符号表收集关于类型的信息。

```
3 #define HELLOWORLD ("hello world\n")
4
5 int a[4];
6 int a[5];
7
```

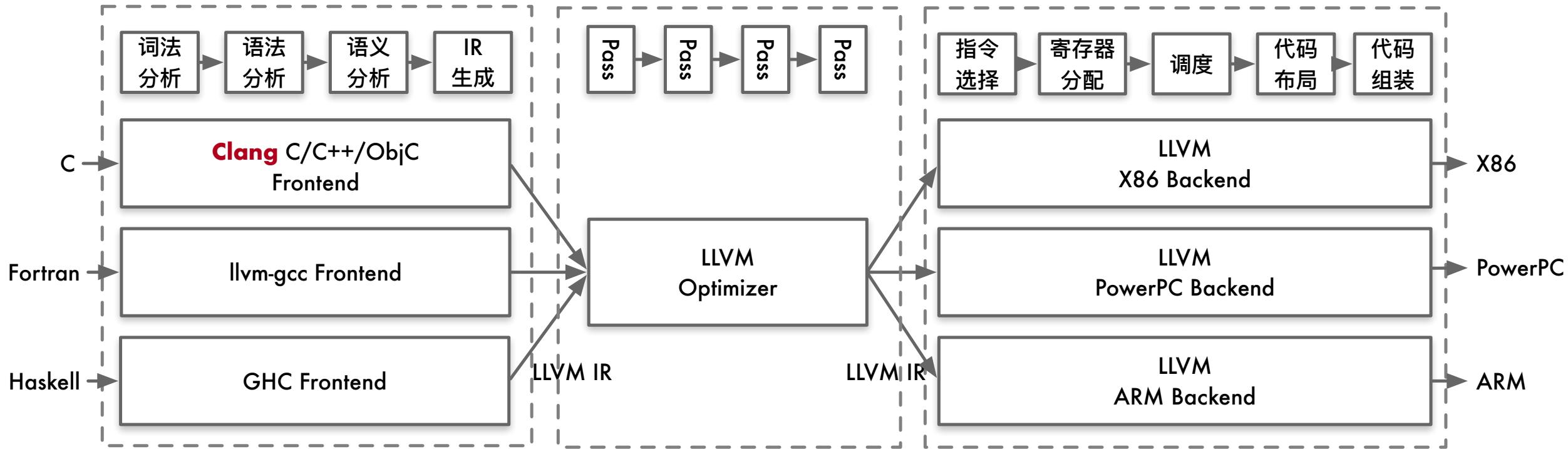
```
hello.c:6:5: error: redefinition of 'a' with a different type: 'int [5]' vs 'int [4]'
int a[5];
^
hello.c:5:5: note: previous definition is here
int a[4];
^
1 error generated.
```

\$ clang -c hello.c

LLVM 优化层

LLVM Architecture

目标无关优化，理解优化操作，实际上就是理解 IR 如何在 pass 流水线中被修改，这需要知道每个 pass 执行的修改，还有各个 pass 是以什么顺序被执行。



Finding Pass

优化通常由分析 Pass 和转换 Pass 组成。

- 分析 Pass：负责发掘性质和优化机会；
- 转换 Pass：生成必需的数据结构，后续为后者所用；

```
====  
 26 Finding Pass https://llvm.org/docs/Passes.html  
 27 ... Pass execution timing report ...  
====  
 Total Execution Time: 0.0003 seconds (0.0003 wall clock)  
  
 27 ---User Time--- --System Time-- --User+System-- ---Wall Time--- ---Instr--- --- Name ---  
 0.0001 ( 72.7%) 0.0000 ( 68.2%) 0.0002 ( 71.6%) 0.0002 ( 70.1%) 481094 BitcodeWriterPass  
 ★ 0.0000 ( 11.7%) 0.0000 ( 13.6%) 0.0000 ( 12.2%) 0.0000 ( 12.7%) 127722 VerifierPass  
 0.0000 ( 5.4%) 0.0000 ( 12.1%) 0.0000 ( 7.0%) 0.0000 ( 7.3%) 78056 RequireAnalysisPass<llvm::DominatorTreeAnalysis, llvm::Function>  
 0.0000 ( 7.8%) 0.0000 ( 3.0%) 0.0000 ( 6.6%) 0.0000 ( 6.8%) 56681 VerifierAnalysis  
 0.0000 ( 2.4%) 0.0000 ( 3.0%) 0.0000 ( 2.6%) 0.0000 ( 3.2%) 33040 DominatorTreeAnalysis  
 0.0002 (100.0%) 0.0001 (100.0%) 0.0003 (100.0%) 0.0003 (100.0%) 776593 Total  
  
 28 Finding Pass https://llvm.org/docs/Passes.html  
 29 LLVM IR Parsing  
====  
 Total Execution Time: 0.0003 seconds (0.0003 wall clock)  
 $ opt hello.bc -instcount -time-passes -domtree -o hello-tmp.bc -stats  
  
 29 ---User Time--- --System Time-- --User+System-- ---Wall Time--- ---Instr--- --- Name ---  
 0.0003 (100.0%) 0.0001 (100.0%) 0.0003 (100.0%) 0.0003 (100.0%) 869304 Parse IR  
 ★ 0.0003 (100.0%) 0.0001 (100.0%) 0.0003 (100.0%) 0.0003 (100.0%) 869304 Total
```

