

LLVM CodeGen代码详解

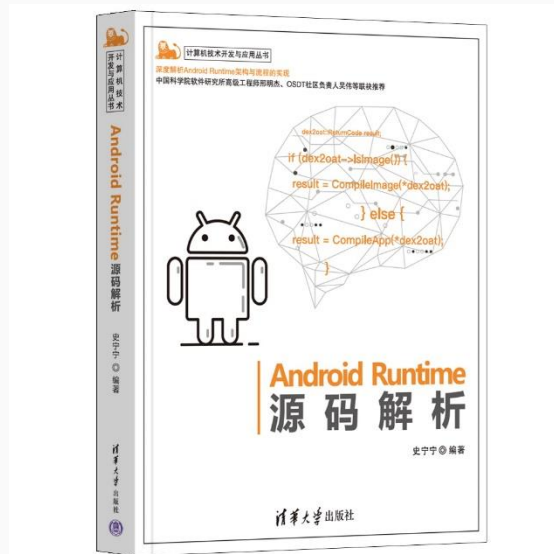
史宁宁

PLCT实验室

shiningning@iscas.ac.cn

自我介绍

史宁宁，PLCT实验室项目主管。2012年起，作为核心开发人员和项目经理参与组织开发多个编译器项目。长期坚持撰写技术博客，其中“LLVM每日谈”、“方舟编译器学习笔记”等系列影响较大，曾出版图书3部。目前活跃在OSDT/HelloGCC/HelloLLVM、方舟编译器、RISC-V等开源社区，主要研究内容为Clang/LLVM、JVM/ART等。



Github: [shining1984](https://github.com/shining1984)

知乎: 编译船夫 (曾用名: 小乖他爹)

