

从 .II 到 ARM 汇编：老太太也能懂的流程

主线 aarch64_target.cpp：

寄存器

整数通用寄存器分工 (AArch64 常规 ABI, 结合你后端的用法) :

- x0-x7: 函数参数与返回值 (整数/指针) ; 返回值主用 x0。
- x8: 备用/间接返回地址指针 (按 SysV 习惯) 。
- x9-x15: caller-saved 临时。你在栈降低里用 x15 做大偏移基址, x9 作为保存/恢复 callee-saved 时的偏移临时。
- x16-x17: IP0/IP1, 调用序列内部可作跳板, 一般当作 caller-saved。
- x18: 平台寄存器 (在部分 ABI 保留), 通常不动。
- x19-x28: callee-saved, 被调用者需成对保存/恢复 (你在 StackLowering 里按需分配保存槽再 STP/LDP) 。
- x29: 帧指针 (fp) 。
- x30: 链接寄存器 (lr, 返回地址) 。
- x31: 编码上是零寄存器/别名 sp, 真实栈指针单独用 □sp。
- 栈帧/序言尾声约定 (你代码里) :
 - 序言: STP x29,x30,[sp,#-16]! → □mov x29, sp → □sub sp, sp, #frameSize (大立即数则 x9 装数再 SUB) → 保存需要的 callee-saved。
 - 尾声: 逆序恢复 callee-saved → □add sp, sp, #frameSize (或 x9+ADD) → LDP x29,x30,[sp],#16 → □ret。
- 特殊约定 (你实现里的临时选择) :
 - x15: FI 展开、大偏移寻址的 scratch (StackLowering 用于 sp+大偏移) 。
 - x9: 保存/恢复 callee-saved 时, 偏移放不进立即数时的 scratch; 也用于序言/尾声调栈大立即数时装数再 SUB/ADD。
- 浮点/向量寄存器 (简述) :
 - v0-v7: 浮点参数/返回值 (s0/d0 作标量视图) 。
 - v8-v15: caller-saved (常规) 。
 - v16-v31: callee-saved (常规, 需按 16 字节对齐保存恢复) 。

GEP

下面按“生成步骤”拆解 AArch64 GEP 降低过程, 对应 aarch64_ir_isel.cpp:1168-1379:

1. 取基址信息
 - 用 □resolveAddr(addr_map_, inst.basePtr) 判定基址类型: 帧槽(FrameIndex+offset) / 全局符号 / 已有寄存器地址 / 兜底。
2. 准备维度和元素大小
 - □elemSize = typeSize(inst.dt)。
 - 记录维度 dims 与索引列表 idxs, 用于算步长。
3. 计算步长 (每个索引的 stride)
 - □calcStride(i): 行主序, 后续维度相乘再乘元素大小; 若维度信息不足有兜底逻辑。
4. 拆分偏移: 常量偏移 constOffset
 - 遍历索引: 立即数索引 □imm * stride 累加进 □constOffset (int) 。

5. 拆分偏移：动态偏移 dynOffset (寄存器)

- 对寄存器索引：
 - a) \square SXTW 把 i32 扩成 i64。
 - b) stride $\neq 1$ 时：装 stride 立即数，MUL idx64, stride 得 scaled。
 - c) 多个动态索引用 \square ADD 累加到同一个 \square dynOffset；存在则 \square hasDyn=true。

6. 为结果分配寄存器

- \square resReg = ensureVReg(..., BE::PTR) 用于承载最终地址。

7. 辅助加法 emitBaseAdd(base, off, fiop)

- 如果 off 能编码在 ADD 立即数里：ADD resReg, base, #off (可带 fiop 标记 FrameIndex)。
- 否则：装 off 到寄存器，再 \square ADD resReg, base, offReg。

8. 按基址类型生成地址

- Frame: emitBaseAdd(sp, constOffset, fiop=FrameIndex); 若有 dyn，再 \square ADD resReg, resReg, dynOffset。记录：无 dyn 时仍标记 Frame (frameIndex+offset)，有 dyn 时改标记 Reg。
- Global: LA baseReg, symbol $\rightarrow \square$ emitBaseAdd(baseReg, constOffset+baseInfo.offset); 若有 dyn 再加。记录为 Reg。
- Reg: emitBaseAdd(baseInfo.reg, baseInfo.offset+constOffset); 若有 dyn 再加。记录为 Reg。
- 兜底：若 basePtr 是寄存器按 Reg 路走，否则用 sp 作基址，同样处理 dyn，记录为 Reg。

9. 写回地址信息

- \square addr_map_[resId] = AddrInfo{kind=Frame or Reg, reg/resolved frameIndex+offset}。
- 后续 load/store 会据此决定能否用 \square [sp,#imm] 直接寻址，或用寄存器基址/加法结果。

10. 收益

- 常量偏移能编码时，后面直接用 sp+imm 或 reg+imm，少插入额外 ADD。
- 有动态索引时，提前把偏移算好放寄存器，load/store 直接用寄存器基址。

使用到的寄存器相关

ISel 这几行在做啥（大白话版）

- 这一步把中间的 LLVM IR“翻译”成 AArch64 后端能理解的指令形式（目标相关的 MIR）。相当于“把通用伪代码换成 ARMv8 能用的指令拼块”。
- 创建 IRIsel 对象时，传进去当前的 IR 模块、后端模块容器和目标信息，然后调用 run() 完成这次翻译。调用链（从这里往下）

1. runPipeline 里构造 BE::AArch64::IRIsel isel(ir, backend, this);
2. 调 isel.run()
3. 进入 IRIsel::run() (在 isel 目录里定义，负责遍历 IR)
4. IRIsel::run() 内部：
 - 遍历 LLVM IR 的模块/函数/基本块/指令，把“通用 IR 指令”翻译成“目标相关的中间表示 (MIR) + 虚拟寄存器”。
 - 具体做的准备包括：
 - 建立函数参数与目标寄存器/栈槽的对应关系（入参搬到虚拟寄存器，必要时从 x0-x7 / v0-v7 或栈取出）。

