LLVM架构简析

—— 南盘江计划

中科院软件所PLCT实验室 史宁宁

shiningning@iscas.ac.cn

2023年4月22日

目录

- 自我介绍与资料推荐
- LLVM整体框架
- LLVM的前端Clang
- LLVM的IR
- LLVM的pass
- LLVM的后端

目录

史宁宁 中科院软件所PLCT实验室

- 知乎ID：小乖他爹　　CSDN ID：snsn1984

- LLVM/Clang相关技术文章（写作时间较早）

  LLVM每日谈 https://www.zhihu.com/column/llvm-clang

  深入研究Clang https://www.zhihu.com/column/clang

  LLVM简介 https://llvm-cn.blog.csdn.net/article/details/47039035

  LLVM零基础学习 https://blog.csdn.net/snsn1984/category_9261412.html

史宁宁 中科院软件所PLCT实验室（续）

- 公开报告的PPT和视频 https://github.com/shining1984/talks
  《浅谈LLVM的后端》
  《浅谈LLVM的异常处理》
  《LLVM基础及Pass介绍》
  《聊聊我最近读的编译器后端论文》
- 技术书籍
  《华为方舟编译器之美》 2020年
  《Android Runtime源码解析》 2022年
  《编译器后端理论与实践》 2024年

LLVM书籍介绍

- 目前LLVM的书共有八本，其中有三本翻译为中文，具体如下：

1. Getting Started with LLVM Core Libraries --Bruno Cardoso Lopes, Rafael Auler

   中文版：《LLVM编译器实战教程》

2. LLVM Essentials: Become familiar with the LLVM infrastructure and start using LLVM libraries to design a compiler -- Suyog Sarda, Mayur Pandey

3. LLVM Cookbook --Mayur Pandey, Suyog Sarda

   中文版：LLVM Cookbook中文版

4. LLVM Techniques, Tips, and Best Practices—— Clang and Middle-End Libraries

5. Learn LLVM 12

   中文版（非官方）GitHub - xiaoweiChen/Learn-LLVM-12: 《Learn LLVM 12》的非专业个人翻译

LLVM书籍介绍（续）

6. Tutorial: Creating an LLVM Backend for the Cpu0 Architecture (Github published) --Chen Chung-Shu

7. Tutorial: Creating an LLVM Toolchain for the Cpu0 Architecture (Github published) --Chen Chung-Shu

8. Learn LLVM Core Libraries, 2nd Edition --Dmitrii Borisenkov / Bruno Lopes / Rafael Auler

LLVM官方资源

- https://llvm.org/devmtg/ 历年LLVM会议资料，多数配有视频。
- https://llvm.org/docs/ LLVM文档
- https://llvm.org/doxygen/ LLVM doxygen文档
- https://clang.llvm.org/doxygen/ Clang doxygen文档

目录

LLVM

- The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. Despite its name, LLVM has little to do with traditional virtual machines. The name "LLVM" itself is not an acronym; it is the full name of the project.

From: https://llvm.org/

The primary sub-projects of LLVM

The primary sub-projects of LLVM			
LLVM Core	Clang	LLDB	libc++ libc++ ABI
compiler-rt	MLIR	OpenMP	polly
libclc	klee	LLD	BOLT

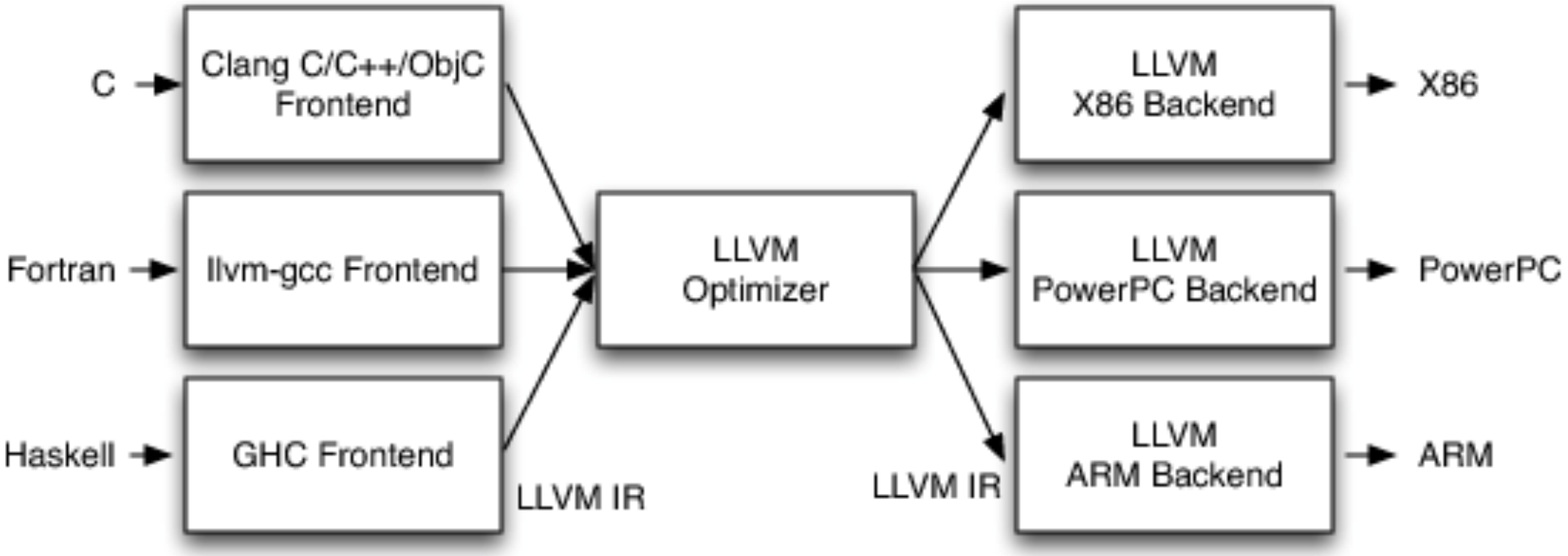
From: https://llvm.org/

LLVM架构

Three Major Components of a Three-Phase Compiler

From: The Architecture of Open Source Applications (Vol 1) LLVM
https://aosabook.org/en/v1/llvm.html

Three-Phase Design



From: The Architecture of Open Source Applications (Vol 1) LLVM
https://aosabook.org/en/v1/llvm.html

Projects built with LLVM

- dragonegg
- vmkit
- DawnCC
- Terra Lang
- Codasip Studio
- Pony Programming Language
- SMACK Software Verifier
- DiscoPoP: A Parallelism Discovery Tool
- Just-in-time Adaptive Decoder Engine (Jade)
- The Crack Programming Language
- Rubinius: a Ruby implementation
- MacRuby
- pocl: Portable Computing Language
- TTA-based Codesign Environment (TCE)
- The IcedTea Version of Sun's OpenJDK
- The Pure Programming Language Compiler
- LDC - the LLVM-based D Compiler
- How to Write Your Own Compiler
- Register Allocation by Puzzle Solving
- Faust Real-Time Signal Processing System
- Adobe "Hydra" Language
- Calysto Static Checker
- Improvements on SSA-Based Register Allocation
- LENS Project
- Trident Compiler
- Ascenium Reconfigurable Processor Compiler
- Scheme to LLVM Translator
- LLVM Visualization Tool
- Improvements to Linear Scan register allocation
- LLVA-emu project
- SPEDI: Static Patch Extraction and Dynamic Insertion
- An LLVM Implementation of SSAPRE
- Jello: a retargetable **J**ust-In-Time compil**e**r for **LL**VM bytec**o**de
- Emscripten: An LLVM to JavaScript Compiler
- Rust: a safe, concurrent, practical language
- ESL: Embedded Systems Language
- RTSC: The Real-Time Systems Compiler
- Vuo: A modern visual programming language for multimedia artists