\$ opt hello.bc -instcount -time-passes -domtree -o hello-tmp.bc -stats

Finding Pass

<https://llvm.org/docs/Passes.html>

优化通常由分析 Pass 和转换 Pass 组成。

- **分析 Pass**：负责发掘性质和优化机会；
- **转换 Pass**：生成必需的数据结构，后续为后者所用；

- [-adce:Aggressive Dead Code Elimination](#)

积极的死代码消除。此pass类似于DCE，但它假定值是死的，除非得到其他证明。这类似于SCCP，除了用于值的活动性。

- [-constmerge: Merge Duplicate Global Constants](#)

将重复的全局常量合并到一个共享的常量中。这是有用的，一些passes在程序中插入许多字符串常量，不管现有字符串是否可用。

Understand Pass Relation

在转换Pass和分析Pass之间，有两种主要的依赖类型：

- **显式依赖**：转换Pass需要一种分析，则Pass管理器自动地安排它所依赖的分析Pass在它之前运行；

```
DominatorTree &DT = getAnalysis<DominatorTree>(Func);
```

- **隐式依赖**：转换或者分析Pass要求IR代码运用特定表达式。需要手动地以正确的顺序把这个Pass加到Pass队列中，通过命令行工具（clang或者opt）或者Pass管理器。

Pass API

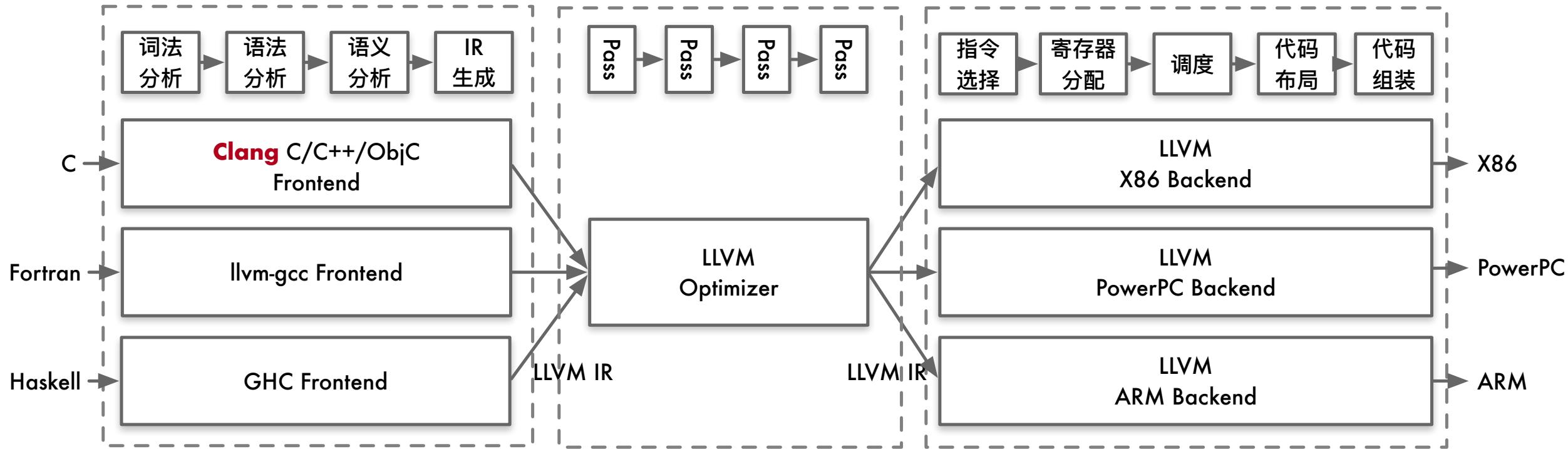
Pass类是实现优化的主要资源。然而，我们从不直接使用它，而是通过清楚的子类使用它。当实现一个Pass时，你应该选择适合你的Pass的最佳粒度，适合此粒度的最佳子类，例如基于函数、模块、循环、强联通区域，等等。常见的这些子类如下：

