LLVM基础及Pass介绍

史宁宁

2019年6月8日

自我介绍

- 2012年6月开始做LLVM相关的项目
- CSDN

自我介绍

- 知乎

小乖他爹 编译器，LLVM/Clang，LLVM专栏：LLVM每日谈

居住地	现居吉林省
所在行业	高等教育
教育经历	吉林大学 · 计算机软件与理论
个人简介	形而上者谓之道，形而下者谓之器。

我的专栏

TensorFlow世界
和TensorFlow相关的知识。
发表 9 篇文章 · 共 9 篇文章 · 928 人关注

LLVM每日谈
每天聊点LLVM
发表 64 篇文章 · 共 65 篇文章 · 2,843 人关注

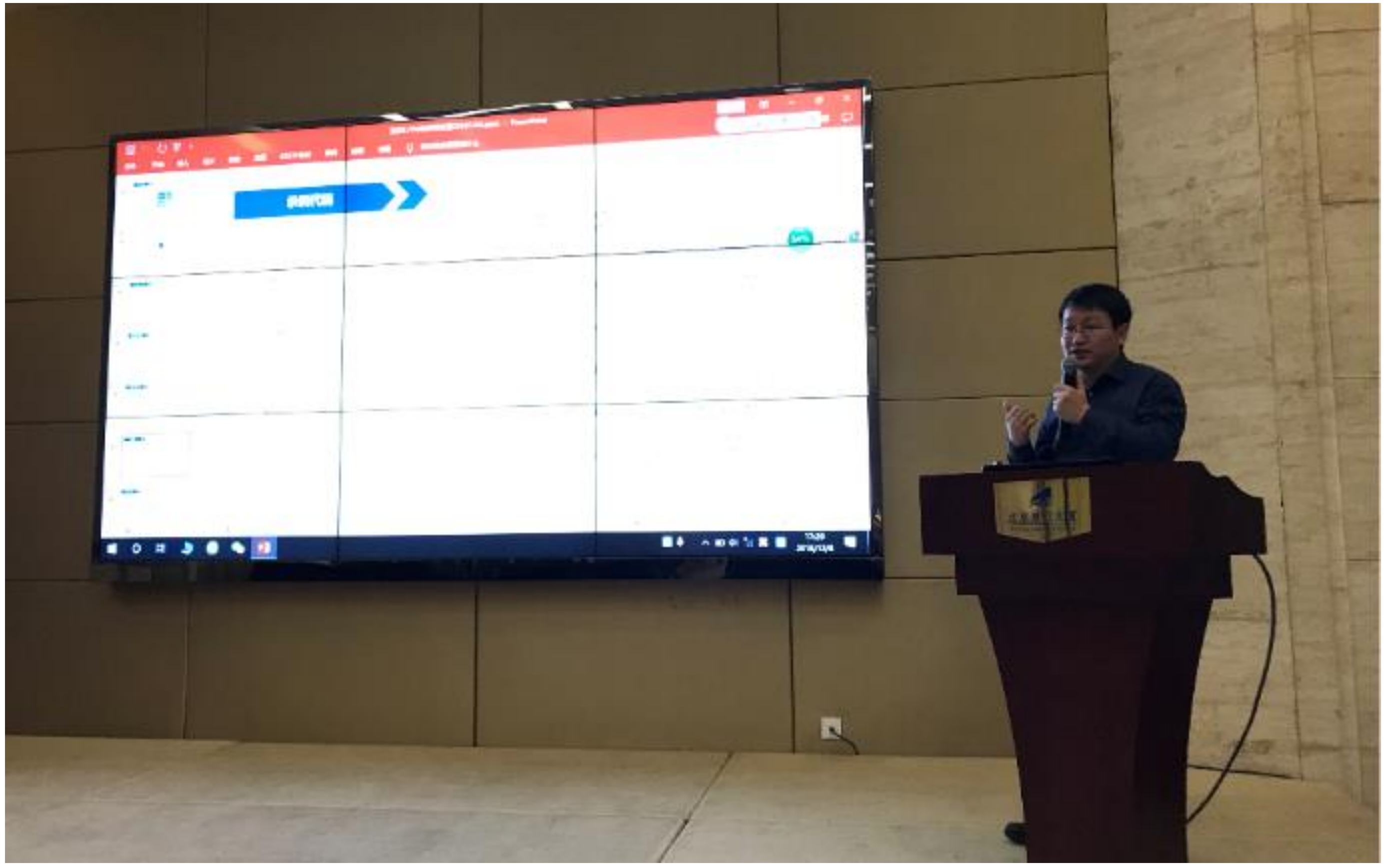
深入研究Clang
发表 12 篇文章 · 共 12 篇文章 · 1,297 人关注

技术动态
一些个人觉得需要关注的技术动态。
发表 4 篇文章 · 共 4 篇文章 · 9 人关注

《左传》杂谈
谈谈《左传》相关的材料和故事
发表 6 篇文章 · 共 6 篇文章 · 11 人关注

自我介绍

- HelloLLVM 2018年上海站、杭州站
- OSDT2018



目录

- LLVM简介
- LLVM IR
- LLVM Backend
- Pass的简介

LLVM简介

LLVM

- The LLVM Project is a collection of modular and reusable compiler and toolchain technologies.