- 为 IR 里的每个 SSA 值分配后端虚拟寄存器，记账哪些值存在哪个 vreg。
- 根据不同 IR 指令生成对应的 AArch64 目标指令（算术、比较、load/store、call、ret、GEP 等），必要时物化立即数/地址。
- 处理特殊节点如 phi：在合适位置插入 move，消除 SSA phi。
- 记录全局变量、帧对象、地址偏移等信息，后面栈处理/寄存器分配要用。

简单理解：ISel 就是把“抽象的 LLVM IR”先翻成“带虚拟寄存器的 ARMv8 指令蓝图”，后续 RA/Frame/Stack/汇编才能接着干。

调用链条？

aarch64_target.cpp里面的run函数 -> isel_base.h，然后从.h再通过函数runImpl() -> aarch64_ir_isel.cpp函数runImpl() -> 调用 apply()函数。

apply函数才是核心，那究竟在干什么呢？

apply函数会遍历llvm内的所有语句，然后根据语句类型选择不同的visit函数进行处理，例如处理类似 $+$ $*$ 这种运算型的ir，会进算术分支的visit函数进行操作，类似br这样的分支语句会进入分支语句的visit函数内进行各自的处理（visit函数都在aarch64_ir_isel.cpp内）

例如对于算术类指令：

把一条 IR 的算术/逻辑/移位指令翻成对应的 AArch64 指令，并把结果放到目标虚拟寄存器里。

- 核心步骤（按执行顺序讲大白话）

1. 确定结果寄存器：把 IR 里的结果 SSA 名映射到一个后端虚拟寄存器 \square dst。
 2. 把操作数“实体化”为寄存器或立即数：
 - materializeAsReg：无论是寄存器、int 立即数、float 立即数，都先转成寄存器（立即数会先用 emitLoadImm/materializeF32 加载进寄存器）。
 - materializeAsOp：如果指令允许立即数且立即数能放进指令编码，就直接做成 ImmeOperand；否则也会先装进寄存器。
 3. 选指令、判断能否用立即数：
 - 根据 IR 的 opcode 选 AArch64 的 op (ADD/SUB/MUL/SDIV/FADD/.../移位/位运算)。
 - 标记该 op 是否支持立即数（比如 ADD/SUB/移位支持，MUL/SDIV 不支持）。
 - 对右操作数如果是 int 立即数，检查是否编码范围可直接放进指令（12 位无符号、或移位位数范围）。
 4. 特殊处理取模 %：没有直接的 mod 指令，展开成 SDIV 得商 \rightarrow MUL 还原乘积 \rightarrow SUB 得余数。
 5. 生成最终指令：
 - 左操作数必然是寄存器 (lhs)。
 - 右操作数如果能用立即数且被允许，则用立即数，否则用寄存器 (rhs)。
 - 生成三元指令 \square op dst, lhs, rhs，推入当前基本块指令列表。
- 小白速记：
 - 先把 IR 里的值都换成“后端虚拟寄存器/立即数”形式。
 - 选对 AArch64 指令；能塞立即数就塞，否则先装进寄存器。
 - % 要拆成除 \rightarrow 乘 \rightarrow 减三个步骤。
 - 最后落地一条 dst = lhs op rhs 的目标指令，等待后续寄存器分配/栈处理

具体一点讲：

面向小白的大白话总结：IRIsel 遍历 IR，每见到一种 IR 指令就调用对应的 `visit`，生成“带虚拟寄存器的 AArch64 指令蓝图”，供后续寄存器分配/栈处理/汇编输出使用。

核心术语再解释

- 按 ABI 映射参数：遵守 AArch64 调用约定，把前几个参数放指定物理寄存器，多出来的按 8 字节对齐压栈。浮点参数用 v0-v7（实际名 s0/d0），整数/指针用 x0-x7。
- 普通指令：除 phi 以外的所有 IR 指令（算术、load/store、call、branch...）。
- 统一降 phi：先跳过 phi，等基本块指令都生成后，再在各前驱边上插 move，解决 SSA 合流。
- 虚拟寄存器：后端临时编号的寄存器，不是真实硬件寄存器，后面寄存器分配时才映射为 x0/x1... 或溢出到栈。
- LDR/STR：AArch64 的加载/存储指令，LDR 读内存到寄存器，STR 写寄存器到内存。
- addr_map：记录某个 IR SSA 值当前对应的“地址来源”信息（是帧槽？全局符号？已有寄存器？附带偏移多少）。方便后续 load/store/GEP 复用。
- 值物化（materialize）：把一个抽象的 IR 操作数（可能是立即数、全局符号等）具体化成寄存器或可编码的立即数，以便发目标指令。
- 参数溢出上栈：当参数数量超过 ABI 提供的寄存器（整数>8、浮点>8）时，剩余的参数按顺序放到调用者栈帧的“参数区”里（8 字节对齐），被调用者用栈偏移取出。

支持的 IR 类型与处理方式

- Module / Function / Block
 - `visit(Module)`：复制全局变量到后端模块，逐函数处理。
 - `visit(Function)`：按 ABI 映射参数到虚拟寄存器/栈，创建基本块，先降普通指令，最后统一降 phi。
 - `visit(Block)`：空壳，真正工作在遍历块内指令时完成。
- Load / Store
 - `visit(LoadInst)`：根据地址来源（栈帧/全局/寄存器）发 LDR，把值装入目标虚拟寄存器；指针结果会更新 `addr_map`。
 - `visit(StoreInst)`：先把待存的值物化（寄存器或装立即数），再 STR 写入目标地址。
- Arithmetic（含位运算、移位、浮点）
 - `visit(ArithmeticInst)`：选择对应 AArch64 指令；能用立即数就直接编码，不能就先装寄存器；`mod` 展开为 div→mul→sub。
- Compare
 - `visit(IcmpInst)`：CMP + CSET，产出 i32 的 0/1。
 - `visit(FcmpInst)`：FCMP + CSET，同样产出 i32 的 0/1。
- Alloca
 - `visit(AllocaInst)`：为局部分配栈槽，生成指向该槽的指针寄存器，记录到 `addr_map`。
- Branch
 - `visit(BrCondInst)`：先 CMP cond vs 0，再 BNE 到真分支，B 到假分支。
 - `visit(BrUncondInst)`：直接 B 到目标块。