- ModulePass
- FunctionPass
- BasicBlockPass

LLVM 后端

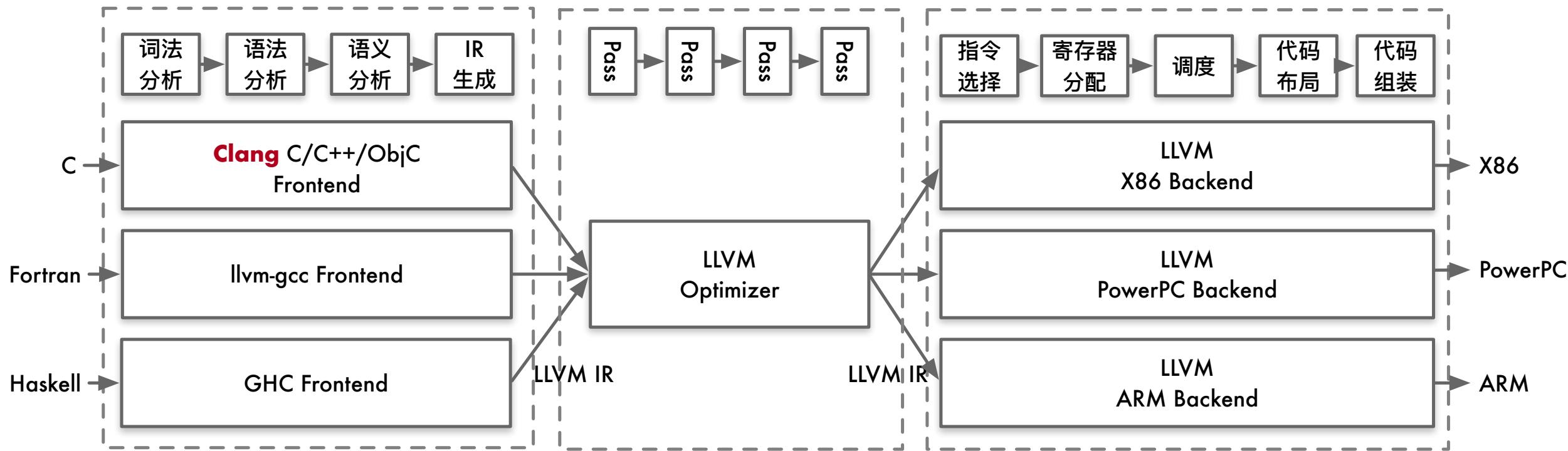
LLVM Architecture

后端由一套分析和转换 Pass 组成，它们的任务是代码生成，即将 LLVM IR 变换为目标代码（或者汇编）



LLVM Architecture

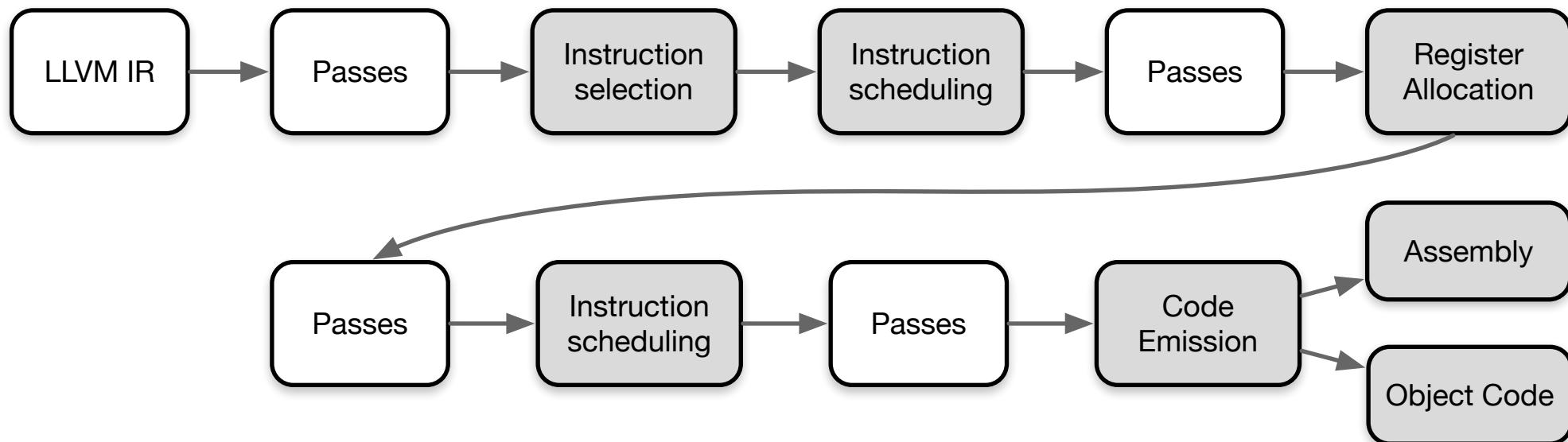
后端由一套分析和转换 Pass 组成，它们的任务是代码生成，即将 LLVM IR 变换为目标代码（或者汇编）