The primary sub-projects of LLVM

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

Projects built with LLVM

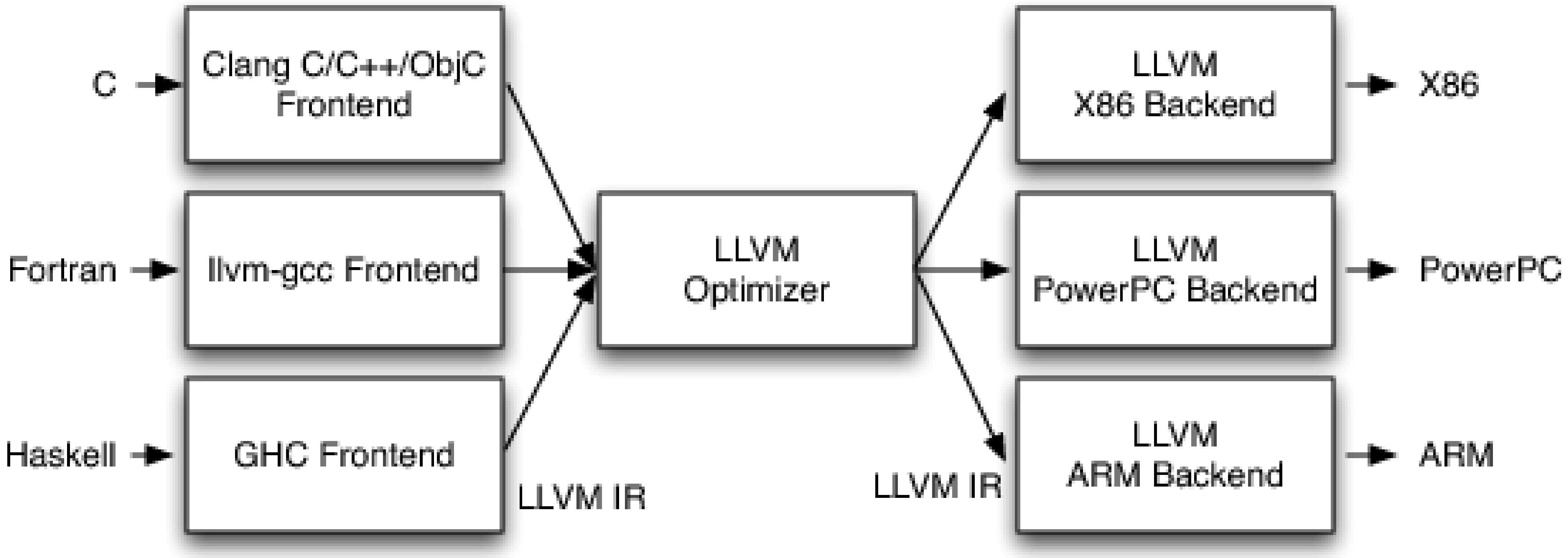
Projects built with LLVM

- 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 Just-In-Time compiler for LLVM bytecode
- 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

LLVM简介

Three Major Components of a Three-Phase Compiler

Three-Phase Design

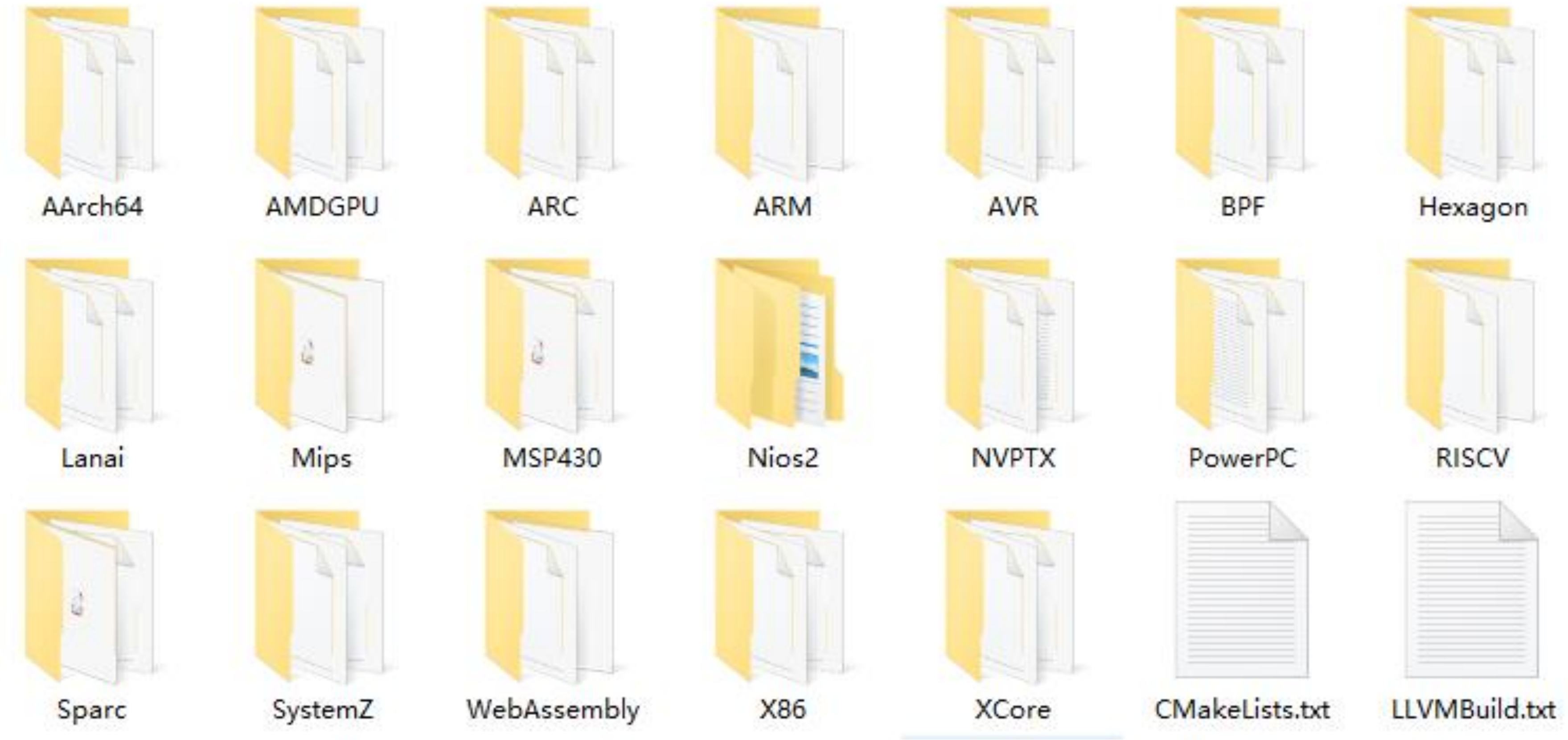


Notes：From LLVM DOC：《Intro to LLVM:Book chapter providing a compiler hacker’s introduction to LLVM》

LLVM所支持的语言

- C， C++， Ruby, Python, Haskell, Java, D, PHP, Pure, Lua,

LLVM所支持的后端



Notes：This is the source code dir of LLVM 8.0.0，its location is LLVM/lib/Target/.