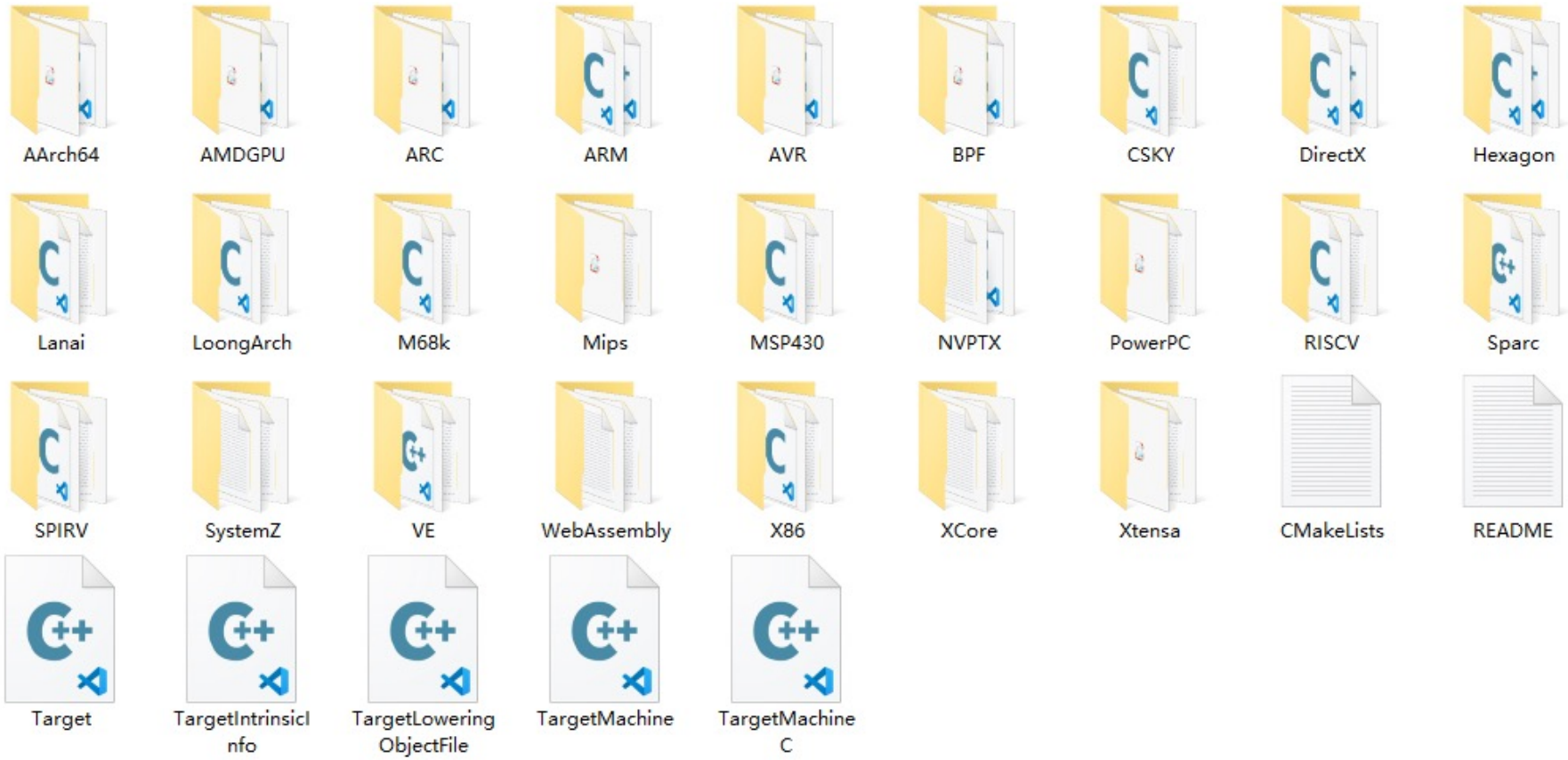
From: https://llvm.org/ProjectsWithLLVM/

LLVM所支持的语言

- Clang: C/C++/Objective-C
- projects that use components of LLVM: Ruby, Python, Haskell, Java, D, PHP, Pure, Lua等

From: https://llvm.org/

LLVM所支持的后端



Notes：This is the source code dir of LLVM 16.0.0，its location is LLVM/lib/Target/.

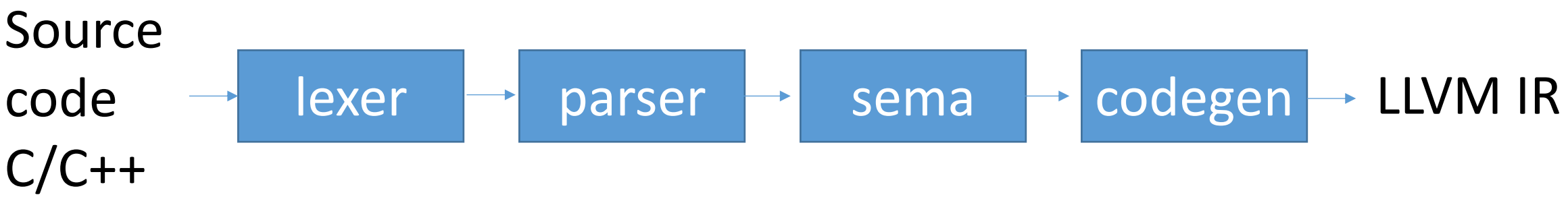
目录

Clang: a C language family frontend for LLVM

- The Clang project provides a language front-end and tooling infrastructure for languages in the C language family (C, C++, Objective C/C++, OpenCL, CUDA, and RenderScript) for the LLVM project. Both a GCC-compatible compiler driver (clang) and an MSVC-compatible compiler driver (clang-cl.exe) are provided

From: https://clang.llvm.org/

Clang的主要环节



Clang的词法分析和语法分析

- 在现阶段编译器开发实践中，前端部分的词法分析和语法分析的实现，已经越来越依靠Flex和Bison这类工具，很少依靠纯手工去实现。
- Clang依然采用C++来手动实现其lexer和parser。

Clang libs

- Lexer Library

The Lexer library contains several tightly-connected classes that are involved with the nasty process of lexing and preprocessing C source code.

- Parser Library

This library contains a recursive-descent parser that polls tokens from the preprocessor and notifies a client of the parsing progress.

- The AST Library

Clang AST nodes (types, declarations, statements, expressions, and so on) are generally designed to be immutable once created.

- Sema Library

This library is called by the Parser library during parsing to do semantic analysis of the input.

- CodeGen Library

CodeGen takes an AST as input and produces LLVM IR code from it.

From:
https://clang.llvm.org/docs/InternalsManual.html

Clang tools

- LibClang

LibClang is a stable high level C interface to clang. When in doubt LibClang is probably the interface you want to use.

- Clang Plugins

Clang Plugins allow you to run additional actions on the AST as part of a compilation. Plugins are dynamic libraries that are loaded at runtime by the compiler, and they’re easy to integrate into your build environment.

- LibTooling

LibTooling is a C++ interface aimed at writing standalone tools, as well as integrating into services that run clang tools.