- Call / Return
 - `visit(CallInst)`: 按 ABI 摆放参数 (浮点进 v0-v7, 整数进 x0-x7, 溢出上栈), 发 BL, 返回值从 x0/v0 搬到虚拟寄存器。
 - `visit(RetInst)`: 把返回值放入 x0/v0 (i32 需扩展到 x0), 发 RET。
- GEP (GetElementPtr)
 - `visit(GEPIInst)`: 计算地址 = 基址 + 常量偏移 [+ 动态索引*元素大小], 结果寄存器更新 `addr_map`。
- 类型转换
 - `visit(FP2SIIInst)`: FCVTZS f32 -> i32。
 - `visit(SI2FPIInst)`: SCVTF i32 -> f32。
 - `visit(ZextInst)`: 宽度扩大时直接 move。
- Phi
 - `visit(PhiInst)`: 在每条前驱边插入 move 搬值, 必要时拆边避免冲突, 完成 SSA 合流。
- 不期望出现的 IR
 - `visit(GlbVarDeclInst / FuncDeclInst / FuncDefInst)`: 防呆, ISel 阶段不应看到这些, 若出现会报错。

Float 与数组处理要点

- 浮点常量: 大多指令不支持直接嵌入浮点立即数, 采用 "int 装值 + FMOV" 物化为寄存器 (`materializeF32`)。
- 浮点运算: FADD/FSUB/FMUL/FDIV, 操作数需在浮点虚拟寄存器。
- 浮点比较: FCMP + CSET 产出 i32 结果。
- 浮点参数与返回: 按 ABI 用 v0-v7 (s/d 寄存器名), 返回值在 v0。
- 数组与 GEP:
 - alloca 数组: 在栈上分配连续空间, 得到基址指针。
 - GEP 计算偏移: 常量索引用常量偏移, 变量索引用 SXTW 扩展为 64 位, 再乘元素大小, 加到基址。
 - 结果地址会写入 `addr_map`, 后续 load/store 直接用。

指令组合再解释

- CMP/CSET (整型比较) : CMP 设置标志, CSET 把标志转成寄存器里的 0/1。
- FCMP/CSET (浮点比较) : FCMP 设置浮点比较标志, CSET 同样转成 0/1。
- LDR/STR: Load/Store 指令, 访问内存。
- LA: 装载全局符号地址到寄存器 (伪指令)。
- MOVZ/MOVK: 组合装载大立即数到寄存器。
- BL: 函数调用跳转, 保存返回地址到 LR。
- RET: 从 LR 返回。

更多示例与步骤

- 示例1: `c = a + 5` (立即数可编码)

1. dst 为 c 分配虚拟寄存器。
 2. lhs 取 a 对应的虚拟寄存器。
 3. 5 可放进 ADD 的 imm12, rhs 用立即数。
 4. 发 `ADD dst, lhs, #5`。
- 示例2: `c = a + 50000` (立即数放不下)
 1. dst、lhs 同上。
 2. 50000 不能直接编码, 用 MOVZ/MOVK 把 50000 装进 tmp。
 3. rhs 用 tmp。
 4. 发 `ADD dst, lhs, tmp`。
 - 示例3: `r = a % b`
 1. `SDIV q, a, b` 得商。
 2. `MUL t, q, b` 得商*除数。
 3. `SUB r, a, t` 得余数。
 - 示例4: 条件分支 `if (cond) T else F`
 1. cond 在寄存器。
 2. `CMP cond, #0`。
 3. `BNE T; B F`。
 - 示例5: 加载全局 `x = *g`
 1. `LA tmp, g` 装地址。
 2. `LDR dst, [tmp]` 读内存。
 - 示例6: 函数调用 (整数多参溢出上栈) `ret = foo(a1, ..., a10)`
 1. a1-a8 放 x0-x7; a9、a10 按 8 字节对齐写到当前栈帧的参数区 (sp 偏移)。
 2. `BL foo`。
 3. 返回值从 x0 搬到虚拟寄存器; 若是指针, addr_map 记录为寄存器来源。
 - 示例7: 浮点调用 `f = bar(x, y)` (float 参数)
 1. x,y 物化为浮点寄存器; 放 v0,v1。
 2. `BL bar`。
 3. 结果在 v0, 搬到目标虚拟寄存器。
 - 示例8: alloca + GEP 访问数组元素 `p = &arr[i]` (arr 是 alloca 的)
 1. alloca 生成 arr 基址 = sp + offset (带 frame index)。
 2. i (i32) 用 SXTW 扩展为 i64 idx64。
 3. 若元素大小 e, 比如 4 字节: `MUL stride = idx64 * e`。
 4. `ADD p = base + stride`, 必要时再加常量偏移。
 5. addr_map 记录 p 的来源 (Reg 或 Frame+offset)。
 - 示例9: 浮点比较 `r = (a < b)`
 1. a,b 在浮点寄存器。
 2. `FCMP a, b`。
 3. `CSET r, LT`, r 为 i32 0/1。

- 示例10: Zext `y = zext i32 a to i64`

1. a 在 i32 虚拟寄存器。
2. 直接 move 到 i64 虚拟寄存器 (零扩展对 move 足够)。

记忆要点

- 能直接编码的立即数就内联，否则先用 MOVZ/MOVK 装寄存器。
 - CMP/FCMP + CSET 把比较结果变成 0/1 的 i32，后续分支用这个。
 - Phi 靠“在前驱边插 move”来消除 SSA，需要时拆边避免互相覆盖。
 - 调用/返回严格按 AArch64 ABI：参数寄存器优先，溢出上栈，返回值在 x0/v0。
-

