浅谈LLVM上的RV指令编译

同构和异构模式下的RV指令及其编译系统的设计

张炅焱 / 2025-06-28

目录

RISC-V和LLVM

什么是RISC-V

RISC-V (reduced instruction set computer V) 采用**模块化**设计，由一组**基础模块**和**多种可选扩展模块**共同构成。基础模块定义了**核心指令及其编码、控制流机制、寄存器结构与位宽、存储器体系与寻址方式、整数运算等基本功能**，以及**必要的辅助设施**。仅凭这些基础模块，便可实现一台功能完备的通用计算机，并获得完整的软件生态支持——包括对主流编译器的全套适配。

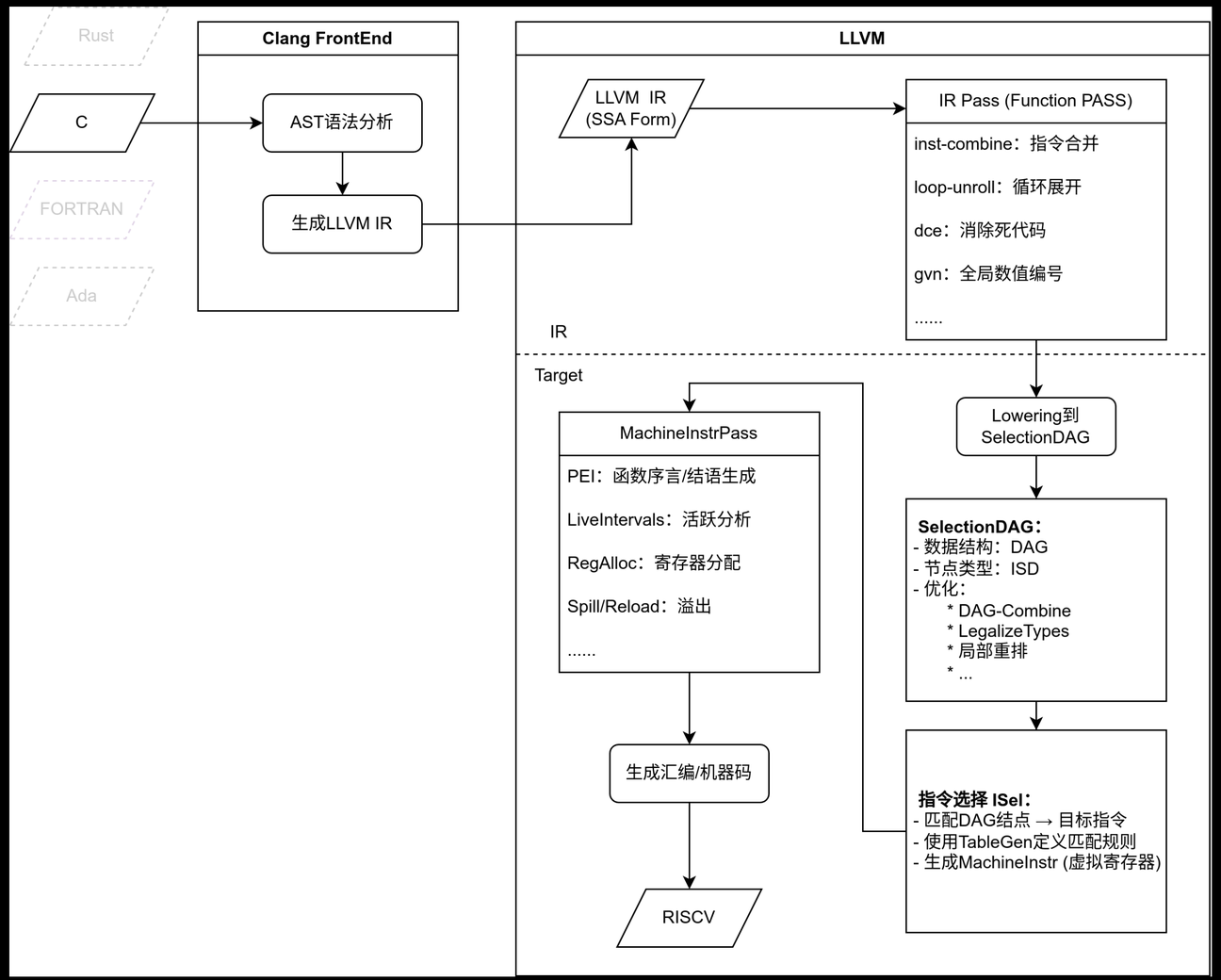
- 开源免费
- 模块化、可扩展
- 精简与一致性

Format	Bit																												
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Store	imm[11:5]							rs2					rs1					funt3			imm[4:0]					opcode						
Branch		imm[10:5]																			imm[4:1]										
Register	funct7																				rd										
Immediate	imm[11:0]																															
Upper Immediate	imm[31:12]																															
Jump		imm[10:1]											imm[19:12]																			

opcode (7 bits): Partially specifies one of the 6 types of instruction formats.
funct7 (7 bits) and funct3 (3 bits): These two fields extend the opcode field to specify the operation to be performed.
rs1 (5 bits) and rs2 (5 bits): Specify, by index, the first and second operand registers respectively (i.e., source registers).
rd (5 bits): Specifies, by index, the destination register to which the computation result will be directed.