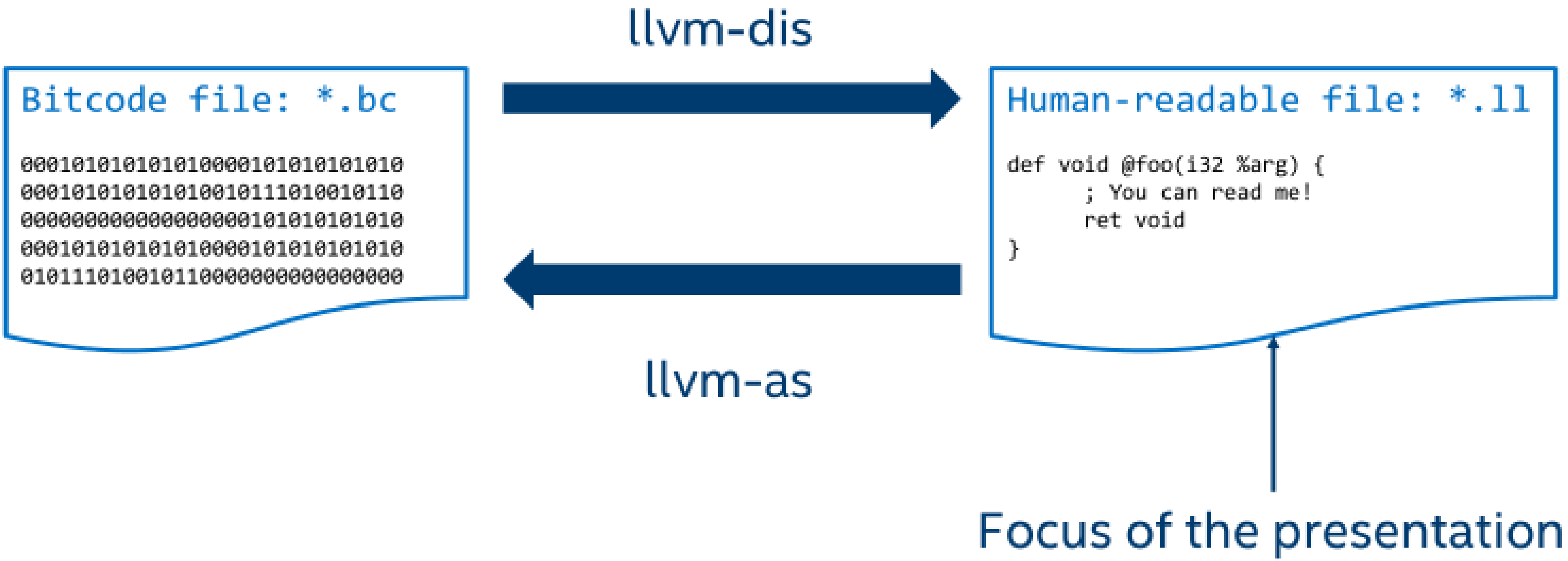
LLVM IR

What is the LLVM IR?

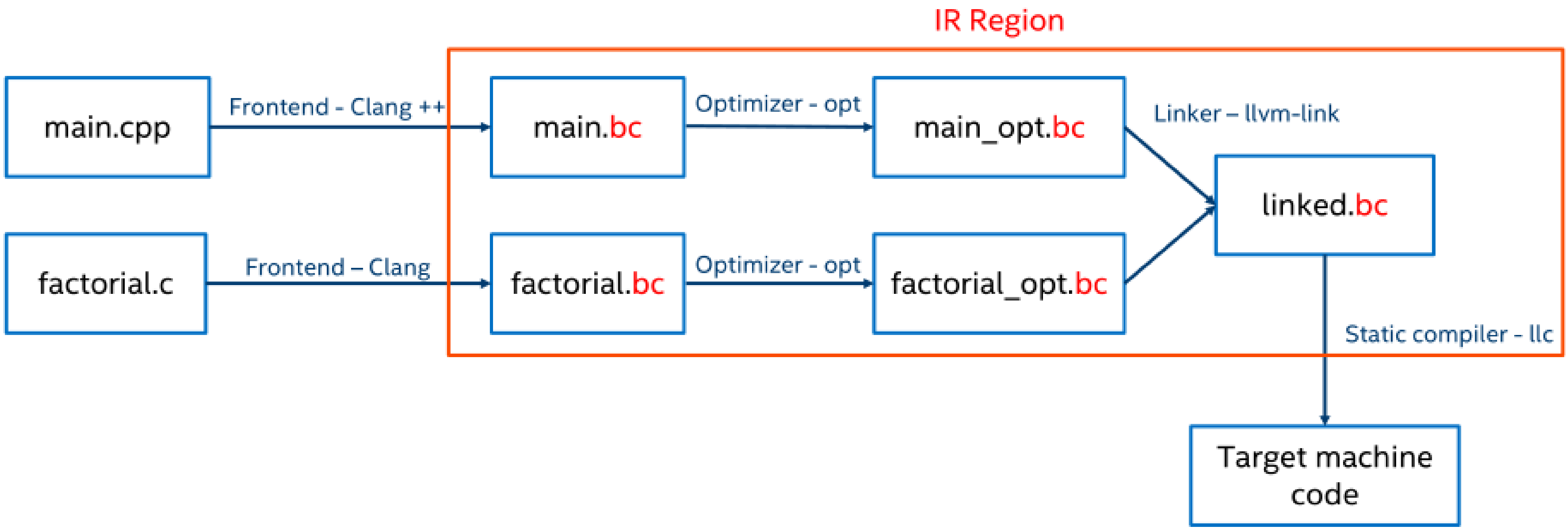
The LLVM **I**ntermediate **R**epresentation:

- Is a low level programming language
  - RISC-like instruction set
- … while being able to represent high-level ideas.
  - i.e. high-level languages can map cleanly to IR.
- Enables efficient code optimization