From: https://clang.llvm.org/docs/Tooling.html

LibClang project

- Screader (2014)

The screader is a soure code reading tool based the libclang. It is implemented by the C. The screader can work on the clang-10.0.0 now.

https://github.com/shining1984/screader

目录

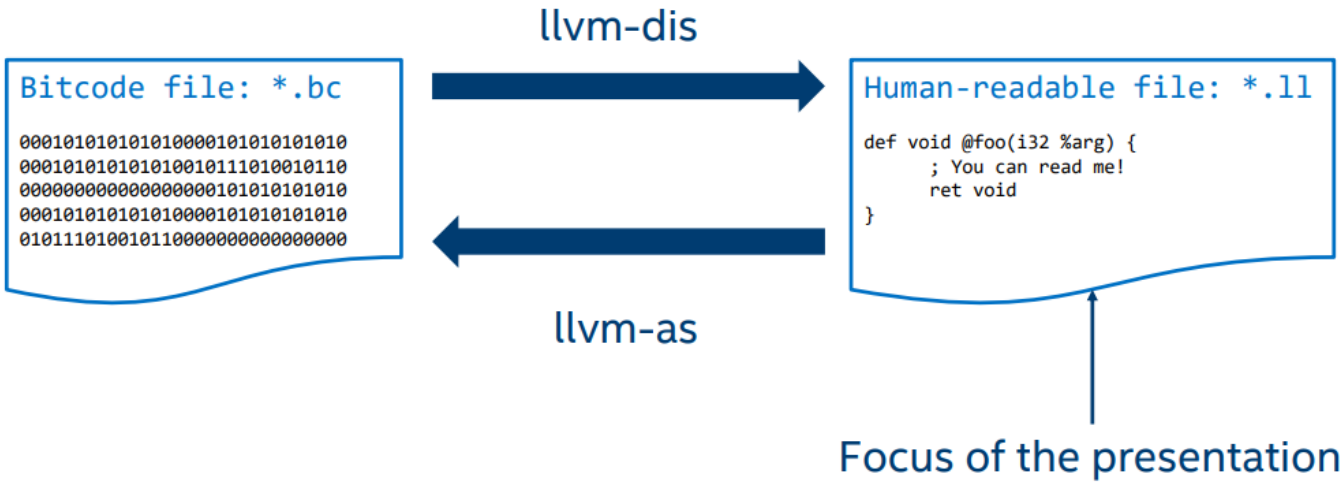
What is the LLVM IR?

- The LLVM Intermediate Representation
- 线性IR
- 单层IR
- 基于SSA

LLVM IR DOC： https://llvm.org/docs/LangRef.html

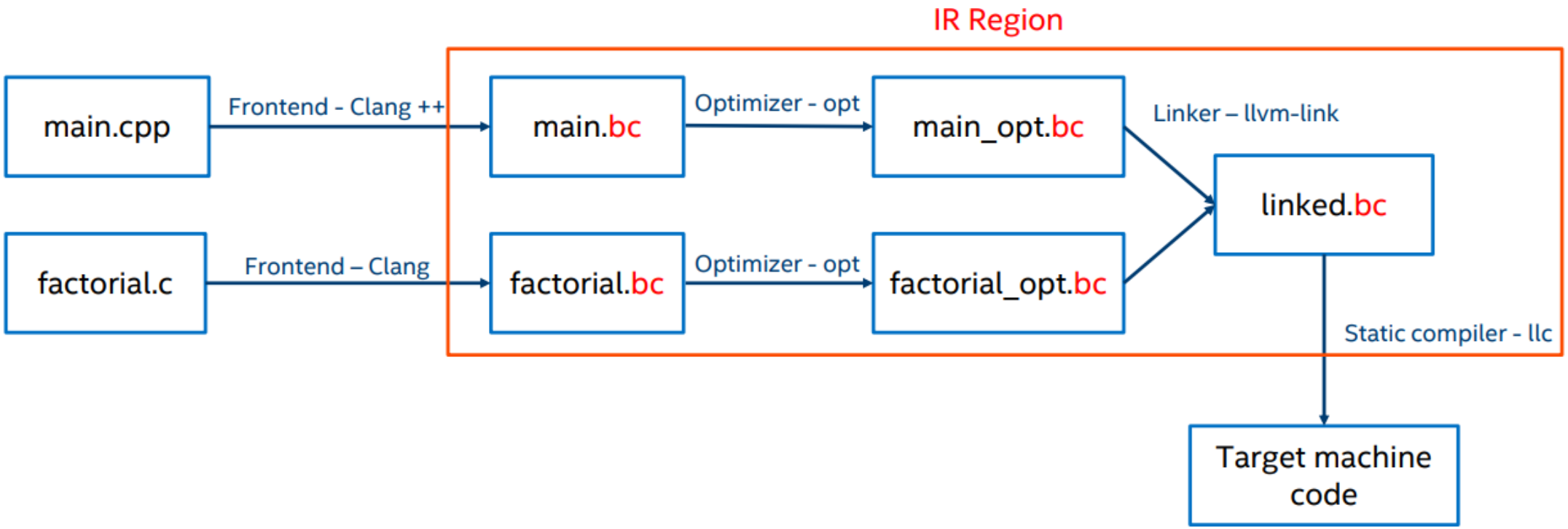
IR 的不同形式

- The LLVM code representation is designed to be used in three different forms: as an in-memory compiler IR, as an on-disk bitcode representation (suitable for fast loading by a Just-In-Time compiler), and as a human readable assembly language representation.



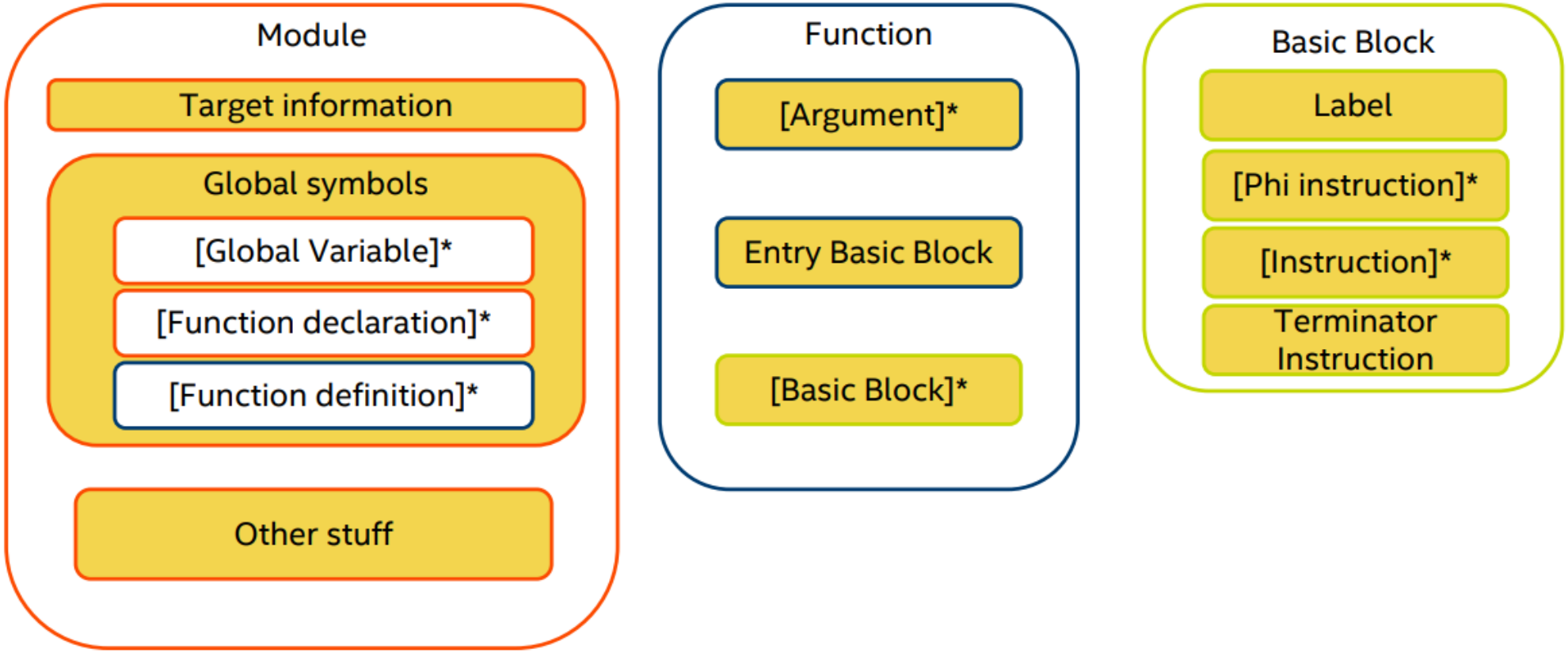
From:
LLVM IR DOC：
https://llvm.org/docs/LangRef.html
LLVM IR TUTORIAL
https://llvm.org/devmtg/2019-04/slides/Tutorial-Bridgers-LLVM_IR_tutorial.pdf