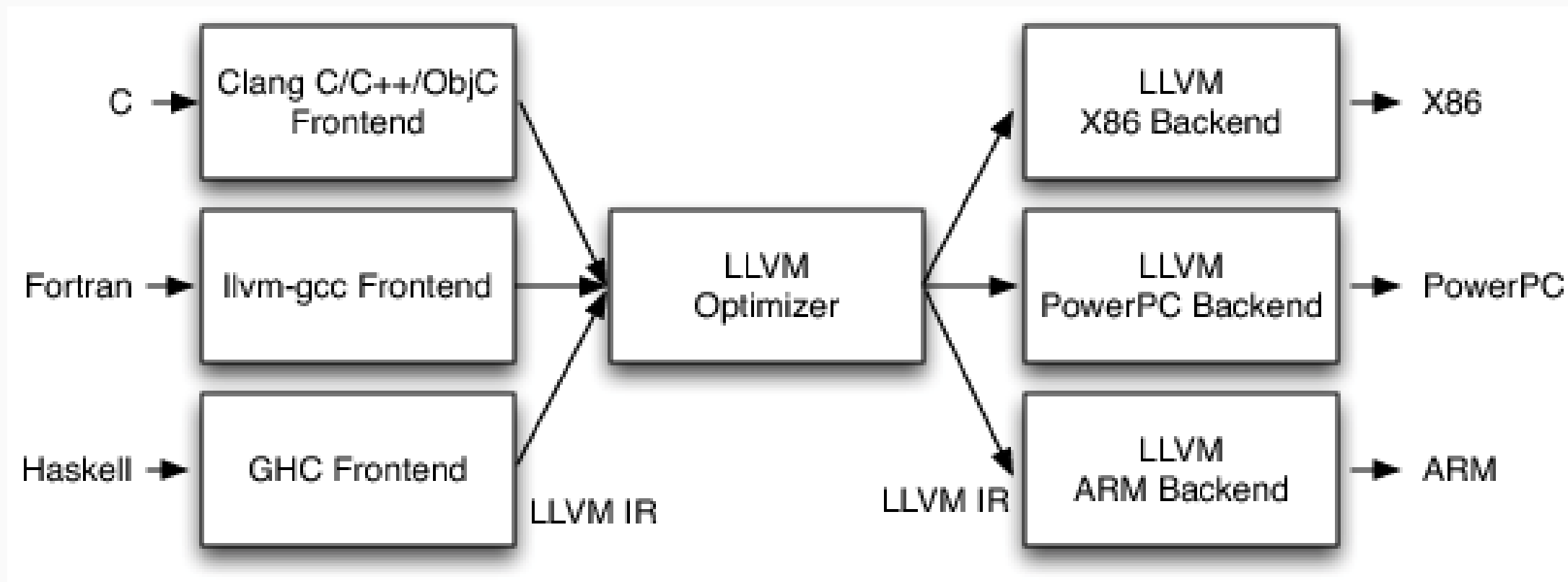
邮箱: shiningning@iscas.ac.cn

微信号: shiningning_talk

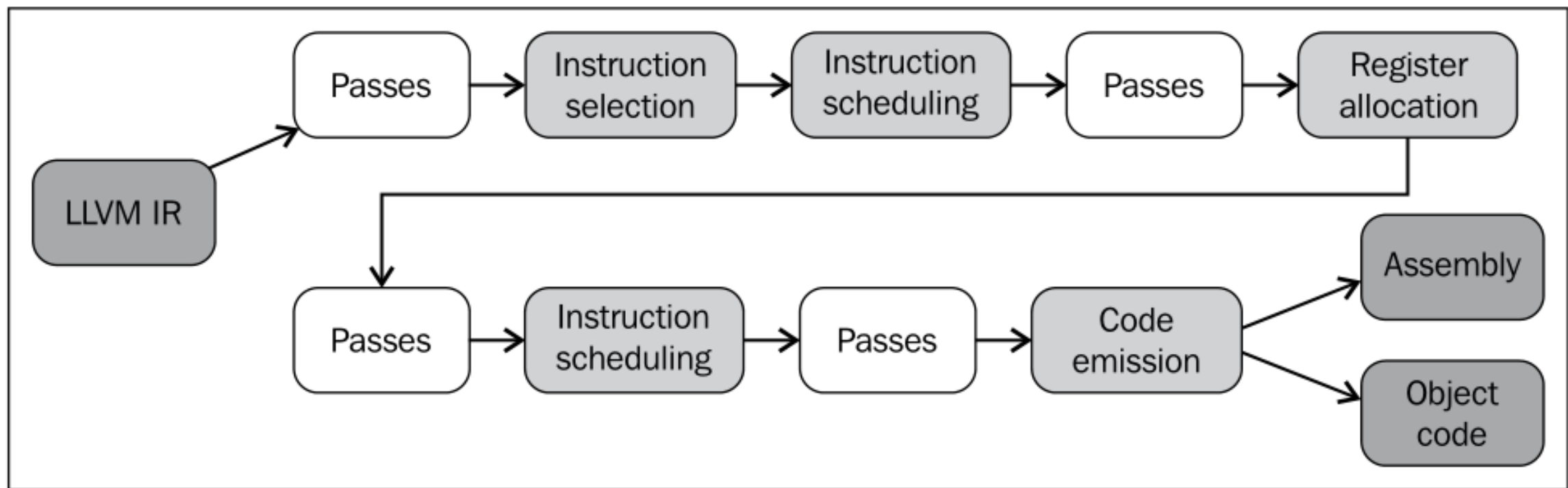
目录

1. LLVM后端的架构与组成
2. LLVM CodeGen中的指令选择实现
3. LLVM CodeGen中的指令调度实现
4. LLVM CodeGen中的寄存器分配实现
5. 小结

LLVM的架构

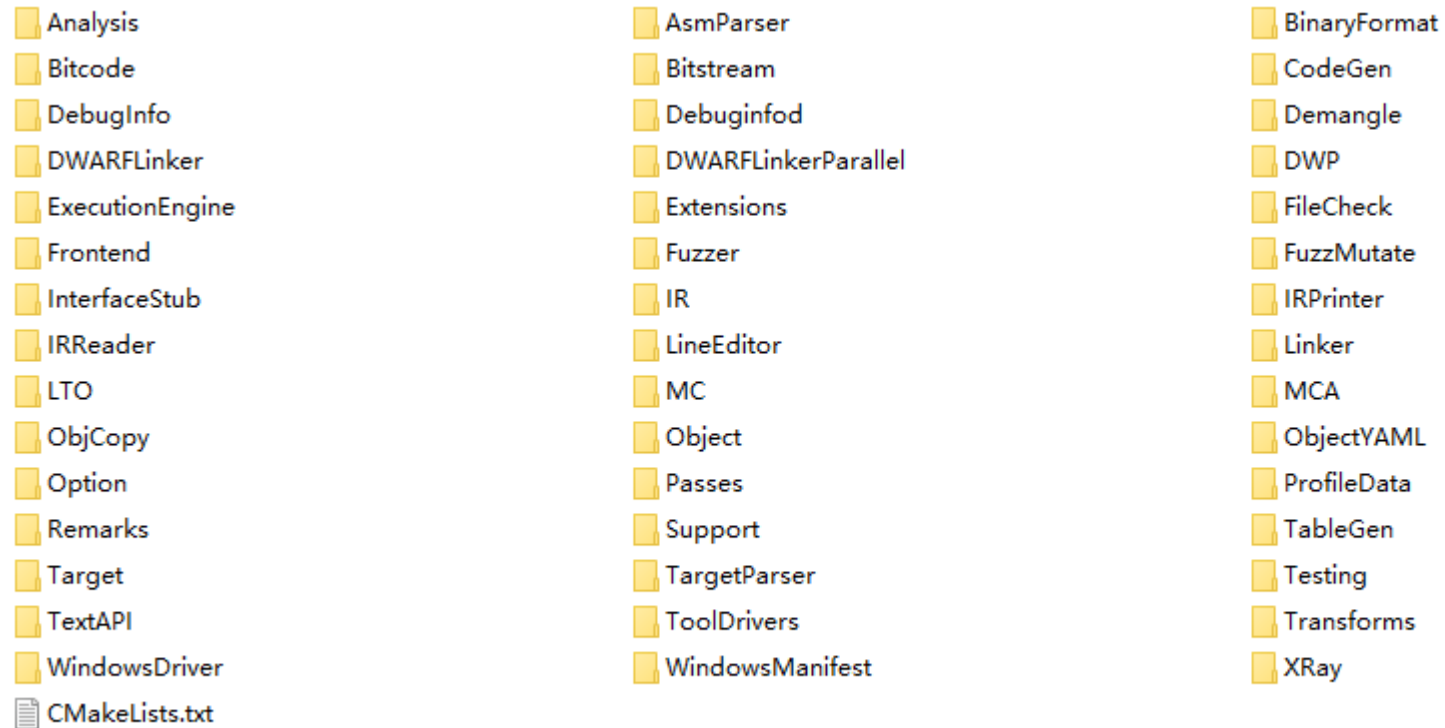


The Steps in LLVM Backend



LLVM libs

everything is a library



Analysis	AsmParser	BinaryFormat
Bitcode	Bitstream	CodeGen
DebugInfo	Debuginfod	Demangle
DWARFLinker	DWARFLinkerParallel	DWP
ExecutionEngine	Extensions	FileCheck
Frontend	Fuzzer	FuzzMutate
InterfaceStub	IR	IRPrinter
IRReader	LineEditor	Linker
LTO	MC	MCA
ObjCopy	Object	ObjectYAML
Option	Passes	ProfileData
Remarks	Support	TableGen
Target	TargetParser	Testing
TextAPI	ToolDrivers	Transforms
WindowsDriver	WindowsManifest	XRay
CMakeLists.txt		

目录

1. LLVM后端的架构与组成
2. LLVM CodeGen中的指令选择实现
3. LLVM CodeGen中的指令调度实现
4. LLVM CodeGen中的寄存器分配实现
5. 小结

LLVM的指令选择

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.