RISC-V 32bit指令码的格式

LLVM架构简介



LLVM是一个庞大的软件系统，用以将各种前端语言转换为IR，再映射到目标平台上。

LLVM架构 (1/6) - 汇编支持

```
class RVInst<dag outs, dag ins, string opcodestr, string argstr,
             list<dag> pattern, InstFormat format>
    : RVInstCommon<outs, ins, opcodestr, argstr, pattern, format> {
  field bits<32> Inst;
  // SoftFail is a field the disassembler can use to provide a way for
  // instructions to not match without killing the whole decode process. It is
  // mainly used for ARM, but Tablegen expects this field to exist or it fails
  // to build the decode table.
  field bits<32> SoftFail = 0;
  let Size = 4;
}

// Pseudo instructions
class Pseudo<dag outs, dag ins, list<dag> pattern, string opcodestr = "", string argstr = "">
    : RVInst<outs, ins, opcodestr, argstr, pattern, InstFormatPseudo> {
  let isPseudo = 1;
  let isCodeGenOnly = 1;
}
```

LLVM对汇编（以及LLVM IR）的支持都通过TableGen这种脚本语言实现。对于RISCV的支持，主要是通过RVInst这个模板实现。RVInst实现了对32-bit汇编的支持，该模板可以被继承：

- outs：out参数（可以是立即数、寄存器）；
- ins：input参数（可以是立即数、寄存器）；
- opcodestr：指令名称，如"addi"
- argstr：参数字段，如"a, b, c"；
- pattern：可以通过这个参数将LLVM IR和汇编指令直接关联，较为简单，该关联也可以在Lowering过程中实现

  - Pseudo：伪指令，不具有自己的指令码，会在AsmParser阶段映射成其他指令

LLVM架构 (2/6) - Clang前端接口

```
// Zknh extension

TARGET_BUILTIN(__builtin_riscv_sha256sig0, "UiUi", "nc", "zknh")

TARGET_BUILTIN(__builtin_riscv_sha256sig1, "UiUi", "nc", "zknh")

TARGET_BUILTIN(__builtin_riscv_sha256sum0, "UiUi", "nc", "zknh")

TARGET_BUILTIN(__builtin_riscv_sha256sum1, "UiUi", "nc", "zknh")



TARGET_BUILTIN(__builtin_riscv_sha512sig0h, "UiUiUi", "nc", "zknh,32bit")

TARGET_BUILTIN(__builtin_riscv_sha512sig0l, "UiUiUi", "nc", "zknh,32bit")

TARGET_BUILTIN(__builtin_riscv_sha512sig1h, "UiUiUi", "nc", "zknh,32bit")

TARGET_BUILTIN(__builtin_riscv_sha512sig1l, "UiUiUi", "nc", "zknh,32bit")

TARGET_BUILTIN(__builtin_riscv_sha512sum0r, "UiUiUi", "nc", "zknh,32bit")

TARGET_BUILTIN(__builtin_riscv_sha512sum1r, "UiUiUi", "nc", "zknh,32bit")

TARGET_BUILTIN(__builtin_riscv_sha512sig0, "UWiUWi", "nc", "zknh,64bit")

TARGET_BUILTIN(__builtin_riscv_sha512sig1, "UWiUWi", "nc", "zknh,64bit")

TARGET_BUILTIN(__builtin_riscv_sha512sum0, "UWiUWi", "nc", "zknh,64bit")

TARGET_BUILTIN(__builtin_riscv_sha512sum1, "UWiUWi", "nc", "zknh,64bit")
```

在Clang前端定义一组接口，用户便可以更加方便地使用C语言接口。该定义在Clang模块的Basic的Builtinxxx.def中进行。

关于TARGET_BUILTIN：

- 第一个参数是**接口名**；
- 第二个参数是**参数列表**，不同的字母或者字母组合代表不同的参数类型，第一个字母代表返回值类型，如“v”代表“void”；
- 第三个参数代表**接口的属性**；
- 第四个参数代表该接口**可用的架构名称**。