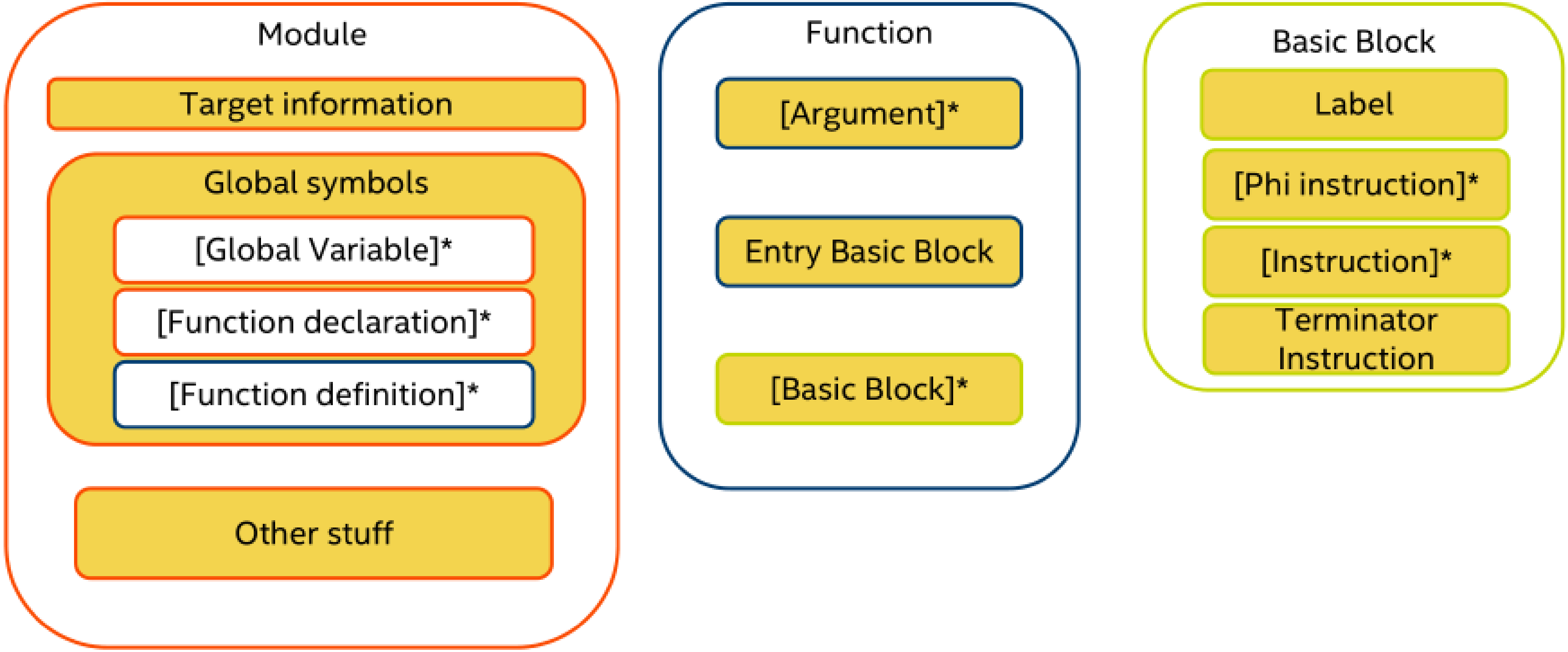
IR representation

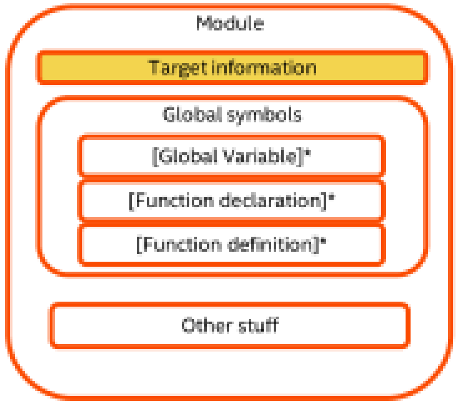


IR & the compilation process



Simplified IR layout





Target information

An IR module usually starts with a pair of strings describing the target:

ELF mangling
Little endian
ABI alignment of i64
Native integer widths

```
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"
```

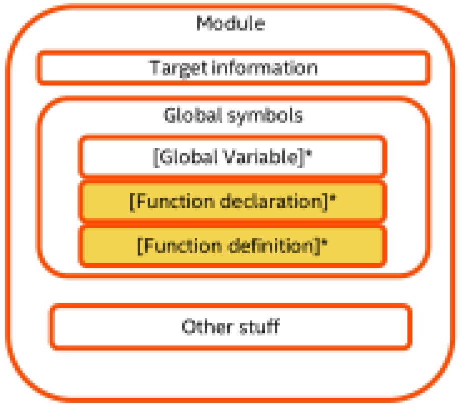
Architecture
Vendor
System
ABI

Global variables

- Name prefixed with “@”.
- Must have a type.
- Must be initialized
- Have the global keyword…
- … xor constant (never stored to!)
  - Not to be confused with C++ const

```
@gv =
@gv = i8
@gv = i8 42   ; Declarations excepted.
@gv = global i8 42
@gv = constant i8 42
```

A basic main program

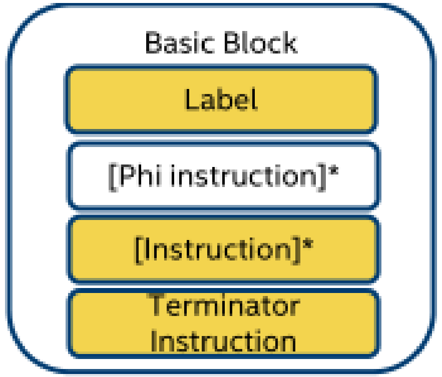


Hand-written IR for this program:

```c
int factorial(int val);

int main(int argc, char** argv)
{
  return factorial(2) * 7 == 42;
}
```

```llvm
declare i32 @factorial(i32)

define i32 @main(i32 %argc, i8** %argv) {
  %1 = call i32 @factorial(i32 2)
  %2 = mul i32 %1, 7
  %3 = icmp eq i32 %2, 42
  %result = zext i1 %3 to i32
  ret i32 %result
}
```

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
}
```

Iterative factorial

```
int factorial(int val) {
  int temp = 1;
  for (int i = 2; i <= val; ++i)
    temp *= i;
  return temp;
}
```

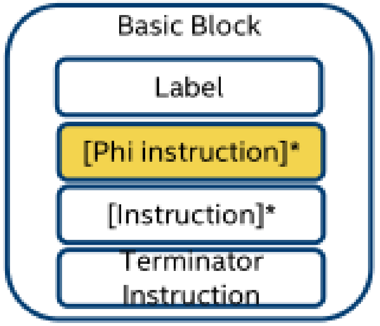
```
define i32 @factorial(i32 %val) {
entry:
  %i = add i32 0, 2
  %temp = add i32 0, 1
  br label %check_for_condition
check_for_condition:

  %i_leq_val = icmp sle i32 %i, %val
  br i1 %i_leq_val, label %for_body, label %end_loop
for_body:

  %new_temp = mul i32 %temp, %i
  %i_plus_one = add i32 %i, 1
  br label %check_for_condition
end_loop:
  ret i32 %temp
}
```

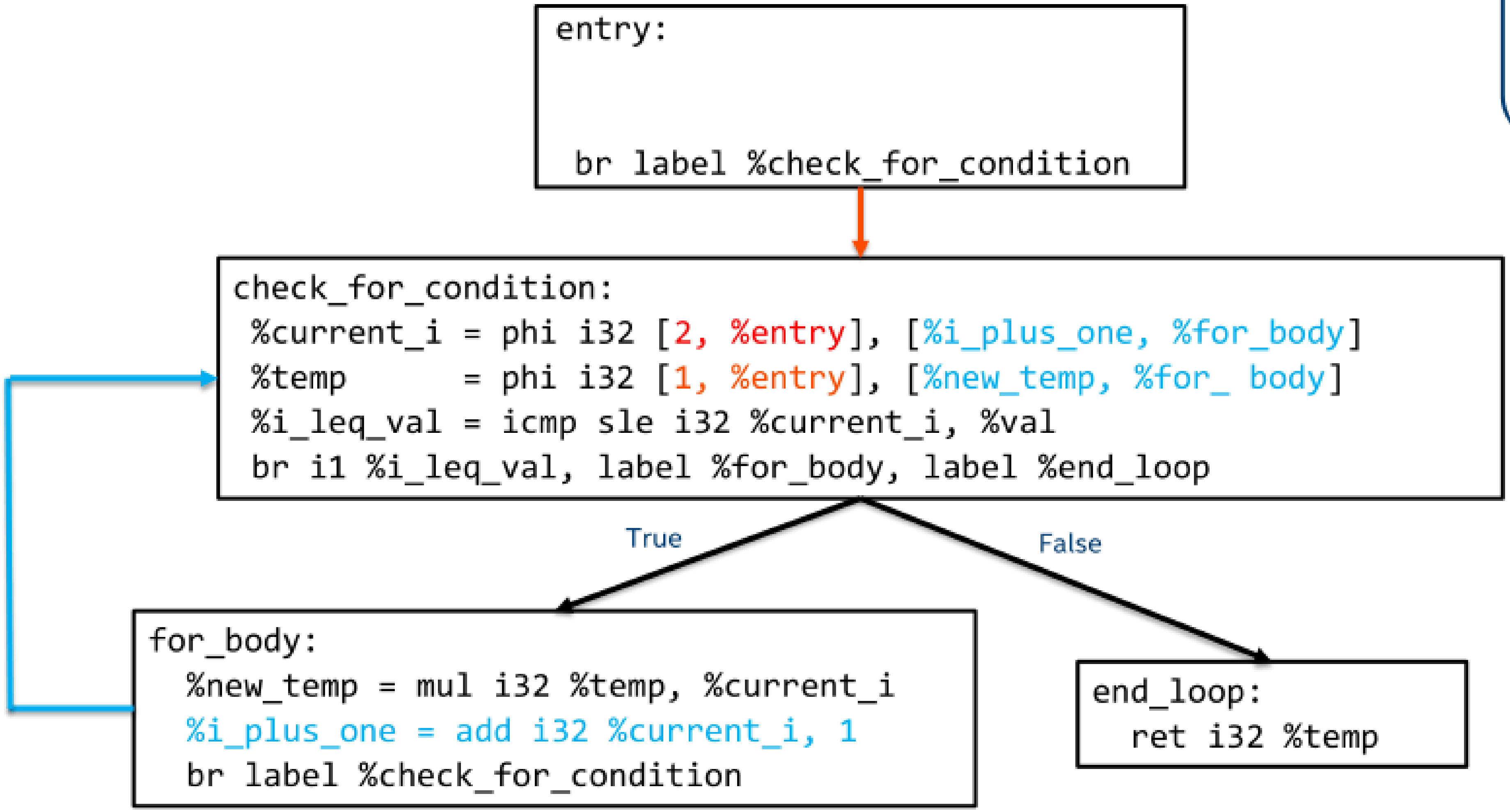
Now *%i* is always *2!*

So you do this:

Now *%temp* is always *1!*

Phis to the rescue!





Allocas to the rescue!

```
entry:
  %i.addr = alloca i32
  %temp.addr = alloca i32
  store i32 2, i32* %i.addr
  store i32 1, i32* %temp.addr
  br label %check_for_condition
```

```
check_for_condition:
  %current_i = load i32, i32* %i.addr
  %temp      = load i32, i32* %temp.addr
  %i_leq_val = icmp sle i32 %current_i, %val
  br i1 %i_leq_val, label %for_body, label %end_loop
```

True

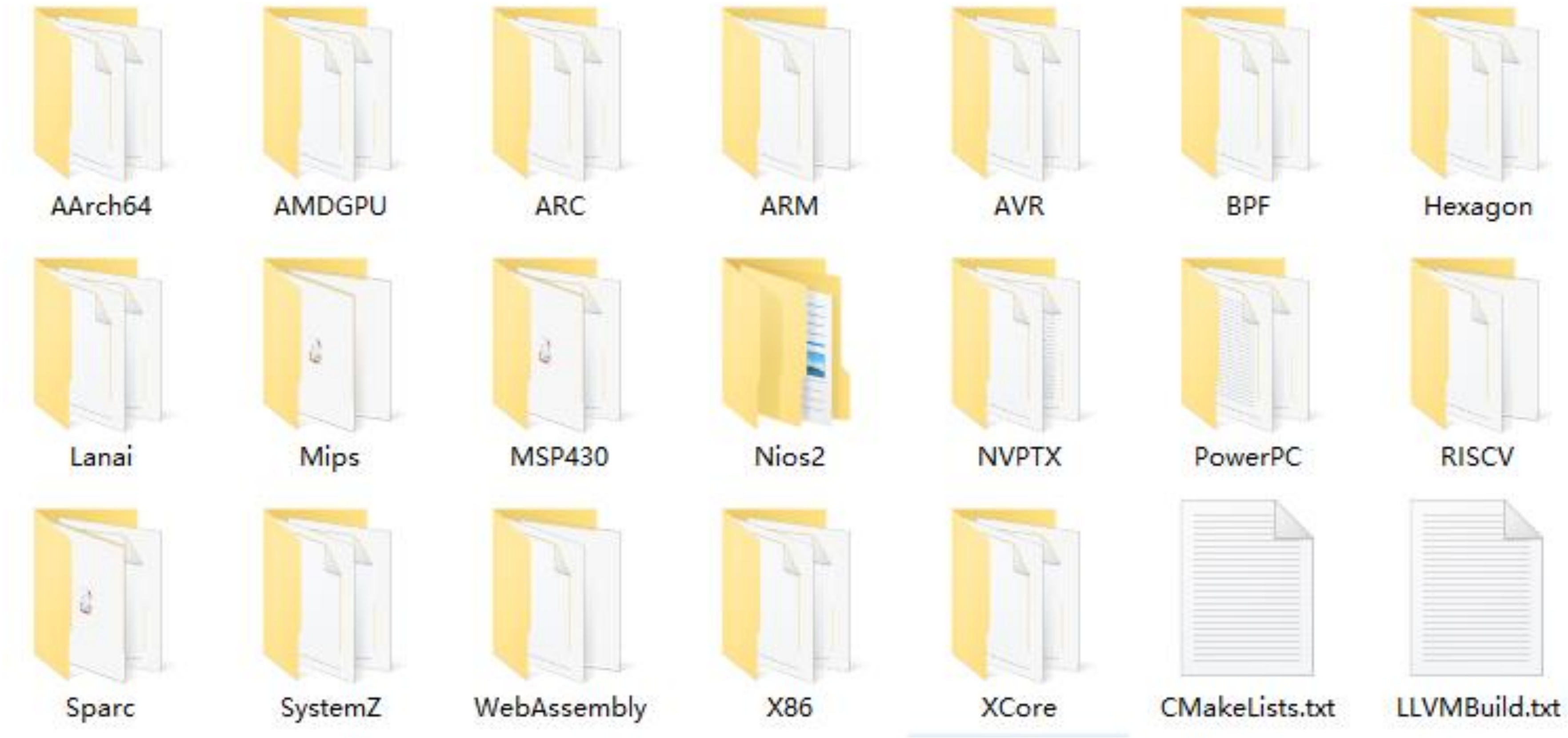
```
for_body:
  %i_plus_one = add i32 %current_i, 1
  %new_temp = mul i32 %temp, %current_i
  store i32 %i_plus_one, i32* %i.addr
  store i32 %new_temp, i32* %temp.addr
  br label %check_for_condition
```

False