LLVM Backend Pass

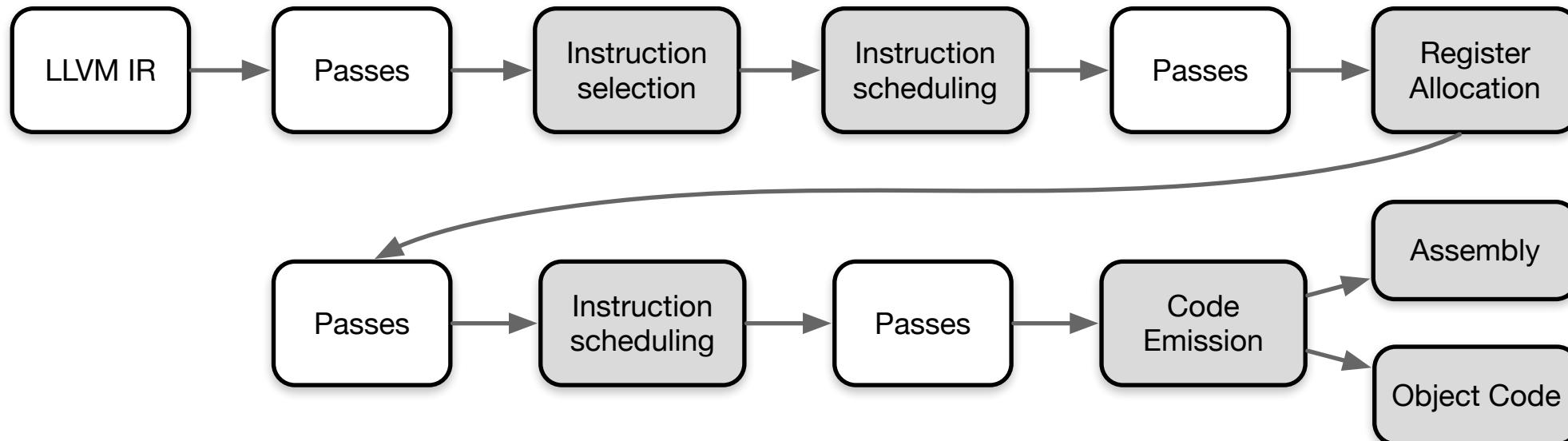
整个后端流水线用到了四种不同层次的指令表示：

- 内存中的 LLVM IR , SelectionDAG 节点 , MachineInstr , 和 MCInst。



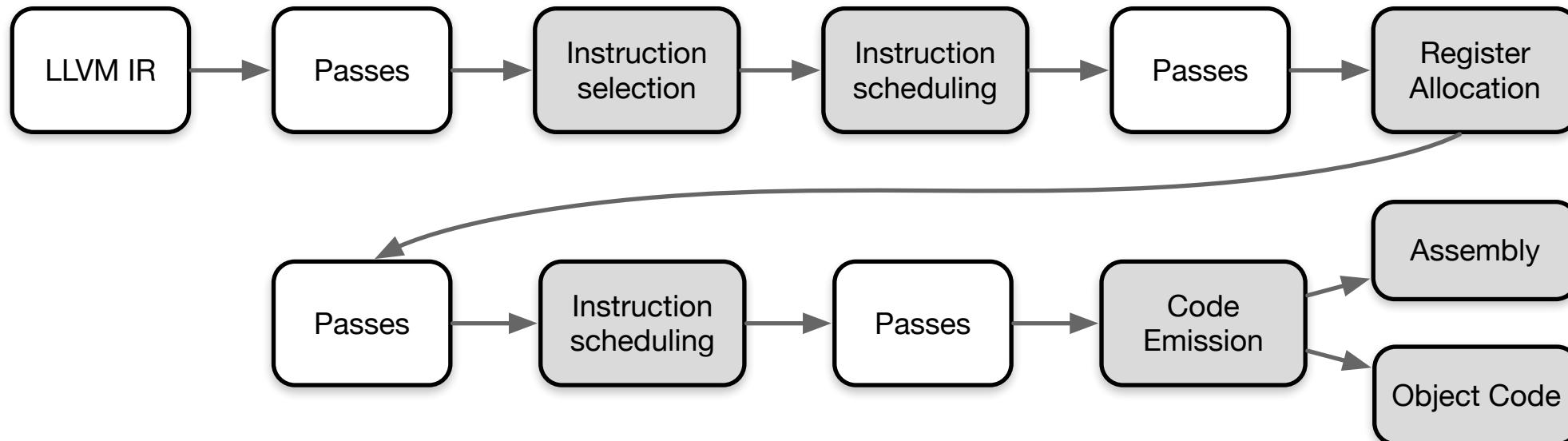
Instruction Selection 指令选择

- 内存中 LLVM IR 变换为目标特定 SelectionDAG 节点；
- 每个DAG能够表示单一基本块的计算；
- 节点表示指令，而边编码了指令间的数据流依赖；
- 让LLVM代码生成程序库能够运用基于树的模式匹配指令选择算法。**