LLVM架构 (3/6) - LLVM IR

```
class SystemZBinaryConvIntCC<LLVMType result, LLVMType arg>
  : Intrinsic<[result, llvm_i32_ty], [arg, llvm_i32_ty],
              [IntrNoMem, ImmArg<ArgIndex<1>>]>;

class SystemZBinaryCC<LLVMType type>
  : SystemZBinaryConvCC<type, type>;

class SystemZTernary<string name, LLVMType type>
  : SystemZTernaryConv<name, type, type>;

class SystemZTernaryInt<string name, LLVMType type>
  : ClangBuiltin<"__builtin_s390_" # name>,
    Intrinsic<[type], [type, type, llvm_i32_ty], [IntrNoMem, ImmArg<ArgIndex<2>>]>;

class SystemZTernaryIntCC<LLVMType type>
  : Intrinsic<[type, llvm_i32_ty], [type, type, llvm_i32_ty],
              [IntrNoMem, ImmArg<ArgIndex<2>>]>;

class SystemZQuaternaryInt<string name, LLVMType type>
  : ClangBuiltin<"__builtin_s390_" # name>,
    Intrinsic<[type], [type, type, type, llvm_i32_ty],
    [IntrNoMem, ImmArg<ArgIndex<3>>]>;
```

LLVM IR有“承上启下”的作用。在定义LLVM IR的时候，一般都由“int_”开头，表明是IR Intrinsics：

- IR Intrinsics继承自Intrinsic类，其第一个参数是返回值类型，第二个参数是输入参数类型，第三个参数是一些属性，如是否读写内存、是否有参数是立即数等；
- 通过ClangBuiltin，可以将Clang接口和LLVM IR对接。

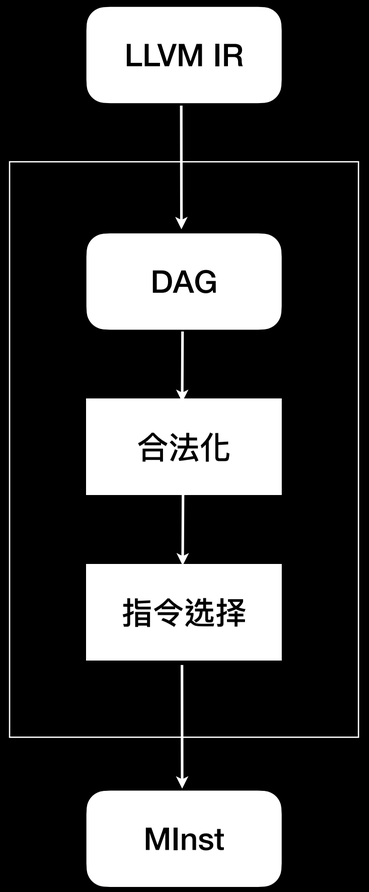
LLVM架构 (4/6) - 类型支持

1. 后端新增物理寄存器类型
   - 在 <Target>RegisterInfo.td中用TableGen**定义新寄存器**（如 RZ）
   - 定义对应的RegisterClass（如 RZRegClass）
   - 在指令模板里用%RZRegClass约束操作数
   - 更新CallingConv/ABI分配规则
2. LLVM IR 层增加新类型
   - 在TypeID枚举里**添加新条目**（如 Integer128TyID）
   - 在TypeNodes.td声明新Type（如i128）
   - 在Type.cpp 提供工厂方法 getInt128Ty()
   - 在DataLayout/TargetLowering中指定大小、对齐
3. Clang 前端暴露给用户
   - AST：在BuiltinType中注册新Builtin（如 Reg128）
   - Parser：识别关键字（如 __reg128），生成对应 QualType
   - CodeGen：将 **AST 类型映射到 LLVM IR 的 i128**

LLVM架构 (5/6) - Lowering

LLVM的Lowering过程是将高级的、与目标无关的LLVM IR逐步转换为目标机器相关的低级代码（如汇编或机器码）的核心步骤

- IR 优化（IR-Level Optimizations）：在 LLVM IR 上进行**通用变换**（常量合并、死代码消除、循环优化等），为后续指令选择打好基础。
- SelectionDAG 构建（Build SelectionDAG）：将已优化的 LLVM IR 逐条转换为**一个目标无关的有向无环图（DAG）**，节点代表操作，边代表数据依赖。
- DAG 合法化（Legalization）：将不被目标机器直接支持的操作或数据类型拆分 / 扩展为**等价的基础操作**（例如，把 64 位除法拆成多条 32 位指令）。
- 指令选择（Instruction Selection）：利用 TableGen 定义的模式匹配规则，将合法化后的 DAG 节点**映射到目标机器指令**，生成 MachineInstr 表示。
- 调度（Instruction Scheduling）：根据目标的流水线结构与资源限制，对 MachineInstr 进行排序，以提高执行并行度、减少气泡（stall）。
- 寄存器分配（Register Allocation）：为每个虚拟寄存器分配物理寄存器，插入必要的 spill / reload 指令，解决寄存器不足问题。
- 后端处理与代码生成（Post-RA Passes & Emission）：包括插入函数序言 / 尾语、块布局优化、最终二进制编码，输出目标机器码或汇编。