```
end_loop:
  ret i32 %temp
```

LLVM Backend

LLVM所支持的后端



Notes：This is the source code dir of LLVM 8.0.0，its location is LLVM/lib/Target/.

The Special Backends

- CBackend (LLVM3.0)

CBackend/	142037	6 years	void	Creating release_30 branch

- CPPBackend (LLVM 3.8)

CppBackend/	257630	2 years	hans	Creating release_38 branch off revision 257626

The C backend does not require register allocation, instruction selection, or any of the other standard components provided by the system. As such, it only implements these two interfaces (TargetMachine and DataLayout), and does its own thing. Note that C backend was removed from the trunk since LLVM 3.1 release. —— 《The LLVM Target-Independent Code Generator》

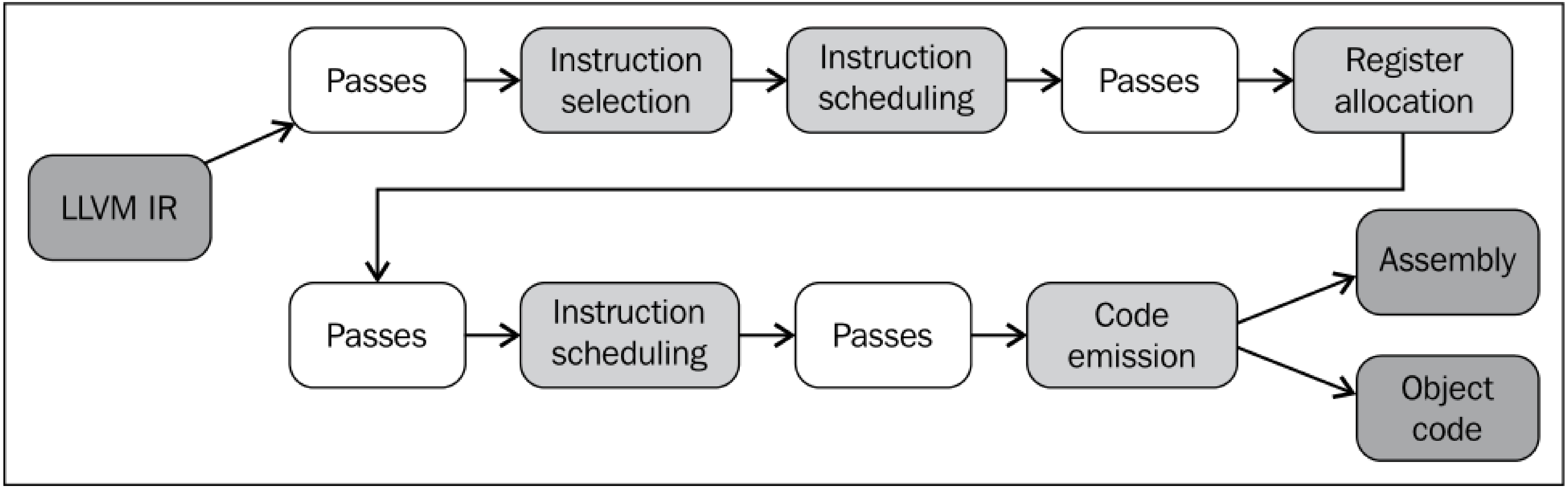
Notes： The screen shots from http://llvm.org/viewvc/llvm-project/.

The Special Backends

- Emscripten compiles LLVM bitcode into JavaScript, which makes it possible to compile C and C++ source code to JavaScript (by first compiling it into LLVM bitcode using Clang), which can be run on the web.Emscripten itself is written in JavaScript.

- WebAssembly backend is presently under development.

The Steps in LLVM Backend



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

LLVM Pass

What is a pass

- The LLVM Pass Framework is an important part of the LLVM system, because LLVM passes are where most of the interesting parts of the compiler exist. Passes perform the transformations and optimizations that make up the compiler, they build the analysis results that are used by these transformations, and they are, above all, a structuring technique for compiler code.

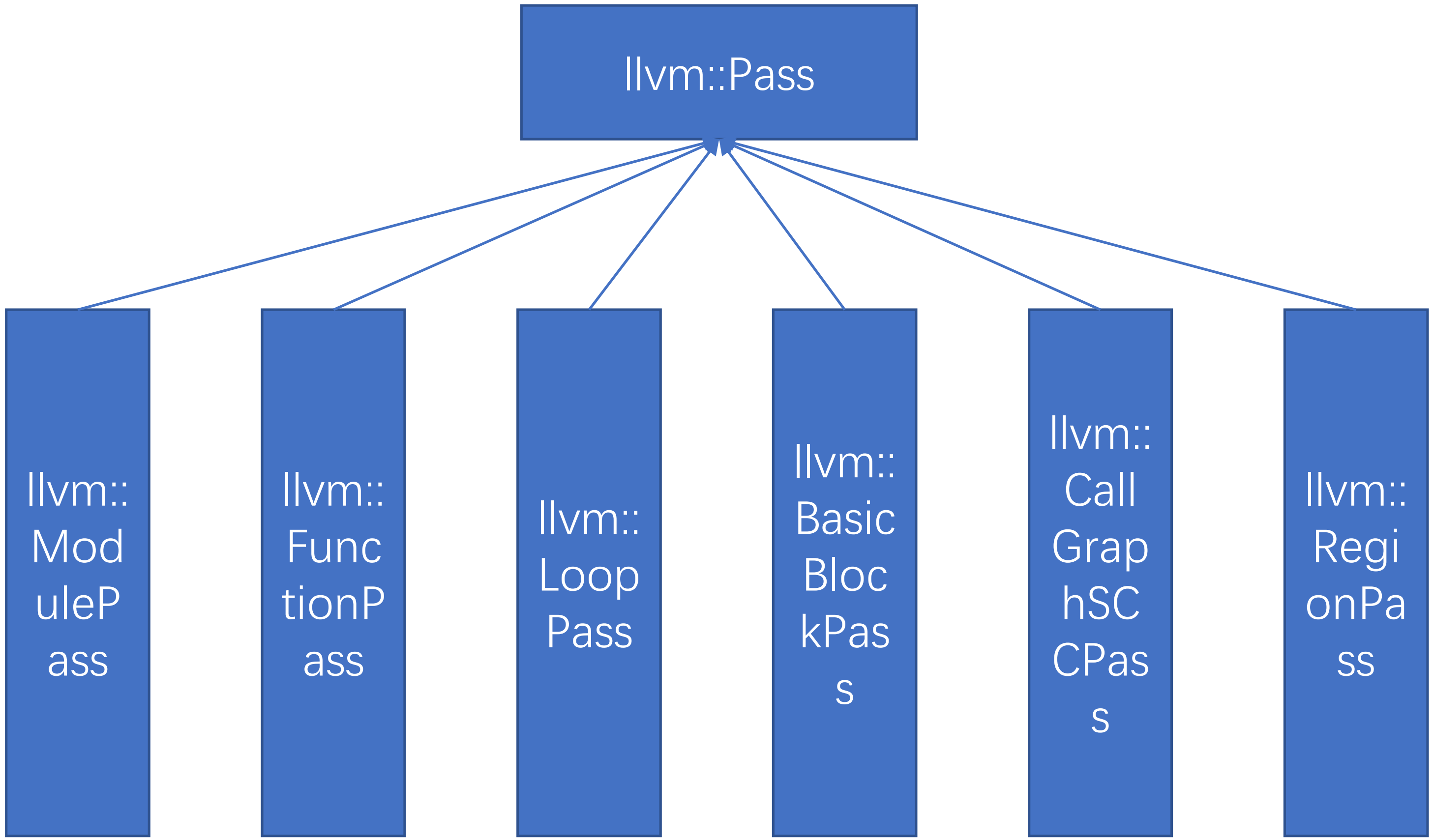
Pass的分类

- Analysis passes compute information that other passes can use or for debugging or program visualization purposes.
- Transform passes can use (or invalidate) the analysis passes. Transform passes all mutate the program in some way.
- Utility passes provides some utility but don’t otherwise fit categorization.

Pass 实例

- lib\Transforms\Hello\Hello.cpp -hello
- Hello.cpp

常用的Pass类及其子类



Pass类

- Pass类的定义 [Pass.h]

- virtual bool doInitialization(Module &) { return false; }