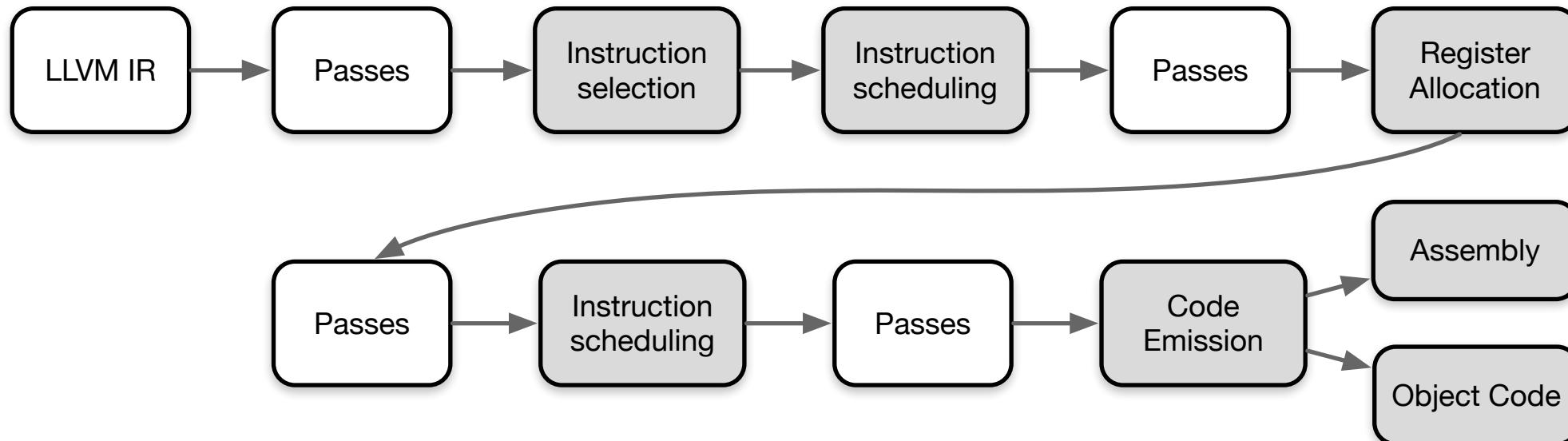
Instruction Scheduling 指令调度

- 第1次指令调度（ Instruction Scheduling ），也称为前寄存器分配（ RA ）调度；
- 对指令排序，同时尝试发现尽可能多的指令层次的并行；
- 然后指令被变换为MachineInstr三地址表示。



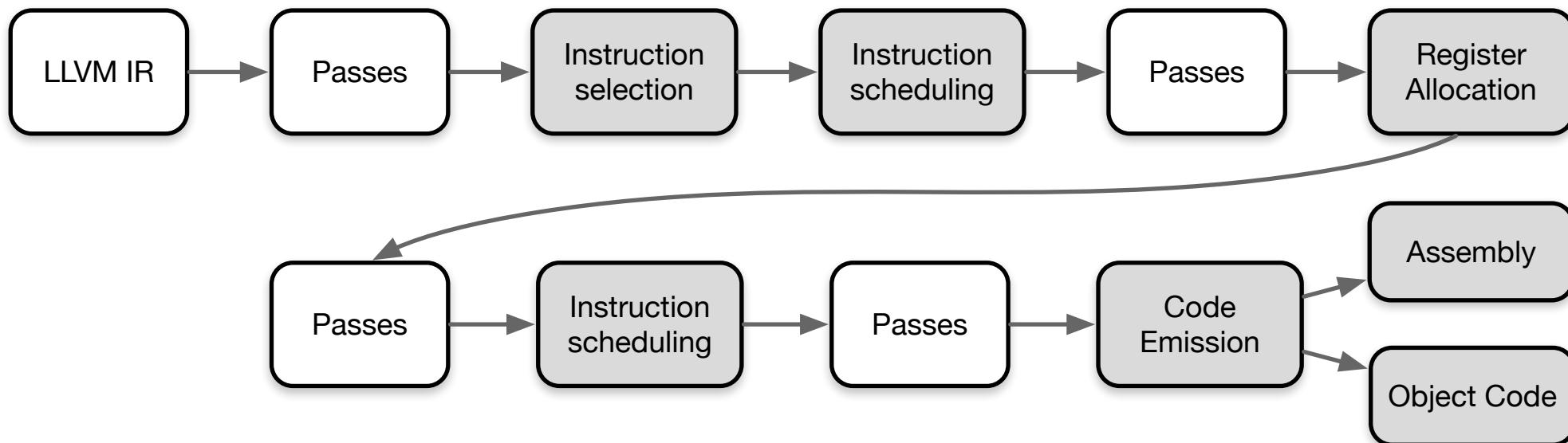
Register Allocation 寄存器分配

- LLVMIR 两个中哟特性之一：LLVM IR 寄存器集是无限；
- 这个性质一直保持着，直到寄存器分配（ Register Allocation ）；
- 寄存器分配将无限的虚拟寄存器引用转换为有限的目标特定的寄存器集；
- 寄存器不够时挤出（ spill ）到内存。