LLVM架构 (6/6) - PASS

Pass 是 LLVM 的核心编译单元，代表对代码的**一次处理过程**（优化/分析/转换）。

- 按功能

1. Analysis Pass：收集信息（如 DominatorTree 控制流分析）
2. Optimization Pass：转换代码（如 InstCombine 指令化简）
3. Tool Pass：调试/输出（如 PrintModulePass 打印 IR）

- 按作用范围

1. Module Pass：跨函数处理（如全局优化 GlobalOpt）
2. **Function Pass：**IR处理（如 ADCE 死代码消除）
3. **Machine Pass：**Machine指令处理（如寄存器分配）

- 关键特性

1. Pipeline 组合：Pass 序列构成优化流程（-O1/-O2）
2. 依赖管理：自动调度分析 Pass 为优化 Pass 提供数据
3. 扩展性：开发者可自定义 Pass 插入编译流程

RISC-V的编译支持

- 前端优化：

输入计算图，关注**计算图整体拓扑结构**，而不关心算子的具体实现。在 AI 编译器的前端优化，对算子节点进行融合、消除、化简等操作，使计算图的计算和存储开销最小。

- 后端优化：

关注**算子节点的内部具体实现**，针对具体实现使得性能达到最优。重点关心节点的输入、输出、内存循环方式和计算的逻辑。

针对后端的优化 (1/7)

RISC-V 后端负责把前端或中端的抽象指令（IR、SelectionDAG、Machine IR）最终转成机器代码。理解其原理，**并掌握可调节的点**，能够在不改动前端 IR 的情况下，显著提升性能与代码密度。

针对后端的优化 (2/7) - 指令选择（Instruction Selection）

原理：将 LLVM 的中间表示（IR）或指令选择图（DAG）中的操作，通过**模式匹配（Pattern Matching）**映射到目标架构的指令。使用**成本模型（Cost Model）**评估各候选模式的执行代价，选择最优实现方案。

优化点：

- **TableGen Patterns：结合匹配规则提升指令复用**

  利用 TableGen 自定义 Pat<Op, Instr> 规则，实现 IR 操作到目标指令的组合映射。

  - 例如：针对常见算子（如加法、乘法）构建**复合指令模式**（如 Load+Op），提升执行效率与代码密度。
  - 优化重点是围绕**热点路径**（hotpath）构建更高效的匹配模板。

- **Cost Model 调优：指导指令选择和调度优化**

  在 .td 文件中设置目标指令的延迟（Latency）与吞吐（Throughput）参数，用于指导 LLVM 的选择与调度策略：

  - 影响指令选择器（ISel）的**优先级判断**；
  - 在后续调度阶段（如 MachineScheduler）中影响指令发射顺序，进而影响指令级并行度。

- **GlobalISel：轻量级替代SelectionDAG的新路径**

  使用 GlobalISel 直接从 IR 生成 Machine IR（MIR），**跳过 SelectionDAG 阶段**：

  - 具有更强的架构可扩展性和插件化能力；
  - 可插入自定义的 Lowering 流程，支持复杂数据类型、向量化或特定平台优化；
  - 支持在 early phase 做 cost-sensitive 选择，适合定制芯片或 RISC-V 这种指令稀疏 ISA。

针对后端的优化 (3/7) - 寄存器分配（Register Allocation）

原理：LLVM 通过分析每条指令的**活跃区间（Live Interval）**，构建变量之间的**干涉图（Interference Graph）**，然后使用特定算法为虚拟寄存器分配物理寄存器。当寄存器压力过大时，将部分变量Spill 到内存（栈）。

优化点：

- **Live-Range 划分：缩小寄存器使用范围，减少溢出**

  将生命周期较长的变量按基本块或指令切分为多个子区间（Split Range）：

  - 减少活跃区间的重叠，**提高寄存器复用**；
  - 降低跨基本块 Spill 的频率。

- **寄存器优先级配置：优化物理寄存器分配策略**

  - 配置Caller/Callee保存约定，指导寄存器在函数调用前后的使用策略；
  - 设置分配优先级顺序，例如**将高频使用寄存器优先分配给热点变量**，以减少 Spill 成本。