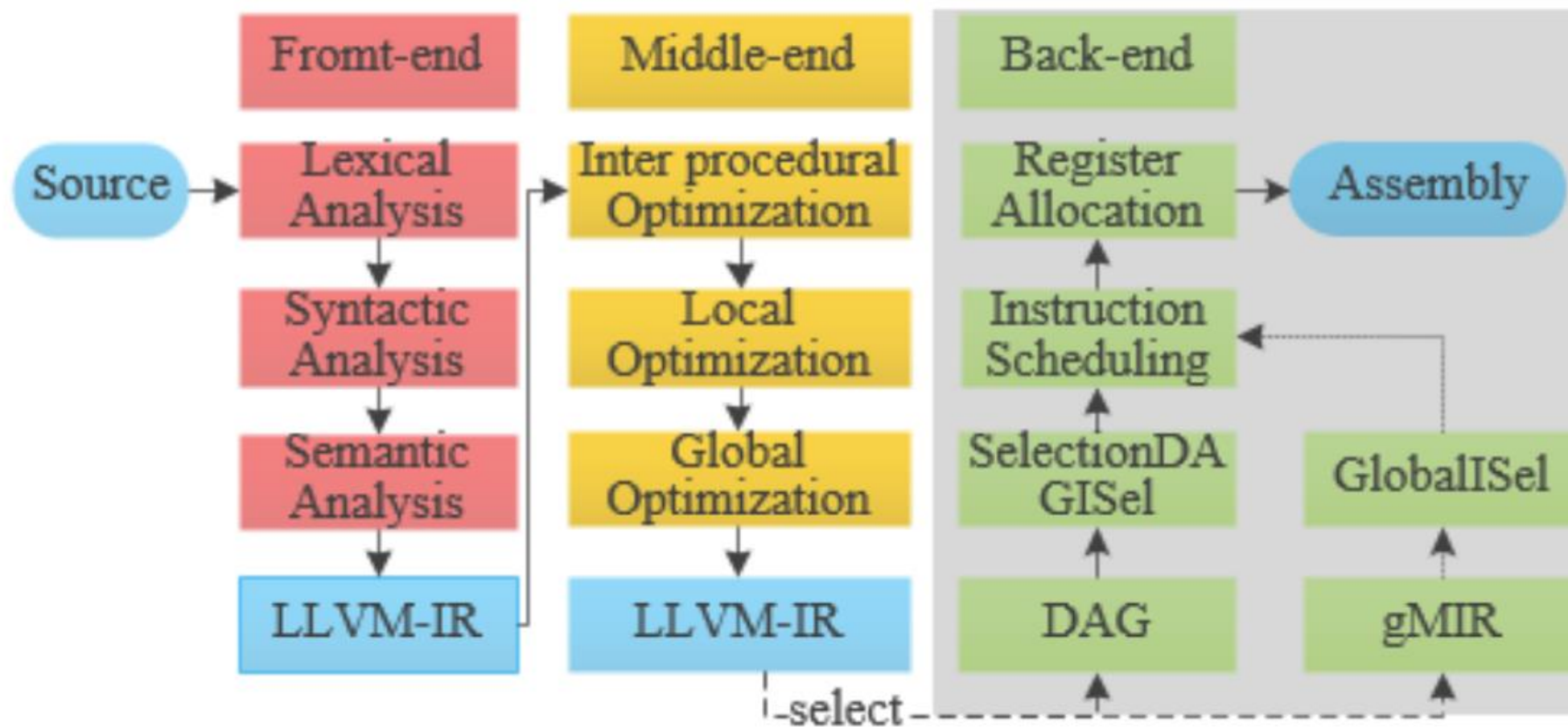
<https://www.llvm.org/docs/CodeGenerator.htm>

GlobalISel

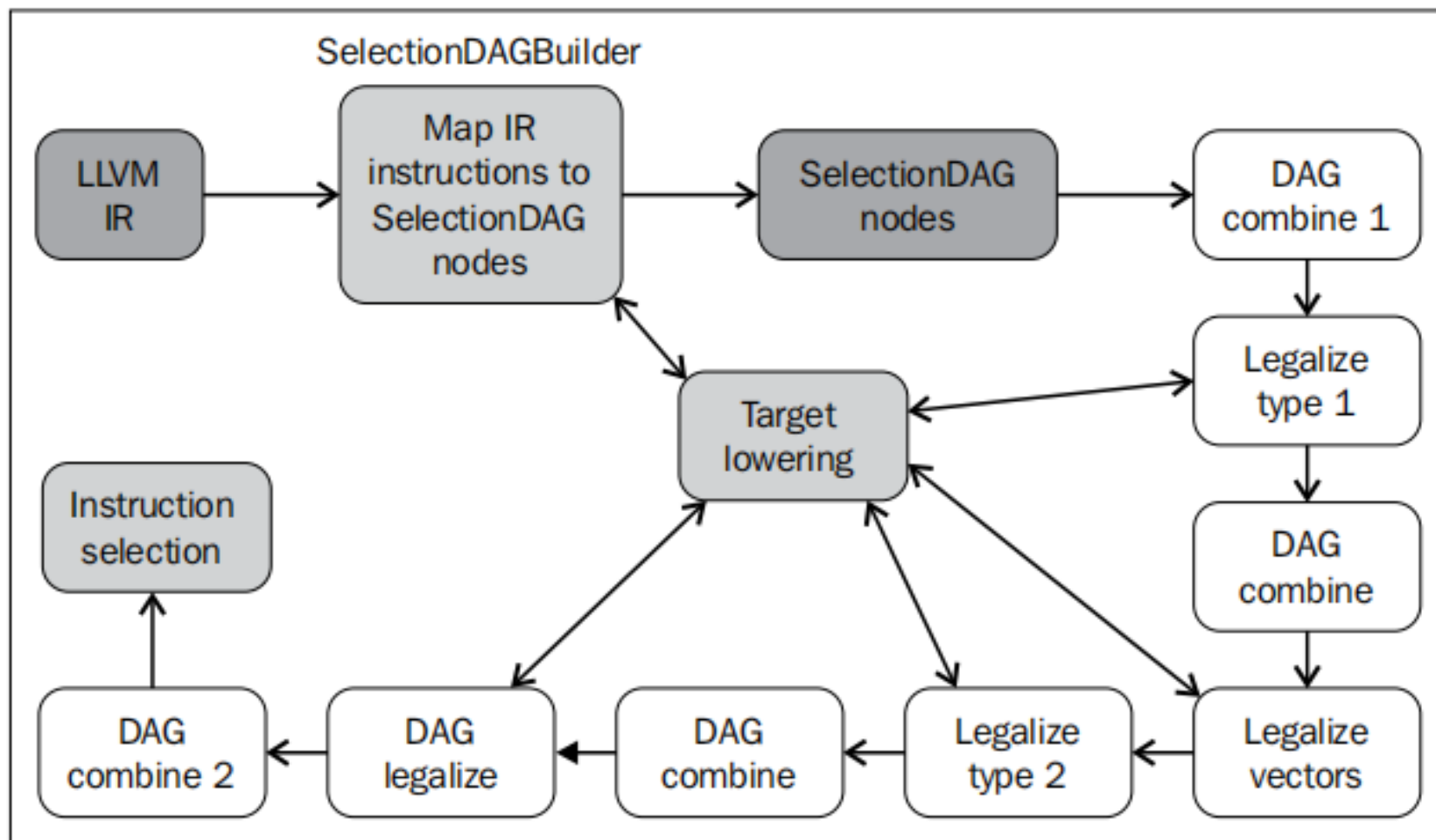
GlobalISel is a framework that provides a set of reusable passes and utilities for instruction selection — translation from LLVM IR to target-specific Machine IR (MIR). GlobalISel is intended to be a replacement for SelectionDAG and FastISel.