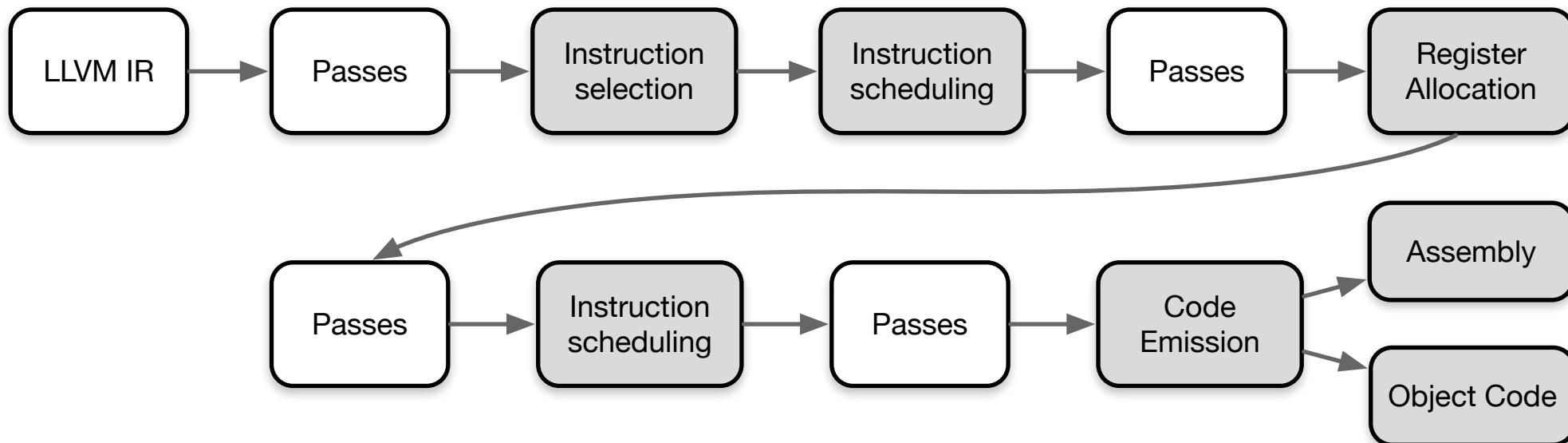
Instruction Scheduling 指令调度

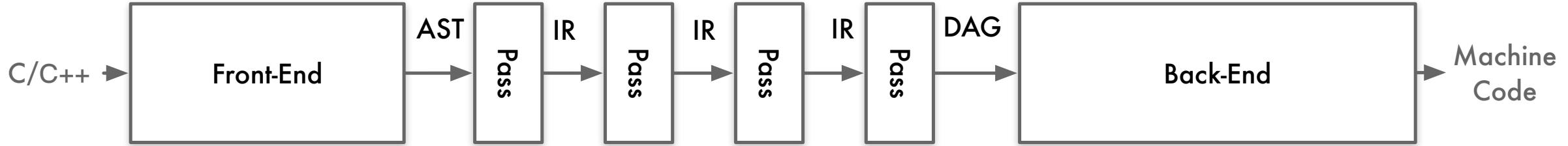
- 第2次指令调度，也称为后寄存器分配（ RA ）调度；
- 此时可获得真实的寄存器信息，某些类型寄存器存在延迟，它们可被用以改进指令顺序。



Code Emission 代码输出

- 代码输出阶段将指令从 MachineInstr 表示变换为 MCInst 实例；
- 新的表示更适合汇编器和链接器，可以输出汇编代码或者输出二进制块特定目标代码格式。





```

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

int loop_add(int* data, int num) {
    int res = 0;
    for (int i = 0; i < num; i++) {
        res += data[i];
    }
    return res;
}

define dso_local i32 @loop_add(i32*, i32) #0 {
    %3 = alloca i32*, align 8
    %4 = alloca i32, align 4
    %5 = alloca i32, align 4
    %6 = alloca i32, align 4
    store i32* %0, i32** %3, align 8
    store i32 %1, i32* %4, align 4
    store i32 0, i32* %5, align 4
    store i32 0, i32* %6, align 4
    br label %7

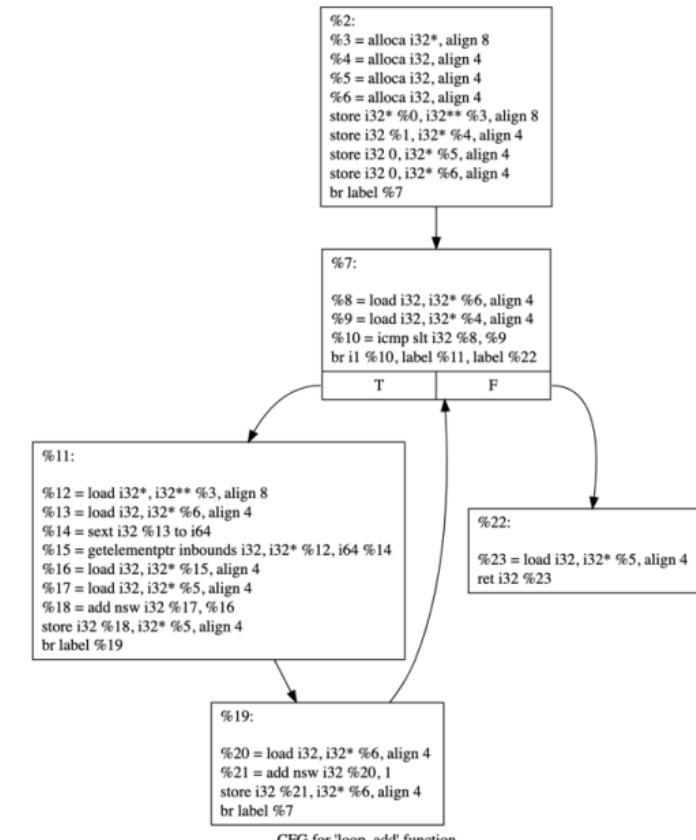
    ; <label>:7:
    %8 = load i32, i32* %6, align 4
    %9 = load i32, i32* %4, align 4
    %10 = icmp slt i32 %8, %9
    br i1 %10, label %11, label %22

    ; <label>:11:
    %12 = load i32*, i32** %3, align 8
    %13 = load i32, i32* %6, align 4
    %14 = sext i32 %13 to i64
    %15 = getelementptr inbounds i32, i32* %12, i64 %14
    %16 = load i32, i32* %15, align 4
    %17 = load i32, i32* %5, align 4
    %18 = add nsw i32 %17, %16
    store i32 %18, i32* %5, align 4
    br label %19

    ; <label>:19:
    %20 = load i32, i32* %6, align 4
    %21 = add nsw i32 %20, 1
    store i32 %21, i32* %6, align 4
    br label %7

    ; <label>:22:
    %23 = load i32, i32* %5, align 4
    ret i32 %23
}

```



基于LLVM的项目

LLVM之父Chris Lattner：编译器的黄金时代



ASPLOS Keynote: The Golden Age of Compiler Design in an Era of HW/SW Co-design by Dr. Chris Lattner

2.7万次观看 · 1年前



SiFiveInc

This week at the ASPLOS 2021 conference, Dr. Chris Lattner gave the keynote address to open the event with a discussion of the ...