第二步——预留的RA

- 作用（理想版）：在寄存器分配前，先把“栈框架/帧索引/前言后语”准备好或打标记，例如：
 - 估出当前函数的局部栈对象，对齐/预留空间的需求。
 - 标记需要保存/恢复的被调用者保存寄存器（callee-saved）。
 - 规划 frame index 的使用方式，让后续能替换成具体的 sp/bp 偏移。
 - 如果需要，还可提前插入伪指令，后续再由栈阶段填充成真实 prologue/epilogue。
 - 调用链：`AArch64Target::runPipeline` → `FrameLoweringPass::runOnModule` → `lowerFunction`（当前空实现）。
 - 当前实现的现实版：
 - `lowerFunction` 里什么都不做 ((void)func)，只是把入口留出来。
 - 真正的栈大小计算、frame index 替换、前言/后语生成、callee-saved 保存恢复，全都在后面的 `StackLoweringPass` 完成。
 - 为什么保留一个空 Pass：
 1. 保持流水线结构清晰：预 RA 的“框架层”有名有分，后续要加逻辑有落点。
 2. 扩展方便：若以后要在 RA 前做对齐、特殊帧布局、或为 GC/调试插桩打标记，可以直接在这里补实现，而不用改流水线。
 3. 兼容习惯：许多后端都有 FrameLowering/StackLowering 两段，习惯分层，阅读者容易对齐心智模型。
-

(Key)第三步——寄存器分配 (Linear_scan.cpp) (虚拟reg->物理reg)

- 调用链：`runPipeline` → `LinearScanRA::allocate`（模板基类驱动）→ 对每个函数调用 `LinearScanRA::allocateFunction`。

“对每个函数调用”是说：`LinearScanRA::allocate`会遍历整个模块的所有函数，每个函数各跑一遍 `allocateFunction` 做寄存器分配。

你现在 ISel 完成后，手里的是目标相关的 MIR，指令已经是 AArch64 形态 (ADD/SUB/...)，但操作数还是“虚拟寄存器 + 少量立即数 + 帧索引占位”等。线性扫描要处理的，就是这些带虚拟寄存器的目标指令，把虚拟寄存器换成物理寄存器；不够用就插入 spill/reload，用栈槽和预留的临时寄存器搬运。换句话说，输入形如 add v3, v1, v2 或 add v3, v1, #imm，输出会把 v1/v2/v3 映射成 xN/vN 或栈槽访问，

保持指令种类不变，只替换寄存器并补充必要的溢出/回填指令。

- 核心目的：把“虚拟寄存器”映射到“物理寄存器”，不够用就“溢出”（spill）到栈，并在用到前“回填”（reload）。
- 关键术语大白话：
 - 虚拟寄存器：编译器自编的编号，想用多少用多少，后面才真的分给硬件寄存器。
 - 物理寄存器：芯片上真的那些寄存器（x0-x30, v0-v31）。
 - 活跃区间：一段代码里某个值“还要被用到”的区间范围。两个值的区间重叠就不能抢同一个寄存器。
 - 溢出/回填（spill/reload）：寄存器不够时，把某个值先存到栈里（spill），等要用再从栈里读回来（reload）。
 - caller-saved / callee-saved：
 - caller-saved（调用者保存）：调用前得自己存好，调用后再恢复，常用的会在调用点被破坏。
 - callee-saved（被调用者保存）：被调函数进来要先存好、出去要还原，跨调用更安全。
 - scratch 寄存器：专门预留来做 spill/reload 的临时寄存器，避免和正常分配冲突。

内部主要步骤（整数/浮点各跑一遍，流程相同，大白话+例子）：

1. 给指令排座位号：

- 把函数里所有指令从上到下编号 0, 1, 2, ..., 记录每个基本块的范围。
- 遇到调用指令（call）做个记号，后面要优先给跨调用的值用“稳妥”的寄存器。
- 类比：发号排队，知道谁在前谁在后，顺便标出“电话亭”（call 点）。

2. 看每条指令用了谁、定义了谁（USE/DEF），算哪些值一直活着：

- 对每条指令，列出读的寄存器（USE）和写的寄存器（DEF）。
- 做个流转表：哪些值在基本块入口还需要（IN），哪些在出口还需要（OUT）。
- 类比：谁手上还捧着东西要带到下一站，谁用完就扔了。

3. 倒着走一遍，画出每个值活着的区间：

- 从块尾往前扫，凡是“还活着”的寄存器，就给它的活跃区间加一段；遇到定义就重置它的开始。
- 合并相邻或重叠的段。
- 例：a 在指令 2 定义，在 2-6 被用，到 7 不再用，则区间 [2, 7)。

4. 标记“跨电话”的值：

- 如果某个值的区间覆盖了 call 指令的位置，就记它“跨调用”。
- 这样后面优先给它发“稳妥的被调用者保存寄存器”，免得打电话时被冲掉。

5. 线性扫描分座位：

- 先按区间起点排序，像按进场时间排队。
- 维护一个“正在占座”的集合 **active**，记录当前哪些值还活着、占了哪些物理寄存器。
- 处理下一个区间前，先把已经结束的值移出 **active**，腾出座位。
- 有空座（空闲物理寄存器）就坐下；没空座就挑一个人去“存包”（**spill** 到栈）：
 - 规则：通常挑“活得更久”的去存包，让“新人”坐下；跨调用的值更想坐“稳妥座位”（**callee-saved**）。
 - 例：物理寄存器池只有 2 个，来了 $a([0, 10))$ 、 $b([1, 3))$ 、 $c([4, 8))$ ：
 - 进 a ，占座1。
 - 进 b ，占座2。
 - b 在 3 结束，腾出座2。
 - 进 c ，此时 **active** 只剩 a ，占座2 给 c ，用完 8 结束； a 10 结束。

6. 补“存包/取包”指令 (spill/reload) :