<https://llvm.org/docs/GlobalSel/index.html>

LLVM的指令选择（续）



SelectionDAG指令选择



LLVM CodeGen中的SelectionDAG指令选择

1. LLVM CodeGen中涵盖了SelectionDAG指令选择的公共代码（非目标架构相关代码）部分。
2. LLVM CodeGen中有专门的llvm/lib/CodeGen/SelectionDAG目录来存放SelectionDAG指令选择相关部分的代码；
3. SelectionDAG指令选择的代码，主要靠SelectionDAGISel类来作为入口，引导其完成指令选择部分的功能；
4. SelectionDAGISel类在实现上，是MachineFunctionPass的子类，所以它也是一个pass，它的入口函数是其runOnMachineFunction()；
5. SelectionDAGISel类的实现位于llvm/lib/CodeGen/SelectionDAG/ SelectionDAGISel.h和llvm/lib/CodeGen/SelectionDAG/SelectionDAGISel.cpp；

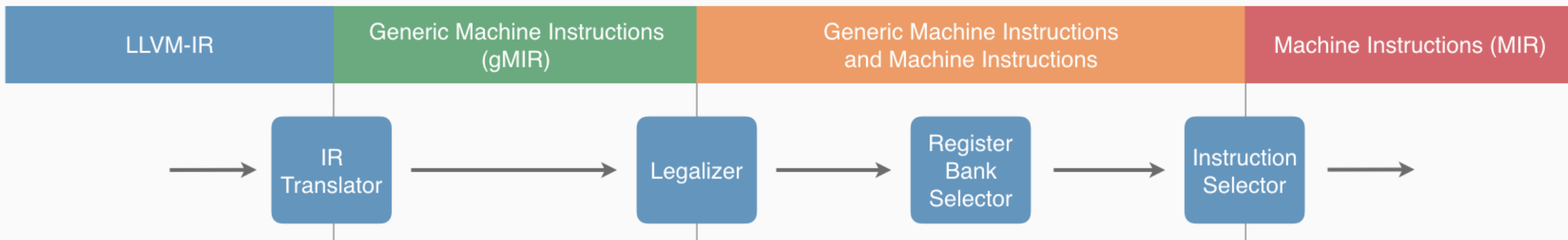
LLVM CodeGen中的SelectionDAG指令选择（续）

```
void SelectionDAGISel::CodeGenAndEmitDAG() {  
    ...  
    CurDAG->Combine(BeforeLegalizeTypes, AA, OptLevel);  
    ...  
    Changed = CurDAG->LegalizeTypes();  
    ...  
    CurDAG->Combine(AfterLegalizeTypes, AA, OptLevel);  
    ...  
    Changed = CurDAG->LegalizeVectors();  
    ...  
    CurDAG->LegalizeTypes();  
    ...  
    CurDAG->Combine(AfterLegalizeVectorOps, AA, OptLevel);  
    ...  
    CurDAG->Legalize();  
    ...  
    CurDAG->Combine(AfterLegalizeDAG, AA, OptLevel);  
    ...  
    DoInstructionSelection();  
    ...  
}
```

LLVM CodeGen中的FastISel指令选择

1. LLVM的FastISel指令选择，设计目标就是为了快速生成比较简易的代码，所以不会做太多的降级、优化，必然就会不支持对一些类型进行合法化，通常是用在编译器的“-O0”模式下；
2. LLVM CodeGen中的FastISel指令选择也是实现了目标指令集无关的公共代码；
3. LLVM CodeGen中的FastISel指令选择在某些没实现的功能上，还会调用SelectionDAG selector的相关代码来补充。它的代码实现位于llvm/include/llvm/CodeGen/FastISel.h和llvm/lib/CodeGen/SelectionDAG/FastISel.cpp之中，对应着具体的FastISel类；
4. 和之前的SelectionDAG指令选择不同，FastISel类并不是一个pass，它就是一个正常的类，不是MachineFunctionPass的子类；
5. LLVM CodeGen中的FastISel指令选择的调用流程，是和SelectionDAG指令选择按照一个流程混合在一起的，并不是单独的流程；