- **Spill 优化: 降低访存开销**

  - 对于热点变量，可以配合调度器提前插入**预取（Preload）**指令或推迟 Spill 时机；
  - 合并相邻的 Spill/Reload 指令，**减少访存指令数量与带宽占用**；
  - 可通过 SpillPlacement 与 LiveRangeEdit 插件自定义策略。

- **替代策略: 应对高寄存器压力场景**

  - LLVM 提供多种 RegAlloc 实现，可根据不同场景替换默认策略。

针对后端的优化 (4/7) - 指令调度（Instruction Scheduling）

原理：LLVM 的指令调度器基于目标 CPU 的**发射宽度（IssueWidth）**、**执行单元拓扑结构**及**各指令的延迟（Latency）**信息，对指令进行重排序。目的是隐藏流水线停顿（Stalls）、提升并行度（ILP），使**硬件资源利用最大化**。

优化点：

- **机器模型：建模执行资源与延迟信息**

  在目标描述文件（.td）中，通过 SchedModel 定义：

  - 每条指令的 **延迟（Latency）** 与 **所占用资源（ResourceUsage）**，如整数 ALU、分支单元、加载单元等；
  - 设置 CPU 的 IssueWidth、流水线深度与资源拓扑，用于约束调度窗口；→ 这些定义直接影响调度器在**选择指令时的并行性评估**。

- **调度算法：控制调度顺序和范围**

  LLVM 支持多种指令调度策略，包括：

  - List Scheduling：按就绪队列和优先级调度，常用于后端（Post-RA）调度；
  - Region Scheduling：在单个基本块或 Region 内全局调度，适合前端（Pre-RA）调度；
  - 可扩展或替换为**自定义调度器**（通过继承 MachineScheduler 插件实现）。

- **热点路径识别：结合 PGO 做调度加权**

  利用 Profile-Guided Optimization（PGO）采样数据，标记**热点基本块（Hot Blocks）或函数**，优先优化这些路径上的调度顺序：

  - 提升关键路径性能；
  - 降低 cache miss 与分支预测失败概率。

- **跨基本块调度（Superblock）：打破基本块边界提升并行度**

  - 在控制流图（CFG）结构较简单的场景下（如直线结构或单分支路径），LLVM 可将多个基本块合并为一个 Superblock 或 Trace。通常借助MachineTraceMetrics。

针对后端的优化 (5/7) - 分支放宽与跳转优化

原理：RISC-V 的分支指令受编码限制，跳转范围有限：

- 短跳（如 BEQ, BNE 等）只能跳转 **±4KB 范围**（12-bit 偏移）；
- 长跳需要使用 **AUIPC + JALR** 组合构造。

编译器在分支距离超限时需进行Branch Relaxation：自动插入额外跳转或替换为远跳模式，以确保程序正确跳转。

优化点：

- **Branch Relaxation：智能选择短跳与远跳**

  LLVM 的 BranchRelaxation Pass 会判断跳转目标距离，并自动选择最合适的跳转方式。优化目标：

  - **优先使用短跳**（12-bit offset）以避免代码体积膨胀；
  - 利用 Profile-Guided Information，将热点路径靠近放置，冷分支置于远处，**减少长跳的必要性**；
  - 对短跳与远跳的混合使用，需避免跳转链影响性能。

- **跳转分区：布局驱动的距离优化**

  编译阶段将冷分支代码（如异常处理、assert）**移出热点路径区域**，减少常见跳转距离：

  - 提高热点路径局部性（ICache 命中率更高）；
  - 降低 AUIPC+JAL 的使用频率，可配合 CodeLayoutPass 或 PGO 分区。

- **分支预测 Hint：提高预测准确率**

  部分 LLVM 后端支持插入预测提示前缀（如 beq.t, beq.f）：

  - .t 表示“likely taken”，.f 表示“likely not taken”；
  - 这些前缀可引导硬件预测器提前判断跳转方向，降低 mispredict penalty
  - 通常依赖 **PGO 或静态分析结果**进行标记。

针对后端的优化 (6/7) - 栈帧布局与栈槽重用

原理：LLVM 中的 FrameLowering 组件负责生成函数的序言（Prologue）与尾声（Epilogue），包括设置栈指针、保存/恢复寄存器、为局部变量和 Spill 分配栈空间。其目标是保证函数调用符合目标平台 ABI 规范，并在保证 correctness 的前提下，**尽可能减少开销**。

优化点：

- **小栈帧压缩（Small Frame Optimization）：生成更紧凑的 prologue/epilogue**

  当函数栈帧大小较小（如 ≤128 Bytes），可使用 RISC-V 的 **压缩指令（Compressed Instructions）**优化栈指针调整：

  - 使用 c.addi4spn, c.addi16sp 替代标准的 addi, li 等；
  - 减少代码体积，提升 I-Cache 命中率；LLVM 会自动判断帧大小，在 emitPrologue() 中选择是否使用压缩形式。