- 对被 **spill** 的值：在它下次要用前，先插一条 **LDR**（取包）；在它定义完后，插一条 **STR**（存包）。
- 搬运用预留的 **scratch** 寄存器，避免和正常分配冲突。
- 例： d 被挑去存包，定义后插 `STR tmp, [slot_d]`；后面用 d 前插 `LDR tmp, [slot_d]`，实际运算用 tmp 。

寄存器分配示例（更口语版）

- 例1：寄存器够用（无溢出）
 1. 两个值 a 、 b 的活跃区间不重叠，各拿到不同物理寄存器，比如 $a \rightarrow x_1$, $b \rightarrow x_2$ 。
 2. 指令里直接用 x_1, x_2 ，无需动栈，等 a 用完后 x_1 还能被下一个值 c 复用，不会浪费。
 3. 这种情况线性扫描几乎不用插入额外指令，生成的汇编最干净。
- 例2：寄存器不够，发生溢出
 1. 同时活跃的值有 a, b, c, d ，物理寄存器池只够分三个。
 2. 给 a, b, c 分到寄存器， d 被挑去 **spill**：在定义 d 后插入 **STR d, [spill槽]**，把 d 放栈里。
 3. 后面用到 d 时，先插一条 **LDR tmp, [spill槽]**，再用 tmp 做运算； tmp 用 **scratch** 寄存器（一次搬一个值，搬完就还回去，可多次复用）。
 4. 如果 d 在后面又被更新，新的值会再次 **STR** 回同一个槽，形成“写-存-读-用”的循环，保证语义正确。
- 例3：跨调用的值优先 **callee-saved**
 1. 有个值 v 要在调用前算好，调用后还要用，它的活跃区间覆盖了 **call**。
 2. 线性扫描时，会优先把 v 分到 **callee-saved**（比如 $x_{19}-x_{28}$ ），这样被调函数回来后值还在。
 3. 若 **callee-saved** 也不够，就只能让 v **spill** 到栈，再在 **call** 后用 **LDR** 取回；相比放 **caller-saved** 省掉在调用点前后的存/取。
- 例4：浮点与整数分开分配
 1. 浮点活跃区间只在浮点寄存器池里竞争 (v_0-v_{31})，不会和整数池抢 (x_0-x_{30})。
 2. 浮点溢出时也用各自的 **spill** 槽（8 字节对齐），回填用预留的浮点 **scratch**；整数溢出/回填则用整数 **scratch**，互不干扰。
 3. 因为池子独立，出现“整数紧张但浮点空闲”时，浮点寄存器不会被借用来救急，遵守硬件分类。
- 例5：遇到调用参数/返回值的物理寄存器
 1. 枚举指令里显式出现的物理寄存器（如 x_0, v_0 ），把这些从可分配池里剔除，避免误分配破坏调用约定。

2. 例如一次调用会用 x0-x3 传参，则这几个在活跃区间覆盖该调用的时段内都视为“占用”，不会再分配给别的值。
 3. 这样做能防止“分配到 x0 的本地值在调用时被参数覆盖”这类 ABI 冲突。
- 例6：活跃区间重叠导致互抢
 1. 如果 a 的区间是 [0,10), b 是 [5,15), 有重叠，不能同寄存器。
 2. 若同时还有 c 的区间 [10,20), c 可以复用 a 占过的寄存器，因为 [0,10) 结束后回收，线性扫描在 10 点会把 a 从 active 集合移除。
 3. 这种“先用后还再复用”的模式是线性扫描的核心优势：逻辑简单、只按时间线推进，不需要全局图着色。

Stack_Lowering:

下面用更口语、更细的方式，把栈降低的 6 步讲清楚，并加一个完整小例子串一遍。对应代码可对照 stack_lowering.cpp。

在指令序列里插入一串装立即数到寄存器的指令 (AArch64 常规套路：MOVZ + 若干 MOVK)，并把迭代器 it 前移，让后续遍历仍然有效。细节逐行看：

- 接口：传入当前基本块、指令迭代器 \square it (会在插入后指向下一条)、目标寄存器 \square reg、要装入的 64 位立即数 \square val。
- 特判 \square val == 0：用一条 \square MOVZ reg, #0, lsl #0 即可，把它插入到 \square it 之前；插完 \square it++。
- 把 \square val 转成无符号 \square uval，因为 MOVZ/MOVK 都是按 16 位分片处理。
- for i=0..3：每次取低 16 位片段 (part = uval >> (i*16))，如果这个 16 位非 0，就发一条指令：
- 第一段非 0 用 \square MOVZ reg, #part, lsl #(i*16)，清零高位并写入这 16 位；
- 后续非 0 段用 \square MOVK reg, #part, lsl #(i*16)，在不破坏其他位的情况下“打补丁”写入对应 16 位。
- 每插入一条都放在 \square it 前面，并 \square it++，这样外层遍历的当前位置始终指向插入片段之后的原指令，避免死循环或跳过。

总效果：在当前位置之前生成 1~4 条 MOVZ/MOVK，把 64 位立即数完整写入指定寄存器。

- Step 1：算栈帧、决定要存哪些寄存器（大致见 stack_lowering.cpp:40-114）
 - 先让 frameInfo.calculateOffsets() 把局部变量、溢出槽（spill 槽）和传出参数区加总，并向上 16 字节对齐，得到“基础栈大小”。
 - 扫全函数的指令，把真正出现过的物理寄存器找出来；和 callee-saved 列表（被调者要负责还原的寄存器）做交集，只保存确实被用到的那些，避免白存。
 - 给要保存的寄存器分配一段“保存区”，再按 16 对齐。这段大小叠加到基础栈大小，得到最终的 frameSizeTotal。它决定 sp 会下调多少。
- Step 2：把 FrameIndex 全换成 sp+偏移（见 stack_lowering.cpp:116-193）

FrameIndex 操作数是编译器中间表示里的“栈槽占位符”：它用一个编号指代某个帧对象（局部变量、溢出槽、形参保存区或临时栈空间），在寄存器分配/栈布局确定之前先占位。到了栈降低阶段，FrameIndex 会被具体的基址+偏移替换（通常是 \square sp + 实际偏移，或 fp + 偏移），变成真实可执行的内存操作数