GlobalSel指令选择



LLVM CodeGen中的GlobalSel指令选择

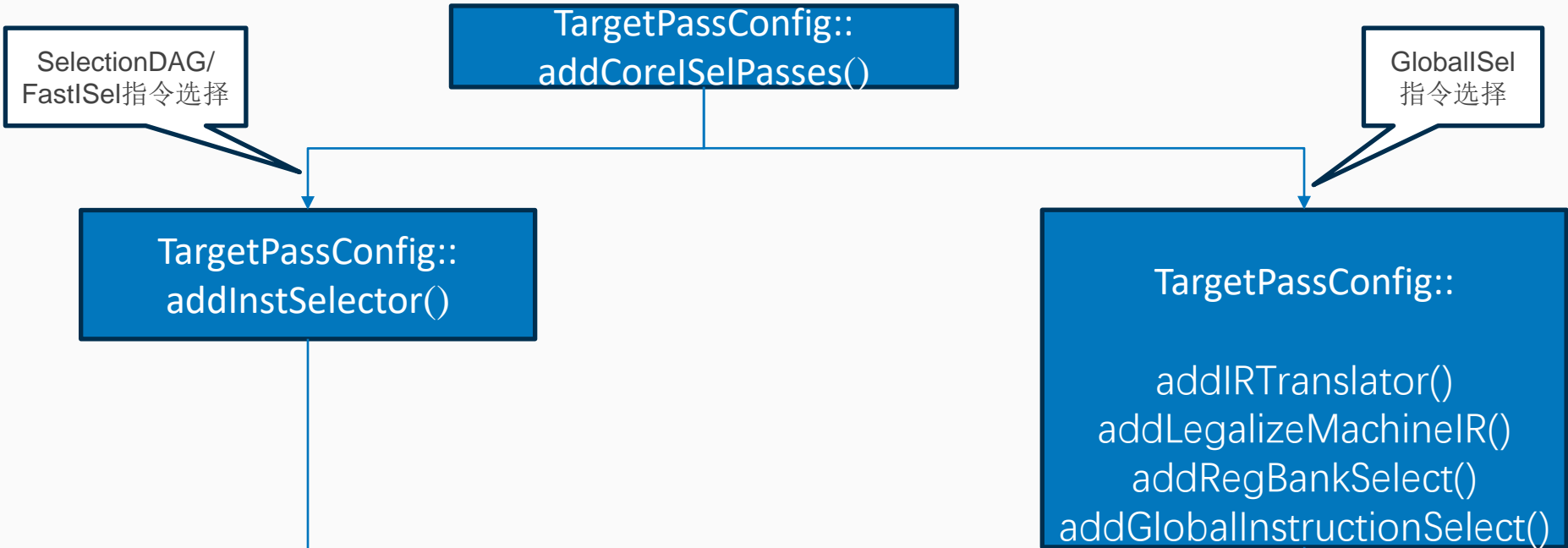
1. LLVM CodeGen中的GlobalSel指令选择也是目标指令集架构无关的公共代码，所有后端的目标指令集相关代码都是基于这个代码之上的；
2. LLVM CodeGen中的GlobalSel指令选择有一个专门的目录llvm/lib/ CodeGen/GlobalSel来存放相关代码；
3. LLVM CodeGen中的GlobalSel指令选择，根据结构图，是分为四个pass来进行实现的，它们分别是：IRTranslator、Legalizer、RegBankSelect和InstructionSelect；
4. 上述四个pass，都是以runOnMachineFunction()的为入口函数，且每个pass都有个同名的实现类；
5. IRTranslator类的实现位于llvm/include/llvm/CodeGen/GlobalSel/IRTranslator.h和llvm/lib/CodeGen/GlobalSel/IRTranslator.cpp；
6. Legalizer类的实现位于llvm/include/llvm/CodeGen/GlobalSel/Legalizer.h和llvm/lib/CodeGen/GlobalSel/Legalizer.cpp；

LLVM CodeGen中的GlobalSel指令选择（续）

7. RegBankSelect类的实现代码位于llvm/include/llvm/CodeGen/GlobalSel/RegBankSelect.h和llvm/lib/CodeGen/GlobalSel/RegBankSelect.cpp;
8. InstructionSelect类的实现代码位于llvm/include/llvm/CodeGen/GlobalSel/InstructionSelect.h和llvm/lib/CodeGen/GlobalSel/InstructionSelect.cpp;
9. InstructionSelect类负责实现指令选择;

LLVM CodeGen中的指令选择的调用流程

llvm/lib/CodeGen/TargetPassConfig.cpp



```
RISCVPassConfig::addInstSelector() {  
  addPass(createRISCVISelDag(...));  
  ...  
}
```

llvm/lib/Target/RISCV/RISCVTargetMachine.cpp

非LLVM
CodeGen
部分

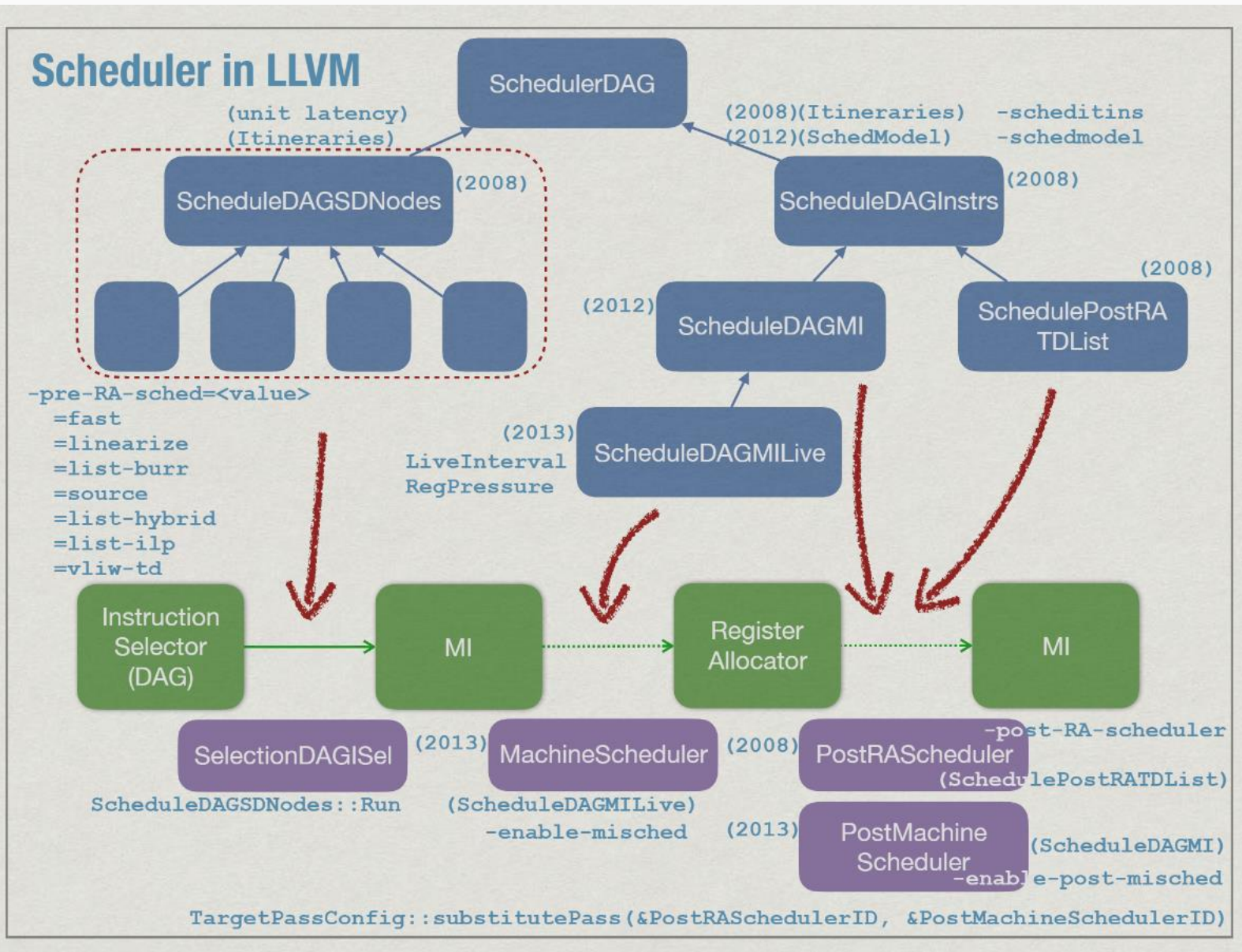
```
RISCVPassConfig::addIRTranslator() {  
  addPass(new IRTranslator(...));  
  ...  
}
```