IR & the compilation process



From: LLVM IR TUTORIAL https://llvm.org/devmtg/2019-04/slides/Tutorial-Bridgers-LLVM_IR_tutorial.pdf

Simplified IR layout



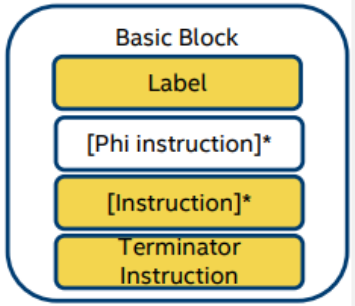
From: LLVM IR TUTORIAL https://llvm.org/devmtg/2019-04/slides/Tutorial-Bridgers-LLVM_IR_tutorial.pdf

Basic Blocks

Basic Blocks

List of non-terminator instructions ending with a terminator instruction:

- **Branch - "br"**
- **Return - "ret"**
- Switch – "switch"
- Unreachable – "unreachable"
- Exception handling instructions

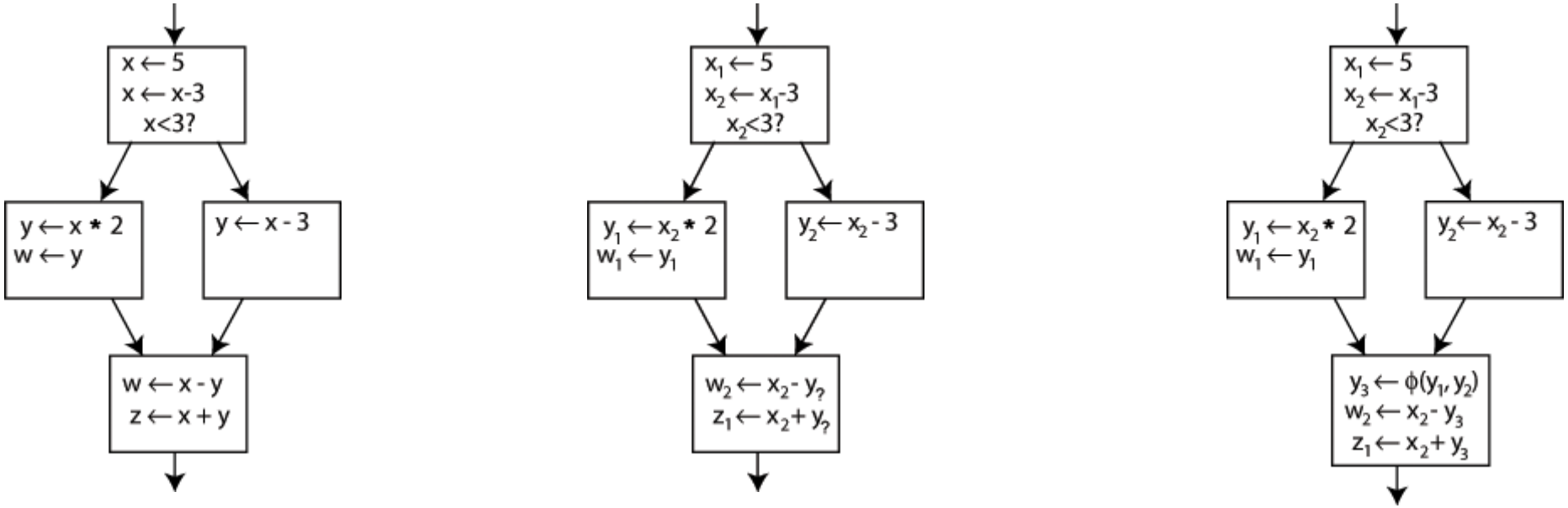



```
; Precondition: %val is non-negative.
define i32 @factorial(i32 %val) {
  %is_base_case = icmp eq i32 %val, 0
  br i1 %is_base_case, label %base_case, label %recursive_case
base_case:
  ret i32 1
recursive_case:
  %1 = add i32 -1, %val
  %2 = call i32 @factorial(i32 %0)
  %3 = mul i32 %val, %1
  ret i32 %2
}
```

From: LLVM IR TUTORIAL https://llvm.org/devmtg/2019-04/slides/Tutorial-Bridgers-LLVM_IR_tutorial.pdf

Static Single Assignment(SSA) & Φ (Phi) function

- 静态单赋值
- 每个变量都只被赋值一次

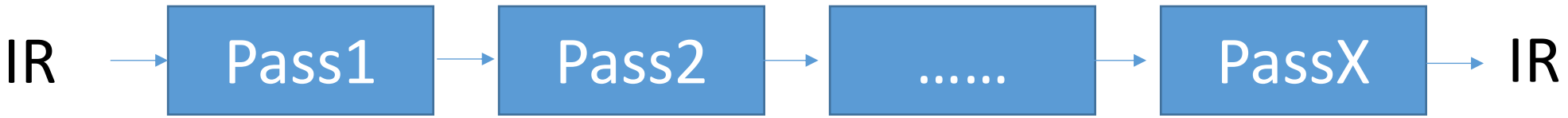


From: https://en.wikipedia.org/wiki/Static_single-assignment_form

目录

Pass

- Optimizations are implemented as Passes that traverse some portion of a program to either collect information or transform the program.



From: https://llvm.org/docs/Passes.html

Pass分类

- Analysis passes
- Transform passes
- Utility passes

From: https://llvm.org/docs/Passes.html

Analysis Passes

- Analysis passes compute information that other passes can use or for debugging or program visualization purposes.

-aa-eval: Exhaustive Alias Analysis Precision Evaluator
-basic-aa: Basic Alias Analysis (stateless AA impl)
-basiccg: Basic CallGraph Construction
-count-aa: Count Alias Analysis Query Responses
-da: Dependence Analysis
-debug-aa: AA use debugger
-domfrontier: Dominance Frontier Construction
-domtree: Dominator Tree Construction
-dot-callgraph: Print Call Graph to “dot” file

From: https://llvm.org/docs/Passes.html

Analysis Passes（续）

-dot-cfg: Print CFG of function to “dot” file
-dot-cfg-only: Print CFG of function to “dot” file (with no function bodies)
-dot-dom: Print dominance tree of function to “dot” file
-dot-dom-only: Print dominance tree of function to “dot” file (with no function bodies)
-dot-post-dom: Print postdominance tree of function to “dot” file
-dot-post-dom-only: Print postdominance tree of function to “dot” file (with no function bodies)
-globalsmodref-aa: Simple mod/ref analysis for globals
-instcount: Counts the various types of Instructions
-intervals: Interval Partition Construction
-iv-users: Induction Variable Users
-lazy-value-info: Lazy Value Information Analysis