- **栈槽重用（Stack Slot Coloring）：减少栈内存占用，提升数据局部性**

  LLVM 在栈分配阶段会进行 Stack Slot Coloring 优化：

  - **重用**生命周期互斥（non-overlapping）的局部变量或 Spill 空间；
  - 减少整体栈帧大小，提升 L1 Data Cache 利用效率；
  - 特别适用于含多个中间变量或短生命周期变量的函数。

- **Callee-Saved 精简：仅保存必要寄存器，减少序言/尾声指令数**

  按目标平台 ABI 约定，函数需在调用中保存 被调用者保存寄存器（Callee-Saved Registers）。优化策略包括：

  - 仅保存在函数中实际被使用的寄存器（由 PrologueEpilogueInserter 分析生成）；
  - **减少序言中 sw, sd 指令的数量，以及尾声的恢复指令**，提升执行效率；
  - 可结合 MachineRegisterInfo 分析寄存器活跃性。

常见优化策略 - 中前端 (3/5) - 跨函数/模块IPO

- 概念与流程
  - IPO：Interprocedural optimization，过程间优化，分析整个程序
  - LTO (Link Time Optimization)：在链接阶段合并各模块 IR
  - ThinLTO：轻量版 LTO，支持并行增量优化
  - 流程：编译 → 生成模块 IR (.bc/.o) → LTO 合并与全局分析 → 目标代码生成

- 核心技术
  - 跨模块内联 (Cross-Module Inlining)：消除函数**调用开销**
  - 全程序常量传播 (Global Constant Propagation)：在模块间**展开常量**
  - 全局死代码消除 (Global DCE) & GVN：跨边界**删除冗余**
  - 堆栈/逃逸分析 (Stack & Escape Analysis)：优化**内存分配与访问**
  - 全局别名分析 (Global AA)：提升**指令合并和循环优化**效果

- 可改进方向
  - 自定义 IPO pass：针对**热点函数或数据结构**设计专用分析/变换
  - Profile-Guided IPO：用**运行时数据**优化跨模块内联与循环转换阈值
  - ThinLTO 并行策略：调优机器级**并行度**、优化单元划分
  - 扩展全局 AA：加入**基于类型或场景的别名推断**，提升跨模块优化精度

针对后端的优化 (7/7) - RVC 指令压缩

原理：RISC-V 支持 **16-bit 宽度的压缩指令集（RVC）**，可将常用指令（如 addi, lw, jalr 等）映射为更短的编码形式。通过将标准 32-bit 指令替换为等效的 16-bit 压缩版本，可显著**减小代码体积**，并**提升指令缓存（I-Cache）命中率与带宽利用率**，对嵌入式或高性能场景均有积极效果。

优化点：

- **压缩模式定义（Compress Pattern）：自动识别压缩机会**

  在 TableGen 中通过设置 CompressibilityPredicate 或 CompressPat，定义哪些 MachineInstr 可映射为对应的 CompressedMCInst：

  - 例如：addi sp, sp, -16 可压缩为 c.addi16sp；
  - LLVM 后端在 MCCodeEmitter 阶段自动完成替换，无需人工干预；
  - 可在目标描述中扩展**自定义压缩规则**（尤其适用于自研指令扩展）。

- **调度优化：优先布局可压缩指令，便于后期打包**

  在指令调度阶段（如 MachineScheduler），可通过**调度权重**微调：

  - 将压缩指令或其组合倾向性调度到相邻位置，增加连续压缩打包成功率；
  - 特别适用于 c.li, c.mv, c.addi 等易出现于开头/结尾的模式；
  - 有助于提升 RVC 压缩覆盖率，降低 padding 与 nop 的使用。

-

常见优化策略 - 中前端 (1/5) - mem2reg

mem2reg：局部变量提升 (Promote Memory to Register)

将 alloca 产生的栈变量提升为 SSA 形式的寄存器定义，**消除冗余 load/store**，为后续优化奠定基础。

主要收益

- 降低内存访问成本

  减少栈读写，显著缩短访存延迟，节省带宽。

- 简化数据流，激活更多优化

  数据依赖**显式化**，便于 DCE、GVN、LICM 等基于 SSA 的优化生效。

- 缩短依赖链，提高 ILP

  寄存器数据就绪更快，为**乱序执行与指令级并行腾出空间**。

可改进方向 & 实践技巧

- 固定大小缓冲区 (Fixed-size Buffer) 处理

  对小型数组 / 结构体做切片或聚合，进一步消除残余 alloca。

- 细粒度启发式：识别高频变量

  借助 PGO / 采样，在热点路径优先执行 mem2reg，避免冷代码无谓膨胀。

- 协同优化

  与 Inlining 配合：函数内联后**暴露更多栈变量**，可一次性提升。

  与 Loop-Unroll 配合：展开循环后寄存器**重用度提高**，减少寄存器压力。

常见优化策略 - 中前端 (2/5) - 冗余消除和循环优化

冗余消除 (Redundancy Elimination)