以RISC-V
为例

目录

1. LLVM后端的架构与组成
2. LLVM CodeGen中的指令选择实现
3. LLVM CodeGen中的指令调度实现
4. LLVM CodeGen中的寄存器分配实现
5. 小结

LLVM的指令调度



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

SelectionDAGISel指令调度

1. SelectionDAGISel指令调度的使用场景是介于Instruction Selector和MI之间，它可以通过参数“-pre-RA-sched”来打开，也可以通过“-pre-RA-sched=<value>”这种形式，传递不同的value值，来选择具体的调度算法实现；
2. SelectionDAGISel作为一个类并不是一个具体的调度器实现，而是一个基于SelectionDAG的指令选择的基础实现类，其中包含了基于SelectionDAG的指令调度；
3. 具体的指令调度算法实现，根据“-pre-RA-sched=<value>”中不同的value值，有不同的实现类；
4. SelectionDAGISel本身包含了基于SelectionDAG的指令选择和指令调度，可以认为SelectionDAGISel指令调度是专门针对SelectionDAG指令选择而进行的指令调度；

LLVM CodeGen中 SelectionDAGISel指令调度

序号	value	对象名称	实现类	位置	描述
1	default	defaultListDAGScheduler		llvm/lib/CodeGen/SelectionDAG/SelectionDAGISel.cpp	Best scheduler for the target
2	fast	fastDAGScheduler	ScheduleDAGFast	llvm/lib/CodeGen/SelectionDAG/ScheduleDAGFast.cpp	Fast suboptimal list scheduling
3	linearize	linearizeDAGScheduler	ScheduleDAGLinearize	llvm/lib/CodeGen/SelectionDAG/ScheduleDAGFast.cpp	Linearize DAG, no scheduling
4	list-burr	burrListDAGScheduler	ScheduleDAGRRLList	llvm/lib/CodeGen/SelectionDAG/ScheduleDAGRRLList.cpp	Bottom-up register reduction list scheduling
5	source	sourceListDAGScheduler	ScheduleDAGRRLList	llvm/lib/CodeGen/SelectionDAG/ScheduleDAGRRLList.cpp	Similar to list-burr but schedules in source order when possible
6	list-hybrid	hybridListDAGScheduler	ScheduleDAGRRLList	llvm/lib/CodeGen/SelectionDAG/ScheduleDAGRRLList.cpp	Bottom-up register pressure aware list scheduling which tries to balance latency and register pressure
7	list-ilp	ILPListDAGScheduler	ScheduleDAGRRLList	llvm/lib/CodeGen/SelectionDAG/ScheduleDAGRRLList.cpp	Bottom-up register pressure aware list scheduling which tries to balance ILP and register pressure
8	vliw-td	VLIWScheduler	ScheduleDAGVLIW	llvm/lib/CodeGen/SelectionDAG/ScheduleDAGVLIW.cpp	VLIW scheduler

LLVM CodeGen中 SelectionDAGISel指令调度（续）

```
/// llvm/lib/CodeGen/SelectionDAG/SelectionDAGSel.cpp
```

```
void SelectionDAGISel::CodeGenAndEmitDAG() {  
    ...  
    // Schedule machine code.  
    ScheduleDAGSDNodes *Scheduler = CreateScheduler();  
    {  
        NamedRegionTimer T("sched", "Instruction Scheduling", GroupName, GroupDescription,  
                           TimePassesIsEnabled);  
        Scheduler->Run(CurDAG, FuncInfo->MBB);  
    }  
    ...  
}
```

MachineScheduler&&PostMachineScheduler

1. MachineScheduler和PostMachineScheduler这两个调度器，都是基于Machine Instruction进行调度的，区别是MachineScheduler在寄存器分配之前，PostMachineScheduler在寄存器分配之后；
2. MachineScheduler调度的是消除过phi节点的machine instruction，并且保留了Liveness，这样在寄存器分配的时候可以使用这些信息；
3. MachineScheduler和PostMachineScheduler都有同名的类，相关代码位于llvm/lib/CodeGen/MachineScheduler.cpp之中；
4. MachineScheduler和PostMachineScheduler都是MachineSchedulerBase的子类，MachineSchedulerBase又继承于MachineFunctionPass，所以MachineScheduler和PostMachineScheduler也是MachineFunctionPass，入口函数都是runOnMachineFunction();

LLVM CodeGen中的MachineScheduler&&PostMachineScheduler代码实现

```
/// llvm/lib/CodeGen/MachineScheduler.cpp
```

```
bool MachineScheduler::runOnMachineFunction(MachineFunction &mf) {  
    ...  
    std::unique_ptr<ScheduleDAGInstrs> Scheduler(createMachineScheduler());  
    scheduleRegions(*Scheduler, false);  
    ...  
}
```

```
bool PostMachineScheduler::runOnMachineFunction(MachineFunction &mf) {  
    ...  
    std::unique_ptr<ScheduleDAGInstrs> Scheduler(createPostMachineScheduler());  
    scheduleRegions(*Scheduler, true);  
    ...  
}
```

```
/// Main driver for both MachineScheduler and PostMachineScheduler.  
void MachineSchedulerBase::scheduleRegions(ScheduleDAGInstrs &Scheduler, bool FixKillFlags) {  
    ...  
}
```