From: https://llvm.org/docs/Passes.html

Analysis Passes（续）

-libcall-aa: LibCall Alias Analysis
-lint: Statically lint-checks LLVM IR
-loops: Natural Loop Information
-memdep: Memory Dependence Analysis
-module-debuginfo: Decodes module-level debug info
-postdomfrontier: Post-Dominance Frontier Construction
-postdomtree: Post-Dominator Tree Construction
-print-alias-sets: Alias Set Printer
-print-callgraph: Print a call graph
-print-callgraph-sccs: Print SCCs of the Call Graph

From: https://llvm.org/docs/Passes.html

Analysis Passes（续）

-print-function: Print function to stderr

-print-module: Print module to stderr

-print-used-types: Find Used Types

-regions: Detect single entry single exit regions

-scalar-evolution: Scalar Evolution Analysis

-scev-aa: ScalarEvolution-based Alias Analysis

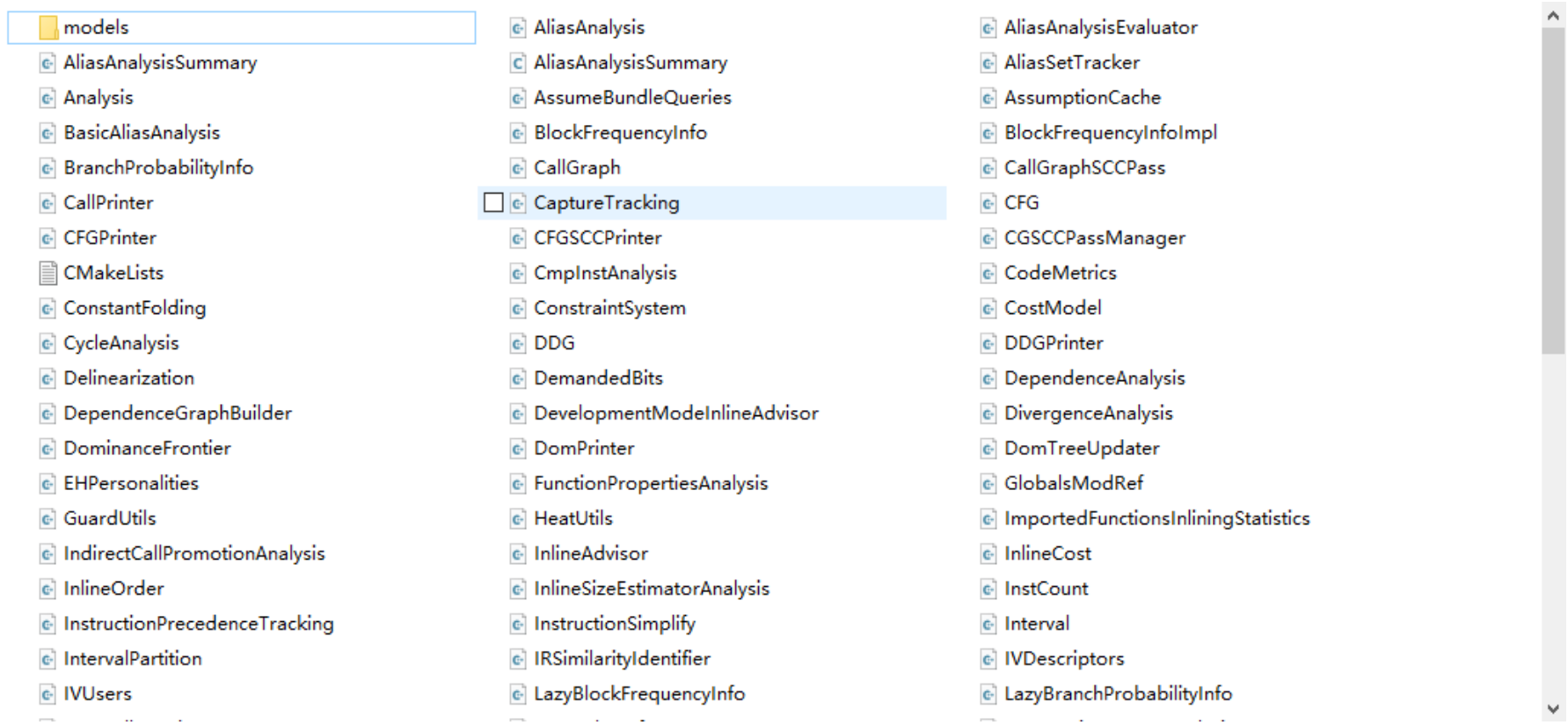
-stack-safety: Stack Safety Analysis

-targetdata: Target Data Layout

From: https://llvm.org/docs/Passes.html

Analysis Passes代码

- llvm/lib/Analysis/



Notes：This is the source code of LLVM 16.0.0.

Transform passes

- Transform passes can use (or invalidate) the analysis passes. Transform passes all mutate the program in some way.

-adce: Aggressive Dead Code Elimination
-always-inline: Inliner for always_inline functions
-argpromotion: Promote ‘by reference’ arguments to scalars
-bb-vectorize: Basic-Block Vectorization
-block-placement: Profile Guided Basic Block Placement
-break-crit-edges: Break critical edges in CFG
-codegenprepare: Optimize for code generation
-constmerge: Merge Duplicate Global Constants

From: https://llvm.org/docs/Passes.html

Transform passes（续)

-dce: Dead Code Elimination
-deadargelim: Dead Argument Elimination
-deadtypeelim: Dead Type Elimination
-die: Dead Instruction Elimination
-dse: Dead Store Elimination
-function-attrs: Deduce function attributes
-globaldce: Dead Global Elimination
-globalopt: Global Variable Optimizer
-gvn: Global Value Numbering
-indvars: Canonicalize Induction Variables

From: https://llvm.org/docs/Passes.html

Transform passes（续)

-inline: Function Integration/Inlining
-instcombine: Combine redundant instructions
-aggressive-instcombine: Combine expression patterns
-internalize: Internalize Global Symbols
-ipsccp: Interprocedural Sparse Conditional Constant Propagation
-jump-threading: Jump Threading
-lcssa: Loop-Closed SSA Form Pass
-licm: Loop Invariant Code Motion
-loop-deletion: Delete dead loops
-loop-extract: Extract loops into new functions

From: https://llvm.org/docs/Passes.html

Transform passes（续)

-loop-extract-single: Extract at most one loop into a new function
-loop-reduce: Loop Strength Reduction
-loop-rotate: Rotate Loops
-loop-simplify: Canonicalize natural loops
-loop-unroll: Unroll loops
-loop-unroll-and-jam: Unroll and Jam loops
-loop-unswitch: Unswitch loops
-lower-global-dtors: Lower global destructors
-loweratomic: Lower atomic intrinsics to non-atomic form
-lowerinvoke: Lower invokes to calls, for unwindless code generators

From: https://llvm.org/docs/Passes.html

Transform passes（续)

-lowerswitch: Lower SwitchInsts to branches
-mem2reg: Promote Memory to Register
-memcpyopt: MemCpy Optimization
-mergefunc: Merge Functions
-mergereturn: Unify function exit nodes
-partial-inliner: Partial Inliner
-prune-eh: Remove unused exception handling info
-reassociate: Reassociate expressions
-rel-lookup-table-converter: Relative lookup table converter
-reg2mem: Demote all values to stack slots