- 每个帧对象/溢出槽都有“相对帧底”的偏移；再加上保存区大小，就得到相对 sp 的最终偏移。
- 偏移若能直接编码进指令（LDR/STR 的小偏移或 scaled 偏移，LDP/STP 的 scaled 偏移），就直接把立即数写进去。
- 放不下：用临时寄存器 x15 装偏移，再 ADD sp, x15 得到基址，内存偏移填 0。
- 如果原来是立即数字面量（例如地址物化）且放得下，就改立即数；放不下，同样装进 x15 变成寄存器操作数。
- 归底：若还有偏移没处理且非零，再用 x15 拼出三操作数形式，彻底消掉帧索引标记。

- Step 3: 把 FILoad/FIStore 伪指令变成真 LDR/STR (见 stack_lowering.cpp:195-270)
 - 先算出每个溢出槽的最终偏移 (同样加保存区)。
 - 偏移能编码就直接发出 [sp,#off] 的 LDR/STR。
 - 编不了: 用 x15 装偏移, ADD sp, x15 当基址, 再发 LDR/STR, 偏移填 0。
 - 伪指令被真指令替换掉。

这段是 Step 3: 把寄存器分配阶段生成的溢出/回填伪指令 (FILoad/FIStore) 展开成真实的 LDR/STR。逐行拆解:

- 外层 for (func->blocks): 遍历所有基本块。
- 内层 for (it = block->insts.begin(); ...): 遍历指令列表, 遇到伪指令就替换; 用迭代器是为了在插入/删除后还能继续正确行走。

处理 FILoad (从溢出槽加载回寄存器) :

- offset = getSpillSlotOffset(fiLoad->frameIndex); offset += savedAreaSize;
- 先取溢出槽的偏移, 再加上保存区大小, 得到相对 sp 的最终偏移。
- size: 根据目标寄存器类型决定 4 字节还是 8 字节的访存。
- 尝试直接用 [sp,#offset]: fitsLdrStrlImm(offset, size) 检查偏移是否能编码进 LDR/STR。
- 如果能编码: mem = new MemOperand(sp, offset), 直接生成 LDR dest, [sp,#offset]。
- 如果放不下: 用预留的 x15 装入偏移, 再 ADD x15, sp, x15 做出一个基址寄存器, 内存偏移置 0, 生成 LDR dest, [x15,#0]。
- insertLoadImm(block, it, tmpReg, offset) 插入 MOVZ/MOVK 序列把偏移装入 x15。
- 插入 ADD tmpReg, sp, tmpReg 计算基址。
- mem 指向这个基址寄存器。
- 最后把伪指令替换掉: 先 erase 掉原来的 FILoad, 再在同一位置插入真实的 LDR。

处理 FIStore (把寄存器值写回溢出槽) :

- 同样计算 offset += savedAreaSize, 决定访存大小 4/8 字节。
- 能编码直接 [sp,#offset] 的 STR, 否则用 x15 装偏移 + ADD sp, x15 做基址, 偏移 0, 再发 STR。
- 同样先删伪指令、再插入真实的 STR src, [base]。

核心逻辑小结:

- 先算最终偏移 (包含保存区)。
- 能塞立即数就用 sp + 立即数的直接寻址。
- 不能塞就用 x15 做“偏移寄存器”, 先装偏移、再加 sp, 当作新的基址寄存器, 访存偏移改为 0。

• 用生成的真 LDR/STR 替换掉伪指令, 迭代器前移继续扫描。

- Step 4: 给要保存的寄存器安排槽位 (见 stack_lowering.cpp:272-312)
 - 保存区紧跟传出参数区, 把每个要保存的寄存器对应的“栈内偏移”记下来, 后面序言/尾声用它来存/取。
- Step 5: 插入序言 (prologue) (见 stack_lowering.cpp:314-363)
 1. STP x29, x30, [sp, #-16]!: 一把把旧帧指针和返回地址压栈, 同时 sp 下调 16。
 2. mov x29, sp: 把新帧指针设到当前 sp。
 3. 下调 sp, 预留整帧空间:
 - 如果 frameSizeTotal 能放进指令的 12 位 (或类似可编码范围), 用 SUB sp, sp, #imm。
 - 否则先用 x9 装大立即数, 再 SUB sp, sp, x9。
 4. 依次把 callee-saved 存进各自槽位:
 - 偏移能编码就直接 STR;
 - 放不下就 x9 装偏移, ADD sp, x9 作基址, 再 STR。
- Step 6: 在每个 RET 前插尾声 (epilogue) (见 stack_lowering.cpp:365-424)

1. 按相反顺序恢复 callee-saved：能编码就直接 LDR，放不下用 x9+ADD+LDR。
2. 抬回 sp：能编码就 ADD sp, sp, #imm，否则 x9 装数再 ADD。
3. LDP x29, x30, [sp], #16：一把拉回帧指针/返回地址，同时把 sp 恢复到进函数前的位置，然后 RET。