PostRAScheduler指令调度

1. PostRAScheduler有同名的类，声明和实现位于Illum/lib/CodeGen/PostRASchedulerList.cpp中；
2. PostRAScheduler也是继承于MachineFunctionPass，所以它也是个pass，且入口函数也是runOnMachineFunction()；
3. PostRAScheduler在实际调度的时候，会和SchedulePostRATDList 类进行互动，通过SchedulePostRATDList实现具体的调度；

LLVM CodeGen中PostRAScheduler指令调度的源码实现

```
/// llvm/lib/CodeGen/PostRASchedulerList.cpp
```

```
bool PostRAScheduler::runOnMachineFunction(MachineFunction &Fn) {
```

```
...
```

```
    SchedulePostRATDList Scheduler(Fn, MLI, AA, RegClassInfo, AntiDepMode, CriticalPathRCs);
```

```
...
```

```
    Scheduler.enterRegion(&MBB, MBB.begin(), Current, CurrentCount);
```

```
    Scheduler.setEndIndex(CurrentCount);
```

```
    Scheduler.schedule();
```

```
    Scheduler.exitRegion();
```

```
    Scheduler.EmitSchedule();
```

```
...
```

```
}
```

```
/// Schedule - Schedule the instruction range using list scheduling.
```

```
///
```

```
void SchedulePostRATDList::schedule() {
```

```
...
```

```
}
```

LLVM CodeGen中的指令调度的执行流程

llvm/lib/CodeGen/TargetPassConfig.cpp

TargetPassConfig::
addMachinePasses()



```
graph TD; A[TargetPassConfig::addMachinePasses()] --> B[TargetPassConfig::addOptimizedRegAlloc() {  
...  
// PreRA instruction scheduling.  
addPass(&MachineSchedulerID);  
...  
}]; A --> C[TargetPassConfig::addFastRegAlloc()]; B --> D["if (MISchedPostRA)  
addPass(&PostMachineSchedulerID);  
else  
addPass(&PostRASchedulerID);"]; C --> D;
```

The flowchart illustrates the execution flow of the `addMachinePasses()` function in `TargetPassConfig.cpp`. It starts with the function call, which branches into two parallel paths: `addOptimizedRegAlloc()` and `addFastRegAlloc()`. Both paths then converge into a single conditional block that checks for `MISchedPostRA` and adds the appropriate scheduler pass.

```
TargetPassConfig::addOptimizedRegAlloc() {  
...  
// PreRA instruction scheduling.  
addPass(&MachineSchedulerID);  
...  
}
```

TargetPassConfig::
addFastRegAlloc()

```
if (MISchedPostRA)  
addPass(&PostMachineSchedulerID);  
else  
addPass(&PostRASchedulerID);
```

LLVM指令调度器的开关

指令调度器	选项
SelectionDAGISel	-pre-RA-sched
MachineScheduler	-enable-misched
PostMachineScheduler	-enable-post-misched
PostRAScheduler	-post-RA-sched

目录

1. LLVM后端的架构与组成
2. LLVM CodeGen中的指令选择实现
3. LLVM CodeGen中的指令调度实现
4. LLVM CodeGen中的寄存器分配实现
5. 小结

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.

- **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.

LLVM CodeGen中的fast寄存器分配

1. fast寄存器分配，对应一个fast寄存器分配器，这个fast寄存器分配器有一个专门的对应类来实现，这个类叫RegAllocFast，RegAllocFast的实现代码位于llvm/lib/CodeGen/RegAllocFast.cpp之中；
2. RegAllocFast继承于MachineFunctionPass，它也是一个MachineFunctionPass，入口函数也是runOnMachineFunction();
3. 函数runOnMachineFunction()的核心动作，是通过对每个MachineBasicBlock 运行allocateBasicBlock()来实现的；
4. allocateBasicBlock()的核心代码部分，是通过逆向顺序遍历MachineBasicBlock中的指令，挨个进行寄存器分配，具体的动作都是在allocateInstruction()中实现的；
5. 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.

LLVM CodeGen中的basic寄存器分配

1. basic寄存器分配是基于Poletto有关线性扫描论文（Linear Scan Register Allocation, 1999）的一个扩展算法实现；
2. 与fast寄存器分配不同，它是属于全局的寄存器分配，在函数的层面进行寄存器分配；
3. basic寄存器分配的实现，位于llvm/lib/CodeGen/RegAllocBasic.cpp之中，它的实现对应着RABasic类；
4. RABasic继承自MachineFunctionPass，也是一个pass，入口函数也是runOnMachineFunction()；
5. runOnMachineFunction()中会调用一个比较关键的allocatePhysRegs()，RABasic并未对allocatePhysRegs()进行实现，所以对allocatePhysRegs()的调用，会转到其父类RegAllocBase的allocatePhysRegs()；
6. 这里进行寄存器分配的时候，对所有的活跃区间的排序，并不是像Poletto有关线性扫描论文（Linear Scan Register Allocation, 1999）中的按照起点的顺序来的，而是根据spill weight从高到底排序的，spill weight是根据活跃区间的使用频率来计算的，使用频率越高，spill weight越高。