- DCE – Dead Code Elimination：删除**无副作用且结果未被使用**的指令
- GVN – Global Value Numbering：识别并**合并**等价子表达式，避免重复计算

改进切入点

- Profile-Guided DCE：利用 PGO 先**清理冷代码**，减少优化耗时
- Target-Aware GVN：针对 RISC-V 扩展（如 Zba/Zbb）设计合并模式，生成**更短指令序列**

循环优化 (Loop Optimizations)

- LICM – Loop-Invariant Code Motion：不变计算**外提**，降低迭代开销
- Unroll / Rotate：展开提高 ILP，旋转改善分支预测与向量化机会

改进切入点

- Cost-Based Unroll：结合指令数、寄存器压力与 RVV 向量长度，动态调整展开度
- Smarter Vector Heuristics：针对 RVV 可变长度，增强自动向量化触发条件

InstCombine

- 基于模式匹配的指令合并，早期反复运行，持续压缩 IR
- 常量折叠、**代数恒等变换**，暴露更多 CSE 机会

改进切入点

- RV-Specific Patterns：如位操作指令 (Zbs) 或乘加 (Zmmul) 的专用简化规则
- 深层表达式折叠：一次性规约连锁计算，**减少遍历轮次**

常见优化策略 - 中前端 (4/5) - PGO / FDO

- 定义 & 流程

  PGO（Profile-Guided Optimization）与 FDO（Feedback-Directed Optimization）本质上是同一类技术：

  1. LLVM/Clang、MSVC 等常用 PGO 一词；
  2. GCC 习惯称 FDO（也会说 PGI、PGO，含义一致）。

  – 基于真实运行数据指导编译优化

  – 步骤：

  1. 编译插桩（-fprofile-generate / -fprofile-instr-generate）
  2. 运行收集 Profile（硬件事件或插桩数据）
  3. 重编译（-fprofile-use / -fprofile-instr-use），应用反馈

- 核心技术
  1. 基于**热度**的函数内联与阈值调优
  2. **分支预测**强化 & **代码布局**优化 (Cache-friendly Layout)
  3. 热/冷区分（Hot/Cold Splitting） & 冷代码剥离
  4. 循环展开、对齐 & 倾斜执行决策

- 常见工具链
  - GCC：-fprofile-generate / -fprofile-use
  - Clang：-fprofile-instr-generate / -fprofile-instr-use
  - Sample-based PGO：perf + AutoFDO

- 可改进方向
  - 自动化采样 PGO（AutoFDO）**集成与评测**
  - 动态/增量 PGO 流水线 (Continuous FDO)
  - **PGO + LTO 联合优化策略**
  - 基于硬件性能计数器的**精准热点分析**

常见优化策略 - 中前端 (5/5) - Auto-Vectorization Hint

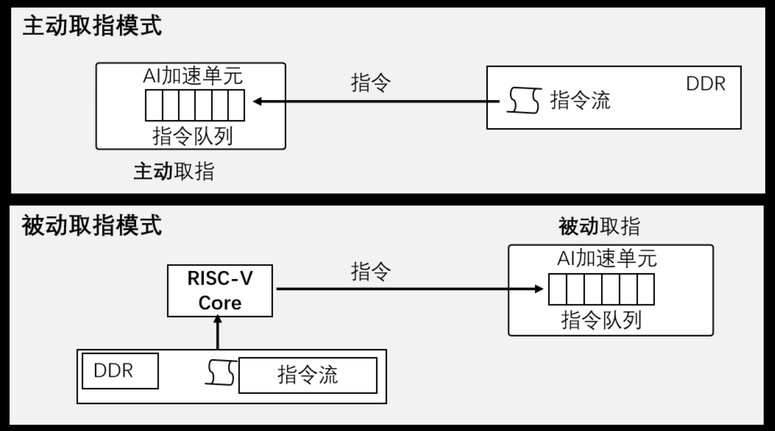
- 前提条件
  - 硬件：启用向量扩展（RVV），编译时 `-march=rv64gcv` 或 `-mattr=+v`
  - 编译器：LLVM/Clang or GCC 支持 Loop Vectorizer
- Hint 机制
  - 编译器依赖分析后，自动选取 RVV 指令（`vsetvl/vsetvli` + `vadd.vv`, `vle.v`, …）
  - 用户可在源代码中插入编译指示，强化矢量化决策
- 常用 Pragma/Attribute
  - Clang: #pragma clang loop vectorize(enable) interleave(enable) unroll_count(4)
  - GCC: #pragma GCC ivdep // 忽略可跨迭代依赖 #pragma GCC unroll 4 // 指定展开因子
  - 通用属性: __attribute__((optimize("tree-vectorize")))
- 性能调优要点
  - 动态 VL（Vector Length） 调整：合理选取 vsetvl 参数
  - 数据对齐 & 预取：保证连续访问、减少跨页/缓存冲突
  - 热路径重排：结合 PGO 划分 Hot/Cold，集中矢量化热点循环
- 延伸研究
  - 自动探索最佳展开、Interleave 与 VL 组合的脚本/框架
  - 基于运行时 Profile 动态切换矢量化策略
  - 比较 Intrinsic vs. Auto-Vectorized 在 RVV 上的性能差异