A New Golden Age for Computer Architecture John L. Hennessy, David A. Patterson June 2018 End o... 22 个章节 ▾

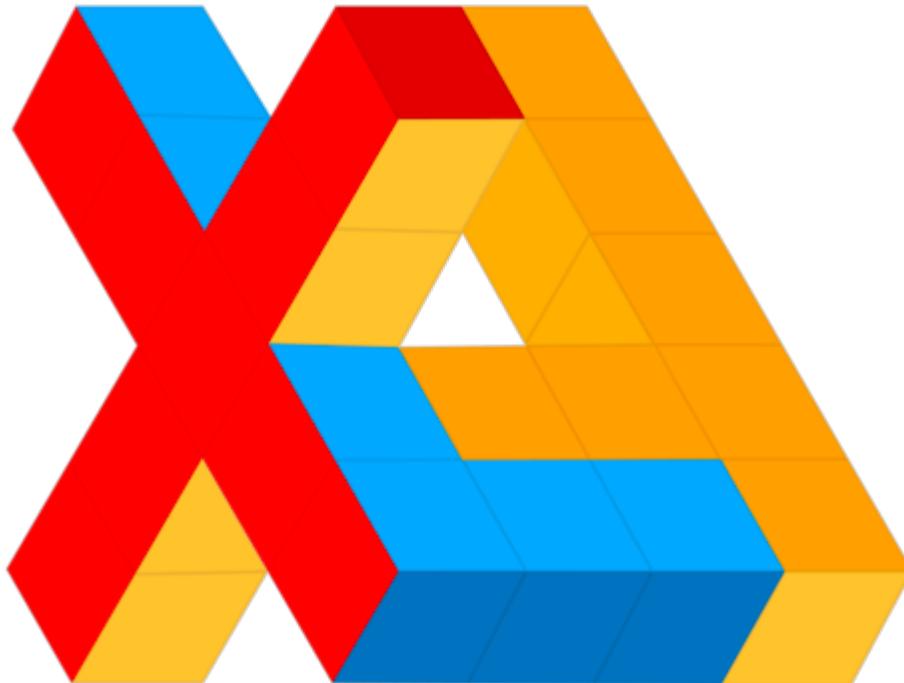
LLVM之父Chris Lattner：编译器的黄金时代



目标是重建全球ML基础设施，包括编译器、运行时，异构计算、边缘到数据中心并重，并专注于可用性，提升开发人员的效率。

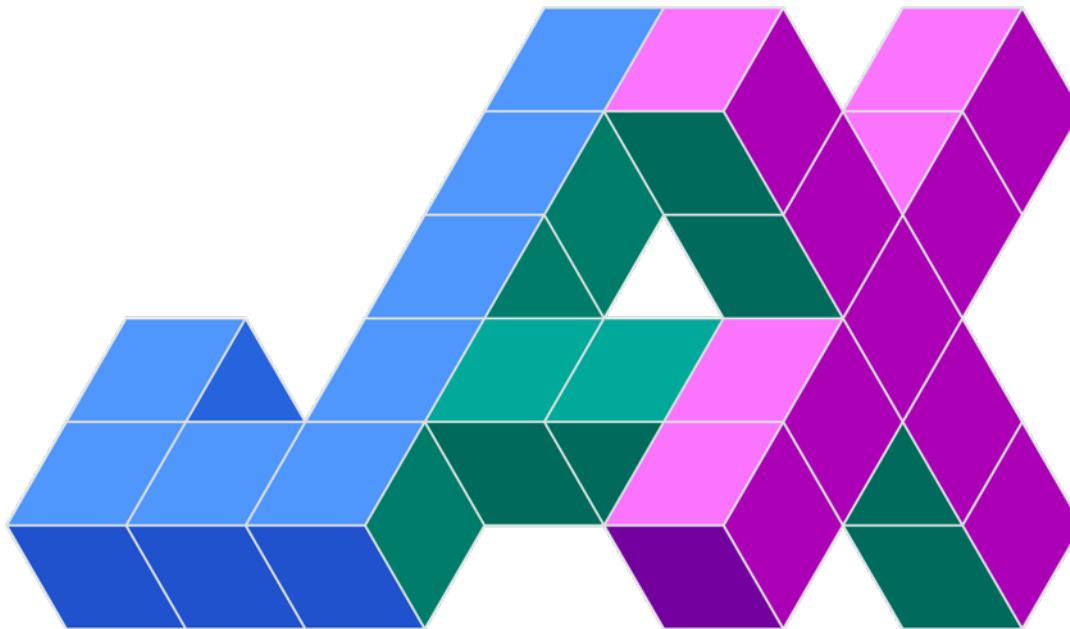
XLA：优化机器学习编译器

- XLA（加速线性代数）是一种针对特定领域的线性代数编译器，能够加快 TensorFlow 模型的运行速度，而且可能完全不需要更改源代码。



JAX : 高性能的数值计算库

- JAX 是Autograd和XLA的结合,JAX 本身不是一个深度学习的框架,他是一个高性能的数值计算库,更是结合了可组合的函数转换库,用于高性能机器学习研究。