LLVM CodeGen中的greedy寄存器分配

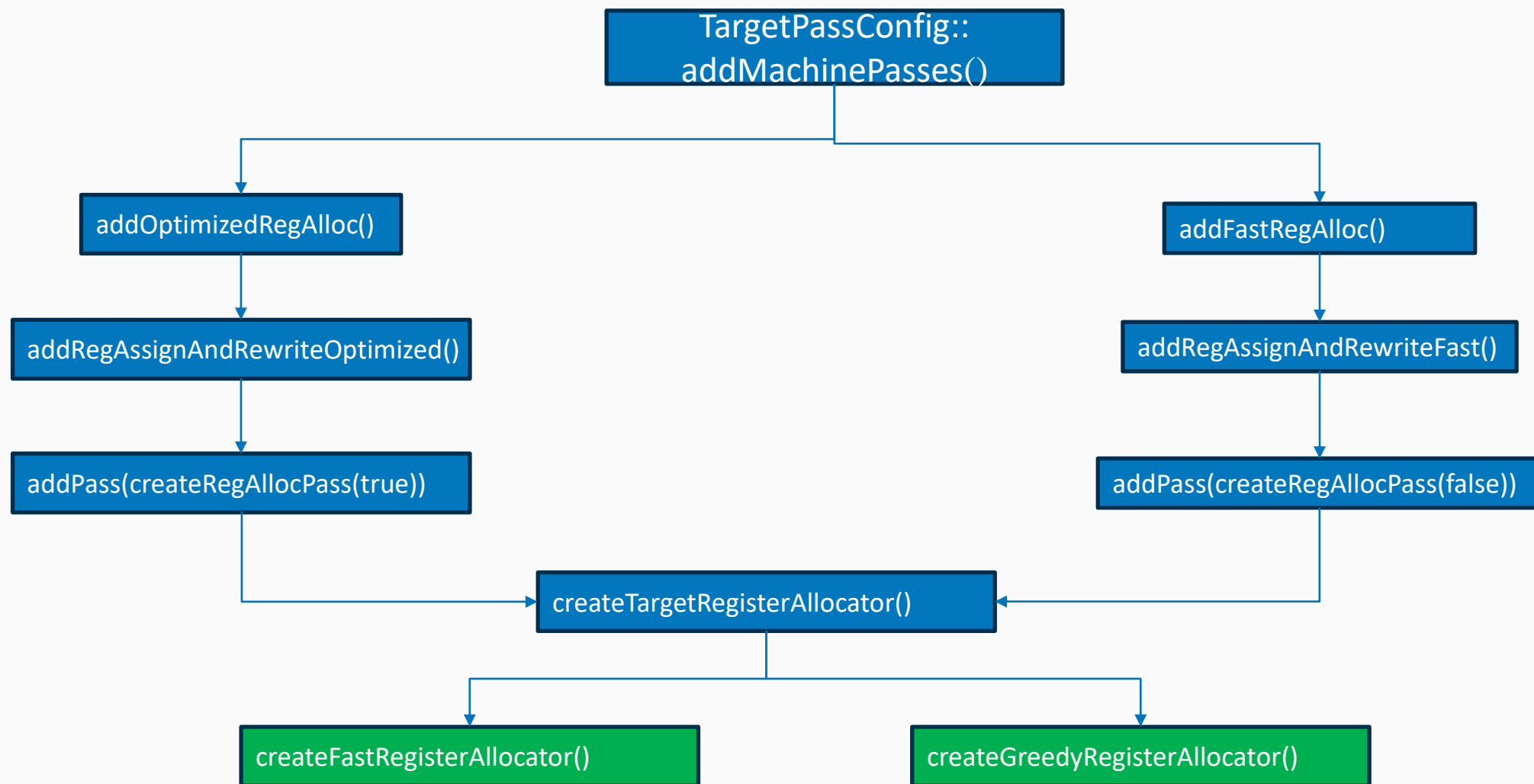
1. **greedy**寄存器分配是对**basic**寄存器分配的改进，所以它依然是属于线性扫描的寄存器算法。相比较于**basic**寄存器分配方式而言，它对寄存器的分配顺序做了调整，优先给比较大的活跃区间分配物理寄存器，而不是根据**spill weight**；
2. **greedy**寄存器分配的代码实现，主要位于llvm/lib/CodeGen/RegAllocGreedy.h和llvm/lib/CodeGen/RegAllocGreedy.cpp之中，主要对应着RAGreedy类；
3. RAGreedy类也继承于MachineFunctionPass，它也是个pass，入口函数为runOnMachineFunction()；
4. runOnMachineFunction()是它执行寄存器分配的核心动作所在，这个函数中的进行寄存器分配的核心动作，依然是调用了calculateSpillWeightsAndHints()和allocatePhysRegs()，这部分和上文介绍的**basic**寄存器分配是完全一样的；
5. **greedy**寄存器分配和**basic**寄存器分配的一个不同点就是分配的时候，使用的队列里活跃区间排列顺序不同，**greedy**寄存器分配使用的策略是按照从大到小排序，区别于**basic**寄存器分配的**spill weight**从高到低排序。

LLVM CodeGen中的PBQP寄存器分配

1. 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. —— 《A Detailed Analysis of the LLVM's Register Allocators》 P190-191
2. This allocator works by constructing a PBQP problem representing the register allocation problem under consideration, solving this using a PBQP solver, and mapping the solution back to a register assignment;
3. Hames, L. and Scholz, B. 2006. Nearly optimal register allocation with PBQP. In Proceedings of the 7th Joint Modular Languages Conference (JMLC'06). LNCS, vol. 4228. Springer, New York, NY, USA. 346-361.
4. Scholz, B., Eckstein, E. 2002. Register allocation for irregular architectures. In Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems (LCTES'02), ACM Press, New York, NY, USA, 139-148.
5. PBQP寄存器分配器的实现类是RegAllocPBQP，它是MachineFunctionPass的子类，它的实现位于llvm/lib/CodeGen/RegAllocPBQP.cpp。

LLVM CodeGen中寄存器分配的调用流程

llvm/lib/CodeGen/TargetPassConfig.cpp



目录

1. LLVM后端的架构与组成
2. LLVM CodeGen中的指令选择实现
3. LLVM CodeGen中的指令调度实现
4. LLVM CodeGen中的寄存器分配实现
5. 小结

LLVM CodeGen中的共性

1. LLVM CodeGen中包含了后端三大模块，指令选择、指令调度和寄存器分配的公共代码；
2. LLVM CodeGen中涉及后端三大模块中的具体实现时候，通常会将功能模块包装成MachineFunctionPass；
3. LLVM CodeGen中是通过LLVMTargetMachine调用TargetPassConfig来组装pass的过程，来实现代码生成的功能；
4. 具体的指令集架构，也会有自己的XXXTargetMachine和XXXPassConfig，分别继承于LLVMTargetMachine调用TargetPassConfig，来实现架构相关的最终代码生成流程；
5. 具体指令集架构的XXXTargetMachine和XXXPassConfig通常位于llvm/lib/Target/XXX目录之下；以RISC-V架构为例， RISCVTARGETMachine和RISCVPassConfig都位于llvm/lib/Target/RISCV/RISCVTargetMachine.cpp之中。

具体架构中TargetPassConfig的调用（以RISC-V为例）

llvm/lib/CodeGen/LLVMTargetMachine.cpp

llvm/lib/CodeGen/TargetPassConfig.cpp



llvm/lib/Target/RISCV/RISCVTargetMachine.cpp

llvm/lib/Target/RISCV/RISCVTargetMachine.cpp



Thanks!