这里逐行说明这段序言/尾声插入逻辑（对应 stack_lowering.cpp:386-527）：

- 判空：函数 block 列表非空才插序言/尾声。
- 入口块定位：entryBlock 取第一个块，it 指向其指令首迭代器，后续 insert 都在 it 前插入，再 ++it 保持遍历正确。
- Step 5-1 序言压栈：STP x29,x30,[sp,#-16]! 预留 16 字节同时把旧 fp/lr 压栈，sp 下调（前递更新寻址）。
- Step 5-2 立新帧指针：mov x29, sp。
- Step 5-3 预留整帧：若 frameSizeTotal 可编码 12-bit unsigned 立即数，则 SUB sp, sp, #frameSizeTotal；否则用 x9 先 insertLoadImm 装立即数，再 SUB sp, sp, x9。
- Step 5-4 保存 callee-saved：遍历 saveSlots（已按偏移排好），调用 insertStoreCallee。能用无符号 scaled 立即数就直接 STR reg, [sp,#off]；否则 x9 装偏移 + ADD x9, sp, x9，再 STR reg, [x9]。此处使用 x9 避免与 x15 (FI 展开) 冲突。
- 尾声插入：遍历所有块，扫描指令找 RET，在其前面插入恢复序列。
 - Step 6-1 逆序恢复 callee-saved：saveSlots 反向遍历，调用 insertLoadCallee。同样先尝试 LDR reg, [sp,#off]，放不下则 x9 装偏移 + ADD x9, sp, x9 + LDR reg,[x9]。
 - Step 6-2 抬栈：若 frameSizeTotal fits 12-bit imm，用 ADD sp, sp, #frameSizeTotal；否则 x9 装立即数，再 ADD sp, sp, x9。
 - Step 6-3 恢复 fp/lr 并回收 16 字节：LDP x29,x30,[sp],#16 使用后递增寻址，一步提回 sp 并弹出旧 fp/lr。之后原本的 RET 继续执行。
- 关键寄存器选择：x15 专用于 FI 展开的大偏移基址构造；x9 专用于 callee-saved 偏移超范围的情况，避免相互踩踏。
- 立即数判定：序言/尾声调栈用 fitsUnsignedImm12 (AArch64 12-bit 无符号)；callee-saved 用 fitsUnsignedScaledOffset (按 8 字节缩放)；FI 展开用 fitsLdrStrImm/fitsLdpStpImm (含有符号小偏移与无符号 scaled)。

插入序言/尾声的目的：进入函数时搭好自己的栈框架并保存必要状态，退出时把现场恢复干净、返回给调用者。对应实现在 stack_lowering.cpp 的后半段。

- 序言 (prologue) 做的事：
 - 把旧帧指针和返回地址一起压栈，同时下调 sp (STP x29, x30, [sp, #-16]!)，保证回来时能复原。
 - 把 x29 设为新的帧指针，后续用它或 sp 作为基址访问局部变量/溢出槽。
 - 按计算好的 frameSizeTotal 再下调 sp，预留整帧空间给局部、溢出、传出区、callee-saved 保存区。
 - 将实际用到的 callee-saved 寄存器写入各自的保存槽（偏移能编码就直接 STR，放不下就先装偏移再存），确保之后被覆盖也能恢复。
- 尾声 (epilogue) 做的事（每个 RET 前）：
 - 逆序把前面存下的 callee-saved 取回（偏移能编码直接 LDR，否则装偏移+ADD 后 LDR），恢复调用者的寄存器状态。
 - 把 sp 抬回去（与序言的下调对称：小立即数用 ADD，大立即数先装寄存器再 ADD）。
 - 用 LDP x29, x30, [sp], #16 一次性恢复旧帧指针和返回地址，并把 sp 提回函数入口时的位置。
 - 最后 RET，干净地把控制权交还调用者。

简言之：序言“先存后减栈”，尾声“先还再加栈”，确保函数内部怎么折腾都不影响调用者看到的寄存器和栈状态。

贯穿例子（更细的口语版）

假设函数情况：

- 局部+溢出一共 24 字节，传出区 16 字节，用到了 callee-saved 的 x19。
- 先对齐：局部区向上到 32，传出区 16，保存 x19 需要 8 字节，再对齐到 16，所以保存区当作 16。总栈帧 frameSizeTotal = 32 + 16 + 16 = 64。这意味着 sp 会整体下调 64 字节。

跑流程：

1. 计算阶段：得到 64，总结出需要保存 x19。
2. 替换帧索引：某局部的偏移是 -8（离 sp 向下 8），加上保存区 16 变成 -24。-24 可编码进 STR/LDR 的小偏移，就把指令的偏移直接改成 24（对 sp 的正偏移写法）。另一个槽在 -520，放不进立即数字段，于是把 520 装进 x15，ADD x15, sp, x15 当基址，偏移写 0。
3. FILoad/FIStore：溢出槽在 -40，加保存区后偏移 -56，正向写是 56。56 能编码，就直接发 LDR reg, [sp,#56]。
4. 保存槽：x19 对应的槽偏移是保存区里的前 8 字节（例如 0 到 7），整体相对 sp 是“传出区 16 + 偏移 0”，也就是 16；若再加上局部区后移，最终存取用这个偏移。
5. 序言发出的指令：
 - STP x29, x30, [sp,#-16]!
 - mov x29, sp
 - SUB sp, sp, #64（64 可编码，直接用立即数）
 - 存 x19：偏移 16 能编码，发 STR x19, [sp,#16]。
6. 尾声在 RET 前：
 - 恢复 x19：LDR x19, [sp,#16]
 - ADD sp, sp, #64 拾栈
 - LDP x29, x30, [sp], #16
 - RET

这样一趟下来，所有帧索引都被替换成真实 sp+偏移，伪指令展开完毕，保存/恢复寄存器和栈指针调整也都插好了，返回时栈帧回到进入函数前的状态。

最后一步：codegen

作用：这一段是 AArch64 后端流水线的最后一步，把已经做完指令选择、寄存器分配、栈降低的 backend 模块输出为汇编文本到 out 流（通常是文件流或 stdout）。参见 aarch64_target.cpp:62-73。

- 流程：
 - 构造 BE::AArch64::Codegen cg(backend, *out)；把 MIR/选后指令的模块和输出流交给代码生成器。
 - 调用 cg.generateAssembly()；遍历函数/基本块/指令，序列化为 AArch64 汇编文本（含全局符号、伪指令、指令、数据段等），写入 out。
- 上下文：前面步骤已完成 ISel、FrameLowering（预 RA）、LinearScanRA、StackLowering，backend 此时应是“可直接发射”的 AArch64 指令序列，无伪指令/FrameIndex。