TensorFlow：机器学习平台

- TensorFlow是一个端到端开源机器学习平台。它拥有一个全面而灵活的生态系统，其中包含各种工具、库和社区资源，可助力研究人员推动先进机器学习技术。



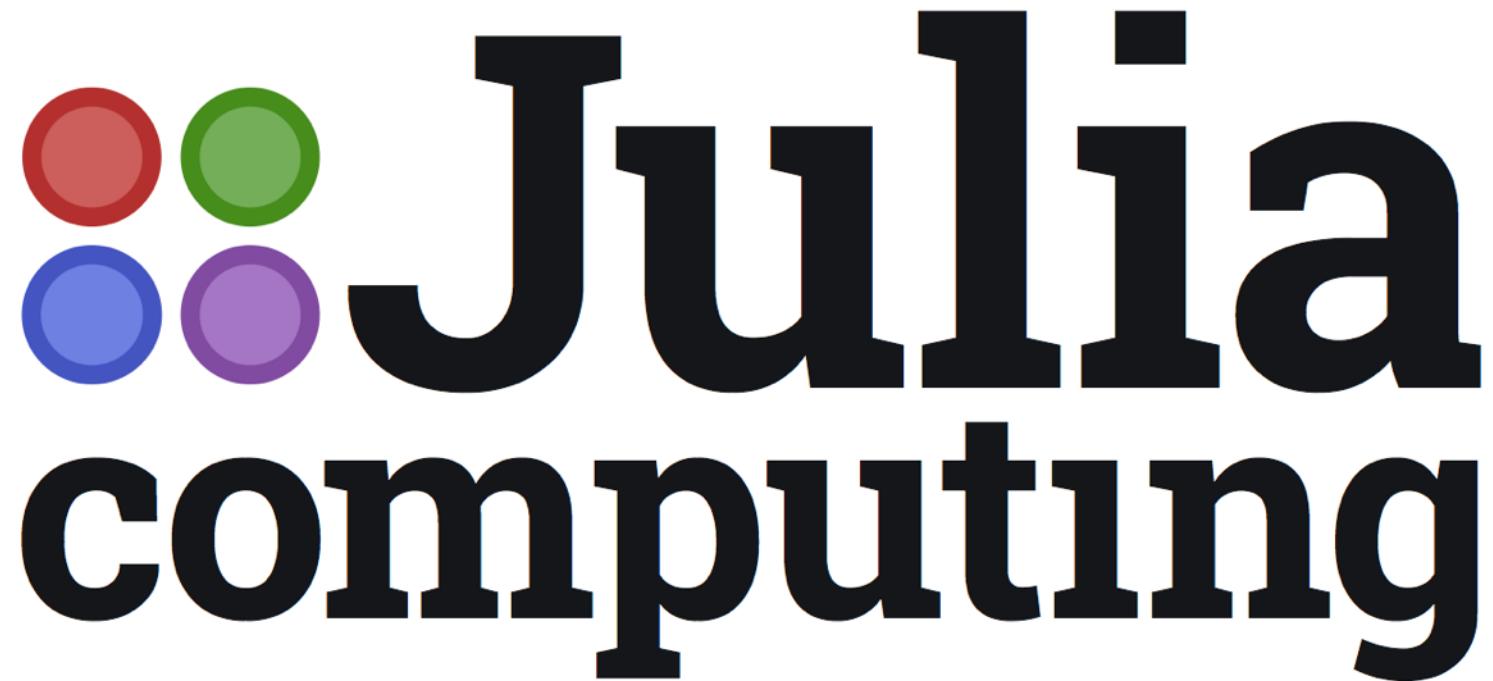
TVM 端到端深度学习编译器

- 为了使得各种硬件后端的计算图层级和算子层级优化成为可能，TVM 从现有框架中取得 DL 程序的高层级表示，并产生多硬件平台后端上低层级的优化代码，其目标是展示与人工调优的竞争力。



Julia：面向科学计算的高性能动态编程语言

- 计算中，Julia使用 LLVM JIT 编译。LLVM JIT 编译器通常不断地分析正在执行的代码，并且识别代码的一部分，使得从编译中获得的性能加速超过编译该代码的性能开销。





BUILDING A BETTER CONNECTED WORLD

THANK YOU

Copyright©2014 Huawei Technologies Co., Ltd. All Rights Reserved.

The information in this document may contain predictive statements including, without limitation, statements regarding the future financial and operating results, future product portfolio, new technology, etc. There are a number of factors that could cause actual results and developments to differ materially from those expressed or implied in the predictive statements. Therefore, such information is provided for reference purpose only and constitutes neither an offer nor an acceptance. Huawei may change the information at any time without notice.