From: https://llvm.org/docs/Passes.html

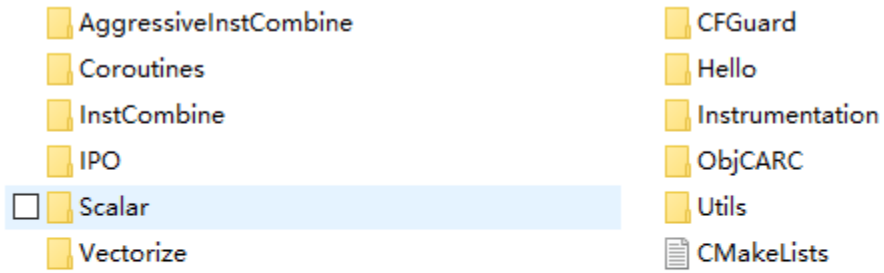
Transform passes（续）

-sroa: Scalar Replacement of Aggregates
-sccp: Sparse Conditional Constant Propagation
-simplifycfg: Simplify the CFG
-sink: Code sinking
-strip: Strip all symbols from a module
-strip-dead-debug-info: Strip debug info for unused symbols
-strip-dead-prototypes: Strip Unused Function Prototypes
-strip-debug-declare: Strip all llvm.dbg.declare intrinsics
-strip-nondebug: Strip all symbols, except dbg symbols, from a module
-tailcallelim: Tail Call Elimination

From: https://llvm.org/docs/Passes.html

Transform passes代码

- llvm/lib/Transforms/



Notes：This is the source code of LLVM 16.0.0.

Utility passes

- Utility passes provides some utility but don’t otherwise fit categorization.

-deadarghaX0r: Dead Argument Hacking (BUGPOINT USE ONLY; DO NOT USE)

-extract-blocks: Extract Basic Blocks From Module (for bugpoint use)

-instnamer: Assign names to anonymous instructions

-verify: Module Verifier

-view-cfg: View CFG of function

-view-cfg-only: View CFG of function (with no function bodies)

-view-dom: View dominance tree of function

-view-dom-only: View dominance tree of function (with no function bodies)

-view-postdom: View postdominance tree of function

-view-postdom-only: View postdominance tree of function (with no function bodies)

-transform-warning: Report missed forced transformations

From: https://llvm.org/docs/Passes.html

PassManager

- Manages a sequence of passes over a particular unit of IR.
- A pass manager contains a sequence of passes to run over a particular unit of IR (e.g. Functions, Modules). It is itself a valid pass over that unit of IR, and when run over some given IR will run each of its contained passes in sequence. Pass managers are the primary and most basic building block of a pass pipeline.

From: https://llvm.org/doxygen/classllvm_1_1PassManager.html

Pass总结

- LLVM 的Pass框架是LLVM系统的一个很重要的部分。每个Pass都是做分析或者转变/优化的工作，LLVM的转化/优化工作就是由很多个Pass来一起完成的。

- Pass是LLVM系统转化/优化的工作的一个节点，每个节点做一些工作，这些工作加起来就构成了LLVM中端和后端的转化/优化的工作。

- Pass架构的可重用性非常好。你可以选择已有的一些Pass，自己去构建出自己想要的分析或者转变/优化。

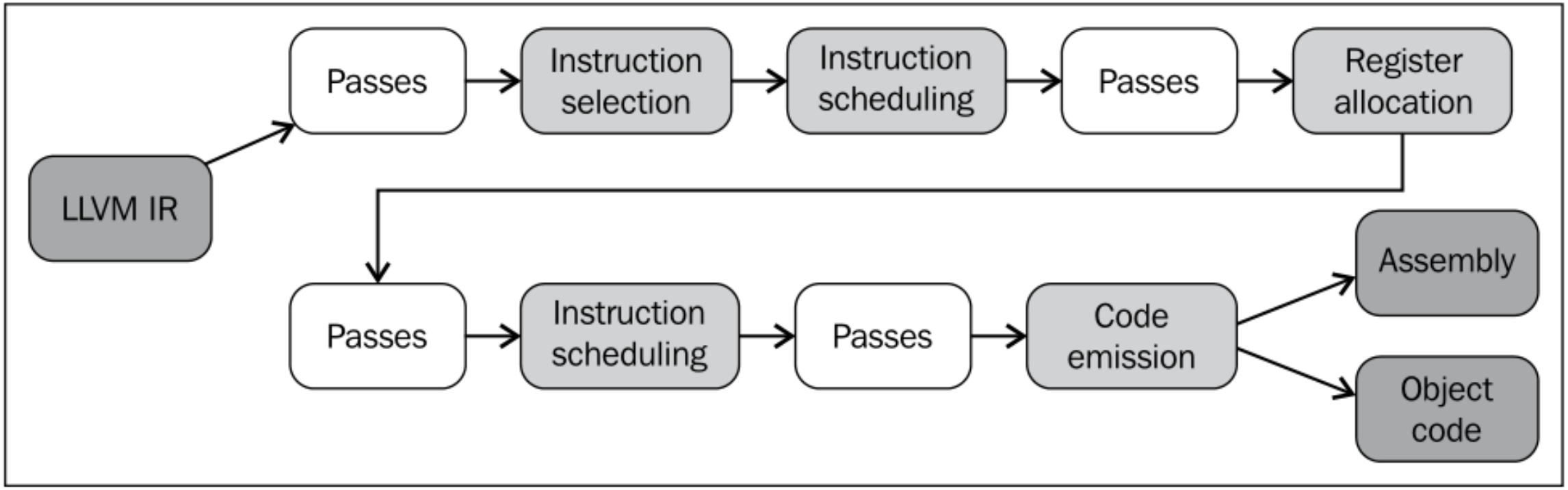
- 新建Pass并不用考虑LLVM之前的优化和转化是怎么做的，可以只运行自己新建的Pass，这样可以方便的实现自己想要的效果。

From: https://zhuanlan.zhihu.com/p/26127188

目录

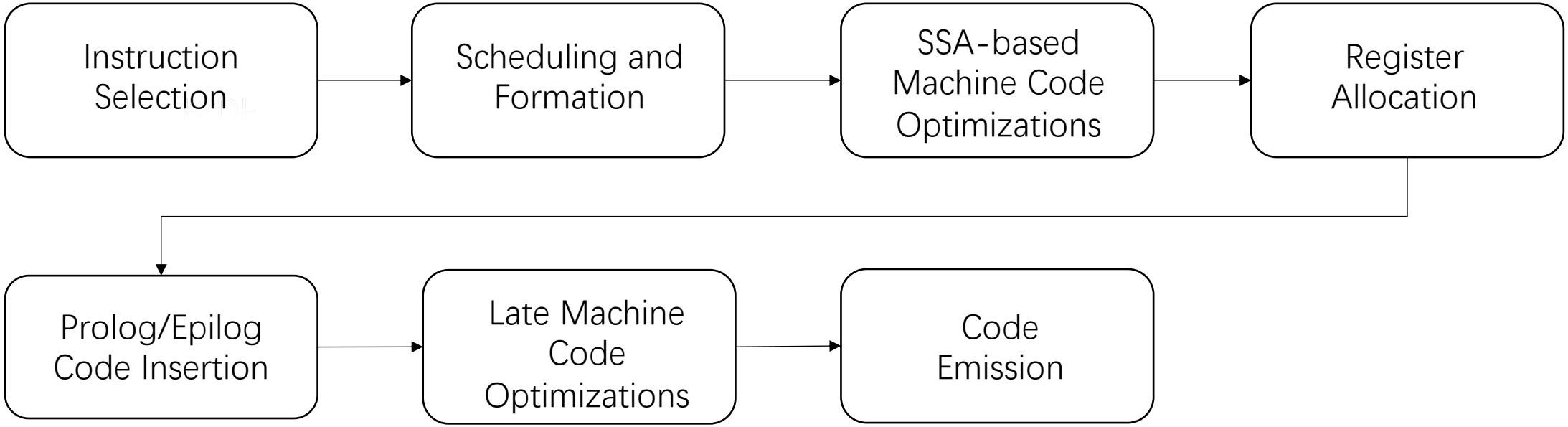
- 自我介绍与资料推荐
- LLVM整体框架
- LLVM的前端Clang
- LLVM的IR
- LLVM的pass
- LLVM的后端

The Steps in LLVM Backend



From: 《Getting Started with LLVM Core Libraries》P134.