把“菜谱”变成“菜”的类比：有菜单（模块），每道菜是一个函数，每勺翻炒是一个基本块，每个动作是指令。最后把菜名和做法写到一本书（汇编文本）里。

这段代码干啥

- 文件位置：backend/targets/aarch64/aarch64_codegen.cpp
- 目的：把内部的“指令对象”翻译成 armv8-a 汇编文字，写进输出流。

- 顺序：函数 → 基本块 → 指令，层层遍历，逐行写字符串。

总流程 (generateAssembly)

- 开头声明：写 `.text` 和 `.arch armv8-a`，告诉汇编器“这是代码，跑在 armv8-a 上”。
- 逐个函数生成代码：`emitFunction`。
- 如果有全局变量，切换到 `.data` 段，按类型写 `.word/.quad` 或 `.zero`（连续 0）。

做一“道菜” (emitFunction)

- 标题：`.globl 函数名`，然后 `函数名:`。
- 逐块输出：对每个基本块调用 `emitBlock`。

炒一“勺” (emitBlock)

- 块标签：`.函数名_块号:`。
- 逐条指令：交给 `emitInstruction`。

下锅 (emitInstruction)

- 特判伪指令 MOVE：看目的寄存器是不是浮点，决定用 `mov` 还是 `fmov`，附带行内注释。
- 其他指令必须是目标指令，否则报错。
- 特判伪指令 LA：输出 `ldr 目标寄存器, =符号`（装载符号地址）。
- 决定格式的关键：`opType`（指令类型枚举），按类别拼字符串：
 - `OpType::L`: 标签跳转，写 `.函数名_块号`。
 - `OpType::SYM`: 符号操作数，直接写符号名。
 - `OpType::P`: 成对存取 STP/LDP，打印两寄存器 + 基址 + 偏移，区分前递 `!` / 后递。
 - `OpType::R2`: 双操作数，内含特判：
 - `LA: ldr dst, =sym`
 - `CMP`: 第二个若是立即数加 `#`，否则寄存器
 - `CSET`: 把数字条件码翻成 eq/ne/lt/... 再输出
 - `MOV`: 若源/目的都是浮点寄存器，用 `fmov`，否则 `mov`
 - 其他：直接“左，右”
 - `OpType::R`: 三操作数：
 - `MOVZ/MOVK/MOVN`: 支持 `ls1 #shift`
 - 其他：`dst, src1, src2/imm`
 - `OpType::M`: 访存二操作数（ldr/str 形式）：`[寄存器, [基址, #偏移]]`
 - `OpType::Z`: 零操作数或特殊格式：
 - `RET`: 不加额外内容
 - `STP/LDP`: 这里也有一份格式特判（前递/后递）
- 如果指令对象里带 `comment`，在行尾追加 `// ...`。

食材预处理 (formatOperand)

- 寄存器：用 `formatRegister` 显示成 `x0/x1/...` 或 `s0/d0`。
- 立即数：前面加 `#`。
- 浮点常数：补 `.0` 再加 `#`。
- 内存：打印成 `[base, #offset]`。

一句话总结

遍历模块里的函数、块和指令，把每个操作数和指令按规则变成 armv8-a 文本格式，先写代码段，再写数据段，最终输出一份可被汇编器吃下去的“菜谱说明书”。

关键步骤细讲（用最直白的说法）

1) IR逐条翻译：把 .II 变成机器草稿

- 做什么：一行 IR 换成一小段 AArch64 机器指令，先不管物理寄存器，用“虚拟寄存器”占位。数组/局部变量的地址记成“帧索引”占位。
- 怎么做：
 - 函数参数：从 aarch64 约定的寄存器 `x0-x7/d0-d7` 里拷到虚拟寄存器，溢出参数从栈取。
 - alloca：为局部留栈槽，生成一条 `add v, sp, #0`，并附上帧索引占位。
 - load/store：全局先 `la` 取地址再读写；栈对象用帧索引占位，偏移以后再算；普通指针直接 `ldr/str`。
 - 算术/逻辑：直接用 `add/sub/mul` 等；立即数放不下就先 `movz/movk` 物化。
 - 比较/条件跳转：`cmp/fcmp + cset` 得布尔值，再 `bne/b...` 跳转。
 - 调用：参数按约定放寄存器或压栈，记录传出参数最大空间，`bl` 调用，返回值从 `x0/d0` 拷到目标虚拟寄存器。
 - GEP：算好常量偏移，变量索引用 `sxtw` 扩展，乘 `stride`，再加基址（栈基址用帧索引占位）。
 - Phi：按边插入 `move`，有多后继会拆边建小块，保证不打架。

2) 线性扫描寄存器分配：给每个虚拟寄存器找座位

- 做什么：把虚拟寄存器换成真实物理寄存器；不够用就溢出到栈，并插入 `reload/spill` 伪指令。
- 怎么做：
 - 统计每条指令用/定义哪些寄存器，做活跃性分析，算出每个虚拟寄存器的生存区间。
 - 按起点排序扫描：能回收的先回收，再从可用物理寄存器池里挑；没位子就挑“活得更久”的去栈里暂存。
 - 对没分到位的 use/def：用预留的 scratch 物理寄存器搬一次，并在前后插入 `FILoad/FIStore` 伪指令访问溢出槽。

3) 栈降低：把占位符变成真实偏移，补上序言/尾声

- 做什么：
 - 重新计算栈大小（局部 + 溢出槽 + 传出参数区），16 字节对齐。
 - 找出真正用到的被调者保存寄存器，分配保存区。
 - 把帧索引占位、FI 伪指令变成具体的 ldr/str，偏移塞进指令，放不下就用临时寄存器装偏移。
 - 插入函数序言：保存 fp/lr，设 fp，sp 下调栈大小，再保存 callee-saved。
 - 在 ret 前插入尾声：恢复 callee-saved，sp 上调，ldp 恢复 fp/lr。

4) 汇编打印：写出最终 .s

- 逐块逐指令吐文本，寄存器用 x/w/d/s 名字，内存用 [base, #off]，全局数据段输出 .word/.quad/.zero。