doInitialization - Virtual method overridden by subclasses to do any necessary initialization before any pass is run.

- virtual bool doFinalization(Module &) { return false; }

doFinalization - Virtual method overriden by subclasses to do any necessary clean up after all passes have run.

Pass类

- virtual void getAnalysisUsage(AnalysisUsage &) const;

getAnalysisUsage - This function should be overriden by passes that need analysis information to do their job.

If a pass specifies that it uses a particular analysis result to this function, it can then use the getAnalysis<AnalysisType>() function, below.

FunctionPass

- FunctionPass源码位置：[Pass.h]()

- This class is used to implement most global optimizations. Optimizations should subclass this class if they meet the following constraints:

1. Optimizations are organized globally, i.e., a function at a time.

2. Optimizing a function does not cause the addition or removal of any functions in the module

FunctionPass(文档)

- All FunctionPass execute on each function in the program independent of all of the other functions in the program.(遍历)
- FunctionPasses do not require that they are executed in a particular order.（无序）
- FunctionPasses do not modify external functions.（内部）

FunctionPass（文档）

To be explicit, FunctionPass subclasses are **not** allowed to:

1. Inspect or modify a Function other than the one currently being processed.（单一）
2. Add or remove Functions from the current Module.（添加、删除函数）
3. Add or remove global variables from the current Module.（添加、删除全局变量）
4. Maintain state across invocations of runOnFunction (including global data).（不能跨runOnFunction 维持状态）

FunctionPass（文档）

- virtual bool doInitialization(Module &M);
- virtual bool runOnFunction(Function &F) = 0;
- virtual bool doFinalization(Module &M);

- FunctionPasses may overload three virtual methods to do their work. All of these methods should return true if they modified the program, or false if they didn't.

FunctionPass子类实例1

- LowerAllocations.cpp
- doInitialization - For the lower allocations pass, this ensures that a module contains a declaration for a free function. This function is always successful.