The Steps in LLVM Backend （Another version）



Notes：According the LLVM DOC《The LLVM Target-Independent Code Generator》https://llvm.org/docs/CodeGenerator.html.

TableGen

- TableGen主要是帮助开发者开发和维护特定领域的信息，方便开发者更好的构建这些信息，避免错误。尤其是在面对大量的信息时，用起来比较方便。
- TableGen的主要使用者是The LLVM Target-Independent Code Generator和Clang diagnostics and attributes. 前者就是我们通常所讲的LLVM后端。
- TableGen有三个backend：LLVM、Clang和通用后端等。TableGen的LLVM后端主要是为了自动为LLVM后端生成指令、调度和架构特征等，TableGen的Clang后端主要为了诊断信息和特征，主要是文本操作。

From: https://zhuanlan.zhihu.com/p/80107428
https://llvm.org/docs/TableGen/
https://llvm.org/docs/TableGen/BackEnds.html

指令选择

- The instruction selector must primarily select the machine instructions which implement the same expected behavior as that of the input program, and as a secondary objective it should result in efficient assembly code. We reformulate this as the following subproblems:

  1. pattern matching – detecting when and where it is possible to use a certain machine instruction; and

  2. pattern selection – deciding which instruction to choose in situations when multiple options exist.

From: <Survey on Instruction Selection> P3
https://arxiv.org/abs/1306.4898

LLVM的指令选择器

- FastISel

"Fast" instruction selection is designed to emit very poor code quickly. "Fast" instruction selection is able to fail gracefully and transfer control to the SelectionDAG selector for operations that it doesn't support.

https://llvm.org/doxygen/FastISel_8cpp_source.html

- SelectionDAG

The SelectionDAG provides an abstraction for code representation in a way that is amenable to instruction selection using automatic techniques (e.g. dynamic-programming based optimal pattern matching selectors). Portions of the DAG instruction selector are generated from the target description (*.td) files. Our goal is for the entire instruction selector to be generated from these .td files, though currently there are still things that require custom C++ code.

https://www.llvm.org/docs/CodeGenerator.html#register-allocator

- GlobalISel

The initial goal is to replace FastISel on AArch64. The next step will be to replace SelectionDAG as the optimized ISel.

https://llvm.org/docs/GlobalISel/index.html

指令调度

- For the selected machine instructions we also need to decide in which order they shall appear in the assembly code, which is taken care of by the instruction scheduler.

From: <Survey on Instruction Selection> P3
https://arxiv.org/abs/1306.4898

LLVM的指令调度选项

- -pre-RA-sched

Instruction schedulers available (before register allocation)

llvm/lib/CodeGen/SelectionDAG/SelectionDAGISel.cpp

- -enable-misched

Enable the machine instruction scheduling pass.

llvm/lib/CodeGen/ MachineScheduler.cpp

Notes：The code is LLVM 16.0.0.

LLVM的指令调度选项

- -enable-post-misched

Enable the post-ra machine instruction scheduling pass.

llvm/lib/CodeGen/ MachineScheduler.cpp

- -post-RA-sched

Enable scheduling after register allocation

llvm/lib/CodeGen/PostRASchedulerList.cpp

Notes：The code is LLVM 16.0.0.

LLVM的指令调度实现

- SelectionDAGISel

class SelectionDAGISel : public MachineFunctionPass{…}

llvm/include/llvm/CodeGen/SelectionDAGISel.h

Notes：The code is LLVM 16.0.0.

LLVM的指令调度实现（续）

- MachineScheduler && PostMachineScheduler

/// Base class for a machine scheduler class that can run at any point.
class MachineSchedulerBase : public MachineSchedContext,
public MachineFunctionPass {…

/// MachineScheduler runs after coalescing and before register allocation.
class MachineScheduler : public MachineSchedulerBase {…

/// PostMachineScheduler runs after shortly before code emission.
class PostMachineScheduler : public MachineSchedulerBase {…

llvm/lib/CodeGen/MachineScheduler.cpp

Notes：The code is LLVM 16.0.0.

LLVM的指令调度实现（续）

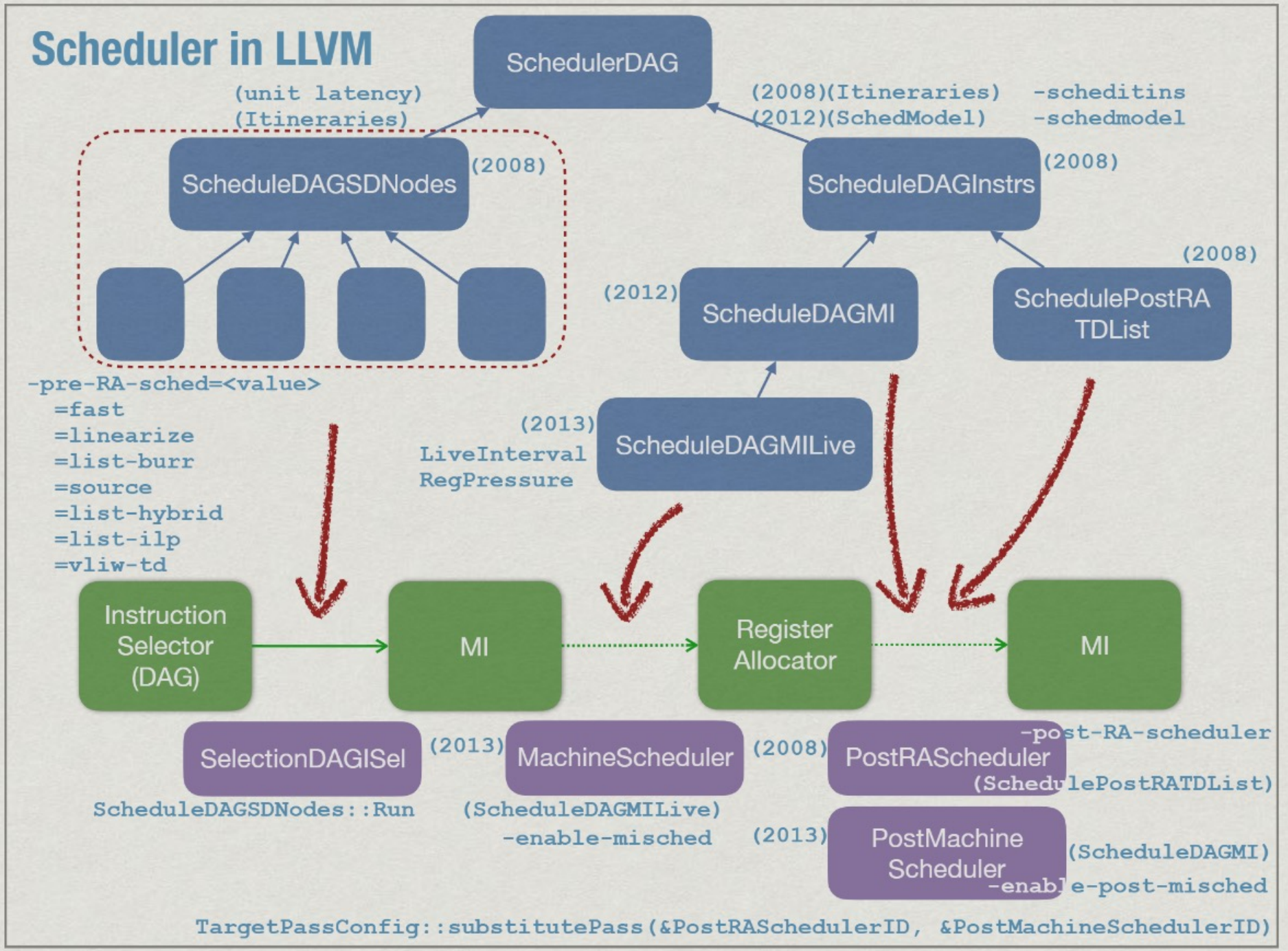
- PostRAScheduler

class PostRAScheduler : public MachineFunctionPass {…

llvm/lib/CodeGen/PostRASchedulerList.cpp

Notes：The code is LLVM 16.0.0.