编译优化思路

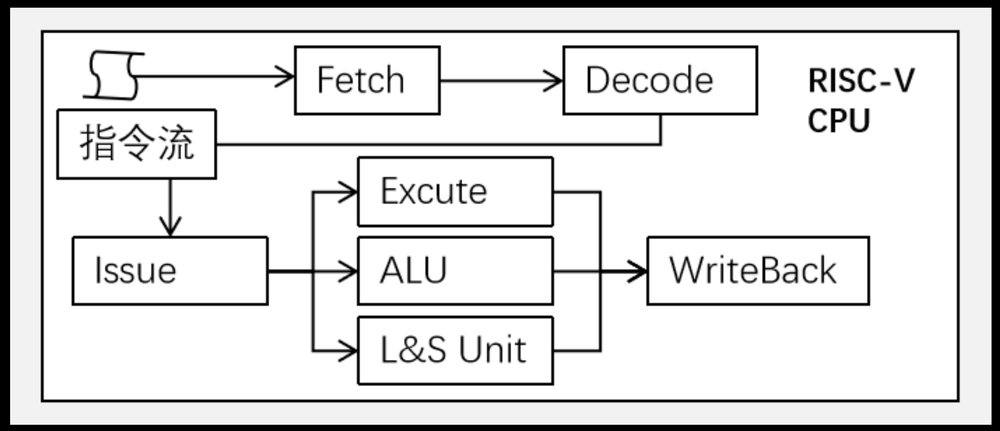
优化思路	优化方法
目标驱动	明确优化维度：性能（吞吐、延迟）、代码体积、能耗或启动时间
	制定可量化指标 & 基准测试用例
分析与反馈	静态分析：利用 Alias/Dependence & Loop 信息
	动态 Profile：PGO/FDO 数据指导热点、分支与内存访问模式
模块化 Pass 设计	分层拆分：前端 IR 清理 → 中端 Loop/SLP/InstCombine → 后端 目标相关
	插件化：按需启用 / 组合，支持 LTO / ThinLTO
参数化与自动调优	可调度参数：内联阈值、循环展开因子、向量化策略
	自动化脚本：批量测试不同组合，收集性能曲线
RISC-V 特性适配	利用 RVV、PMA/Cache Hint、Atomics 优化指令生成
	调整调度器与寄存器分配，降低延迟和冲突
验证与迭代	IR & 代码回归测试，确保正确性
	持续集成：集成 AutoFDO / A/B 测试，定期更新 Profile
文档与协作	清晰 Pass 依赖图 & 接口定义
	与团队协作：Code Review、设计讨论、共享基准

学习资源推荐

学习资源推荐 - 异构和同构的RISC-V架构



异构RISC-V



同构RISC-V

	异构RISC-V	同构RISC-V
目标	最小搬运成本 → 最大加速收益	拉满单核 + 横向扩核
调优要点	- Kernel ≥ 10× 复制延迟；双缓冲覆盖 DMA - Fuse 小算子，减少 launch 次数 - 零拷贝 / Unified Memory；必需时 dma_start() + Cache flush - CPU 侧前后处理再用 RVV SIMD	- RVV 自动矢量化 + 循环展开 - PGO + ThinLTO 把热点贴进 I-Cache - 数据对齐、预取、SoA 布局
总结	“先切分算子再搬运” → Offload + Overlap + 大粒度 Kernel	“编译器榨干CPU能力” → RVV + PGO + 多核并行

学习资源推荐

- LLM：可以随时问任何问题；
- LLVM官网：可以查阅权威文档；
- LLVM论坛：查阅各种常见问题并提问；
- 视频课程：Bilibili或YouTube栏目。

谢谢观看