```
bool LowerAllocations::doInitialization(Module &M) {
  const Type *BPTy = Type::getInt8PtrTy(M.getContext());
  FreeFunc =
M.getOrInsertFunction("free"  ,Type::getVoidTy(M.getContext()),
  return true;
}
```

FunctionPass子类实例2

- RegionInfo.h
- class RegionInfoPass : public FunctionPass {
- Detects single entry single exit regions in the control flow graph.
- RegionInfo.cpp
- void RegionInfoPass::getAnalysisUsage(AnalysisUsage &AU) const {

  AU.setPreservesAll();

  AU.addRequiredTransitive<DominatorTreeWrapperPass>();

  AU.addRequired<PostDominatorTreeWrapperPass>();

  AU.addRequired<DominanceFrontierWrapperPass>();

  }

FunctionPass子类实例2

```
bool RegionInfoPass::runOnFunction(Function &F) {
  releaseMemory();

  auto DT = &getAnalysis<DominatorTreeWrapperPass>().getDomTree();
  auto PDT =
&getAnalysis<PostDominatorTreeWrapperPass>().getPostDomTree();
  auto DF =
&getAnalysis<DominanceFrontierWrapperPass>().getDominanceFrontier();

  RI.recalculate(F, DT, PDT, DF);
  return false;
}
```

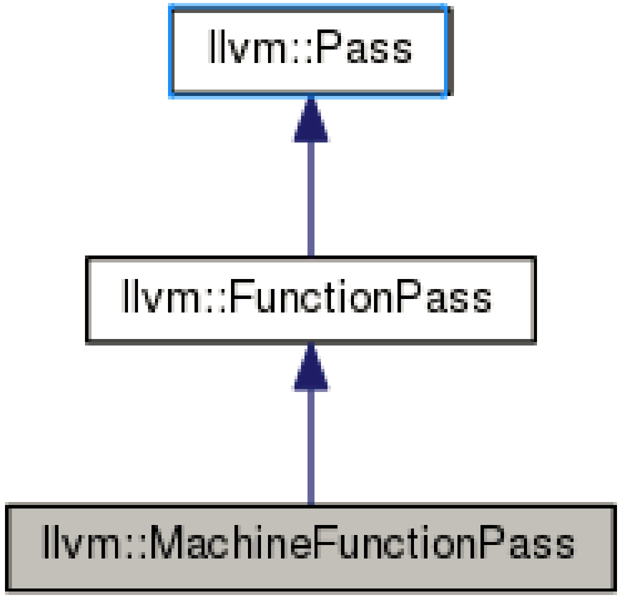
FunctionPass子类实例演示

- hello
- instcount

Region

```
/// A simple control flow graph, that contains two regions.
///
///        1
///       /  |
///      2    |
///     / \    3
///    4   5   |
///    |   |    |
///    6   7  8
///     \  | /
///      \ |/       Region A: 1 -> 9 {1,2,3,4,5,6,7,8}
///        9        Region B: 2 -> 9 {2,4,5,6,7}
```

MachineFunctionPass



MachineFunctionPass 子类的结构图

MachineFunctionPass

- MachineFunctionPass.h

- virtual bool runOnMachineFunction(MachineFunction &MF) = 0;
- runOnMachineFunction - This method must be overloaded to perform the desired machine code transformation or analysis.

MachineFunctionPass子类实例

- MachineLoopInfo
- include/llvm/CodeGen/MachineLoopInfo.h
- MachineLoopInfo() : MachineFunctionPass(ID) {
- bool runOnMachineFunction(MachineFunction &F) override;

MachineFunctionPass子类实例

- MachineLoopInfo

- /llvm/lib/CodeGen/MachineLoopInfo.cpp

```
bool MachineLoopInfo::runOnMachineFunction(MachineFunction &) {
  releaseMemory();
  LI.analyze(getAnalysis<MachineDominatorTree>().getBase());
  return false;
}
```

- MachineFunctionPass子类的使用

Pass Statistic

- The Statistic class is designed to be an easy way to expose various success metrics from passes.
- These statistics are printed at the end of a run, when the **-stats** command line option is enabled on the command line.
- See the Statistics section in the Programmer's Manual for details.

PassManager

- PassManager
- A pass manager is generally a tool to collect a sequence of passes which run over a particular IR construct, and run each of them in sequence over each such construct in the containing IR construct.
- One of the main responsibilities of the PassManager is to make sure that passes interact with each other correctly. Because PassManager tries to optimize the execution of passes it must know how the passes interact with each other and what dependencies exist between the various passes.

PassManager

- If a pass does not implement the getAnalysisUsage method, it defaults to not having any prerequisite passes, and invalidating **all** other passes.
- The PassManager attempts to get better cache and memory usage behavior out of a series of passes by pipelining the passes together. This means that, given a series of consecutive FunctionPass, it will execute all of the FunctionPass on the first function, then all of the FunctionPasses on the second function, etc… until the entire program has been run through the passes.

PassManager

- The PassManager class takes a list of passes, ensures their prerequisites are set up correctly, and then schedules passes to run efficiently. All of the LLVM tools that run passes use the PassManager for execution of these passes.
- An important part of work is that the PassManager tracks the exact lifetime of all analysis results, allowing it to free memory allocated to holding analysis results as soon as they are no longer needed.

LLVM的pass注册机制

- 正常的LLVM中添加Pass，可用的注册方式：

  static RegisterPass<Hello> X("hello", "Hello World Pass");

目前其他方式的注册实例：

INITIALIZE_PASS_BEGIN(RegToMem, "reg2mem", "Demote all values to stack slots", false, false)

INITIALIZE_PASS_DEPENDENCY(BreakCriticalEdges)

INITIALIZE_PASS_END(RegToMem, "reg2mem", "Demote all values to stack slots", false, false)

LLVM的pass注册机制

```
#define INITIALIZE_PASS(passName, arg, name, cfg, analysis)          \
  static void *initialize##passName##PassOnce(PassRegistry &Registry) {    \
    PassInfo *PI = new PassInfo(                                      \
        name, arg, &passName::ID,                                     \
        PassInfo::NormalCtor_t(callDefaultCtor<passName>), cfg, analysis);     \
    Registry.registerPass(*PI, true);                                 \
    return PI;                                                        \
  }
```

llvm/include/llvm/PassSupport.h

LLVM的pass注册机制

- PassRegistry

 This class manages the registration and intitialization of the pass subsystem as application startup, and assists the PassManager

in resolving pass dependencies.

NOTE: PassRegistry is NOT thread-safe. If you want to use LLVM on multiple threads simultaneously, you will need to use a separate PassRegistry on each thread.（非线程安全）

- llvm/include/llvm/PassRegistry.h

LLVM的pass注册机制

- /lib/Passes/PassRegistry.def

This file is used as the registry of passes that are part of the core LLVM libraries.

This file describes both transformation passes and analyses.

Analyses are registered while transformation passes have names registered that can be used when providing a textual pass pipeline.

参考资料

- LLVM doc:《LLVM’s Analysis and Transform Passes》
- LLVM doc:《Writing an LLVM Pass》
- 《Getting Started with LLVM Core Libraries》
- 《Tutorial Bridgers：LLVM IR tutorial》
- 《Brief Intro to LLVM Backend》

Q&A