LLVM的指令调度



From:https://github.com/CatSystemWorkshop/COSCUP2019/blob/master/Instruction%20Scheduler%20in%20LLVM.pdf

寄存器分配前的指令调度调用 SelectionDAGISel

```
/// createBURRListDAGScheduler - This creates a bottom up register usage
/// reduction list scheduler.
ScheduleDAGSDNodes *createBURRListDAGScheduler(SelectionDAGISel *IS,
                                               CodeGenOpt::Level OptLevel);

/// createBURRListDAGScheduler - This creates a bottom up list scheduler that
/// schedules nodes in source code order when possible.
ScheduleDAGSDNodes *createSourceListDAGScheduler(SelectionDAGISel *IS,
                                                 CodeGenOpt::Level OptLevel);

/// createHybridListDAGScheduler - This creates a bottom up register pressure
/// aware list scheduler that make use of latency information to avoid stalls
/// for long latency instructions in low register pressure mode. In high
/// register pressure mode it schedules to reduce register pressure.
ScheduleDAGSDNodes *createHybridListDAGScheduler(SelectionDAGISel *IS,
                                                 CodeGenOpt::Level);
```

From: llvm16.0.0 llvm/include/llvm/CodeGen/SchedulerRegistry.h

寄存器分配前的指令调度调用SelectionDAGISel（续）

```
/// createILPListDAGScheduler - This creates a bottom up register pressure
/// aware list scheduler that tries to increase instruction level parallelism
/// in low register pressure mode. In high register pressure mode it schedules
/// to reduce register pressure.
ScheduleDAGSDNodes *createILPListDAGScheduler(SelectionDAGISel *IS,
                                              CodeGenOpt::Level);

/// createFastDAGScheduler - This creates a "fast" scheduler.
ScheduleDAGSDNodes *createFastDAGScheduler(SelectionDAGISel *IS,
                                           CodeGenOpt::Level OptLevel);

/// createVLIWDAGScheduler - Scheduler for VLIW targets. This creates top down
/// DFA driven list scheduler with clustering heuristic to control
/// register pressure.
ScheduleDAGSDNodes *createVLIWDAGScheduler(SelectionDAGISel *IS,
                                           CodeGenOpt::Level OptLevel);
```

From: llvm16.0.0 llvm/include/llvm/CodeGen/SchedulerRegistry.h

寄存器分配前的指令调度调用SelectionDAGISel（续）

```
/// createDefaultScheduler - This creates an instruction scheduler appropriate
/// for the target.
ScheduleDAGSDNodes *createDefaultScheduler(SelectionDAGISel *IS,
                                           CodeGenOpt::Level OptLevel);

/// createDAGLinearizer - This creates a "no-scheduling" scheduler which
/// linearize the DAG using topological order.
ScheduleDAGSDNodes *createDAGLinearizer(SelectionDAGISel *IS,
                                        CodeGenOpt::Level OptLevel);
}
```

From: llvm16.0.0 llvm/include/llvm/CodeGen/SchedulerRegistry.h

寄存器分配前的指令调度调用MachineSchedulerPass

From: llvm16.0.0
llvm/include/llvm/CodeGen/CodeGenPassBuilder.h

```
template <typename Derived>
void CodeGenPassBuilder<Derived>::addOptimizedRegAlloc(
    AddMachinePass &addPass) const {
…
  // The machine scheduler may accidentally create disconnected components
  // when moving subregister definitions around, avoid this by splitting them to
  // separate vregs before. Splitting can also improve reg. allocation quality.
  addPass(RenameIndependentSubregsPass());

  // PreRA instruction scheduling.
  addPass(MachineSchedulerPass());
…
}
```

寄存器分配后的指令调度调用

```
template <typename Derived>
Error CodeGenPassBuilder<Derived>::addMachinePasses(
    AddMachinePass &addPass) const {
…
  // Second pass scheduler.
if (getOptLevel() != CodeGenOpt::None &&
      !TM.targetSchedulesPostRAScheduling()) {
    if (Opt.MISchedPostRA)
      addPass(PostMachineSchedulerPass());
    else
      addPass(PostRASchedulerPass());
  }
…
}
```

From: llvm16.0.0
llvm/include/llvm/CodeGen/CodeGenPassBuilder.h

寄存器分配

- 寄存器分配实质上就是如何使用处理器有限数量的寄存器。编译器所针对的目标处理器所具有的寄存器都是有限的，如何将这些有限的寄存器给指令使用，或者替换指令中之前使用的虚拟寄存器/变量，就是寄存器分配索要考虑的问题。
- 寄存器分配不仅要满足指令使用寄存器的需求，还要尽可能的提高寄存器的使用效率，以提高指令的运行速度，避免因为寄存器使用带来大量数据转移的指令。

From:《编译器后端理论与实践》（史宁宁，未出版）

寄存器分配常见的算法

- 寄存器分配常见的算法有图着色、线性扫描、基于SSA的寄存器分配、PBQP、线性规划、弦图分配等。其中，基于SSA的寄存器分配属于分配算法的一类，其中包含多种可应用于SSA格式的寄存器分配算法。

From:《编译器后端理论与实践》（史宁宁，未出版）

LLVM的寄存器分配

- LLVM infrastructure provides four register allocator, namely: fast, basic, greedy and PBQP.
- The Fast Register Allocator

  This allocator is a local register allocator, which has the simplest strategy when compared with the others. It scans the program linearly and assigns values to registers as they appear.
- The Basic Register Allocator(linear scan)

  The basic register allocator is an extension of the linear scan register allocator proposed by Poletto with some extensions.
- The Greedy Register Allocator(linear scan)

  The Greedy register allocator is an implementation of the basic allocator that uses global live range.

From: <A Detailed Analysis of the LLVM's Register Allocators> P190-191
https://ieeexplore.ieee.org/document/6694089

LLVM的寄存器分配

- The PBQP Register Allocator

  The PBQP register allocator is the LLVM register allocator that performs allocation based on the Partitioned Boolean Quadratic Programming .PBQP is an algorithm that transforms the problem of register allocation into Partitioned Boolean Quadratic Problem.

- 分区布尔二次规划？分区布尔二次编程？

From: <A Detailed Analysis of the LLVM's Register Allocators> P190-191
https://ieeexplore.ieee.org/document/6694089

LLVM寄存器分配器的使用

The type of register allocator used in `llc` can be chosen with the command line option `-regalloc=...`:

```
$ llc -regalloc=linearscan file.bc -o ln.s
$ llc -regalloc=fast file.bc -o fa.s
$ llc -regalloc=pbqp file.bc -o pbqp.s
```

From: https://www.llvm.org/docs/CodeGenerator.html#register-allocator

参考资料

- 参见具体每页下方标注。

Thanks!