目标代码生成

杨科迪 费迪
徐文斌 贺祎昕 谢子涵
华志远 郭大玮
唐显达 蒋佳豪 王俊杰

2025 年 12 月 5 日

目录

1 实验描述

经过一个学期的学习，编译器的构造到了令人激动的最终阶段，这将是这学期编译原理课程最具成就感的一步，实现编译器的程序员就可以称为一个“浪漫”的程序员了。在本次实验中，同学们需要将中间代码转化为目标代码，最后运行你生成的目标代码，检测其正确性。

2 实验要求

1. 完成指令选择和寄存器分配，寄存器分配可以采用平凡算法 (例如将所有寄存器都溢出到栈上)。最终生成正确的 arm/riscv 目标代码。
2. 在雨课堂的提交栏中提交本次实验的 gitlab 链接。
3. 上机课时，以小组为单位，向助教讲解程序。

3 目标代码生成

目标代码生成宏观上可以分为以下几个步骤：控制流图构建、指令选择、寄存器分配、生成目标代码。接下来我们将详细讲解每个步骤的实现方法。

3.1 控制流图构建

在中间代码生成阶段，我们已经掌握了基本块的划分方式和它们之间的跳转关系，并在每个基本块内插入了对应的 LLVM 指令，最终构建存储 LLVM 指令的控制流图。进入目标代码生成阶段，后端同样需要合适的数据结构来存储这些基本块及其关系。

因此，我们需要重新构建控制流图，用于后续的目标代码生成。在这个过程中，每个基本块将存储对应的 Arm 或 RISC-V 汇编指令。

3.2 实现指令选择

编译器将平台无关的中间语言 (Intermediate Representation) 转换为平台相关的机器指令 (Machine Instruction) 的过程。

后端目前支持两条指令选择路径，差异如下：

1. SelectionDAG 模式：通过 backend/dag/* 和各 target/isel/* 构建、合法化 SelectionDAG，再用匹配规则把 DAG 节点映射为汇编指令；天然支持指令融合、常量折叠、地址模式等局部优化，并为复杂目标的 pattern matching 提供基础设施。
2. 逐条指令转换：跳过 DAG，按 IR 语义手写 lowering，把每条 IR 指令直接翻译成一段汇编指令序列；实现简单、调试直观，代价是缺少 DAG 带来的系统性优化，需要自己管理指令选择中的所有细节。

各位同学可以根据自己的情况自行选择，不同的实现方案不影响最终的得分。

3.2.1 SelectionDAG

SelectionDAG 是 LLVM 中广泛用于指令选择的框架，已被 X86、NVPTX、MIPS、Hexagon 等多个后端采用。

传统的中端优化在基本块内以顺序结构（如 vector、queue 等）存储指令，而 SelectionDAG 将基本块转换为有向无环图（DAG）的形式进行存储。如图3.1 所示，该表示方法显式维护了操作数之间的数据流依赖关系，使编译器能够识别更多优化机会，例如自然地支持公共子表达式消除等经典优化。

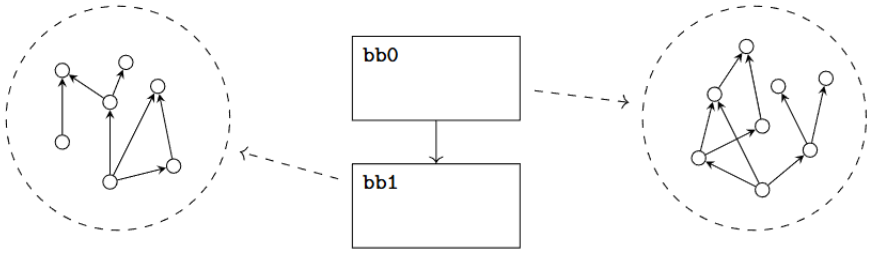


图 3.1: SelectionDAG 中基本块的 DAG 表示

采用 SelectionDAG 进行指令选择后，生成的目标代码结构更为规整，便于后续在代码生成阶段实施进一步的优化，从而获得更显著的性能提升。更多相关内容可参考教程 A Beginner's Guide to SelectionDAG。

3.2.2 指令选择示例

本小节的**符号规则**为：采用 vx 表示编号为 x 的虚拟寄存器。

访存指令 对于访存指令，具体可以分为以下三种情况：

1. 加载一个全局变量或者常量

   对于这种情况，同学们需要生成两条加载指令，首先在全局的地址标签中将其地址加载到寄存器中，之后再从该地址中加载出其实际的值。

```
%t1 = load i32* @a    ====>   adrp v2, a            # 获取页基址
                              add  v2, v2, :lo12:a # 获取页内偏移
                              ldr  v1, [v2]         # arm64
```

```
%t1 = load i32* @a    ====>   lui v1, %hi(a)
                              lw v2,%lo(a)(v1)     #riscv
```

2. 加载一个栈中的临时变量

   由于在 AllocaInst 指令中，已经为所有的局部变量申请了栈内空间，同学们只需要以 FP 为基址寄存器，维护以下每个局部变量的栈内偏移，并根据其栈内偏移生成一条加载指令即可。**对于 cpp-riscv 框架，处理思路为将 alloca 地址的结果保存到一个虚拟寄存器中，加载时直接根据该寄存器计算偏移即可。**

```
# 在 ARM64 中，FP 指代 x29 寄存器
%t1 = load i32* %t2   ====>   ldr v1, [fp, #offset_t2] # arm64
```

```
// 假设%t2 寄存器的值对应的虚拟寄存器是 v2
%t1 = load i32* %t2   ====>   lw v1, 0(v2) #riscv
```

3. 加载一个数组元素

   数组元素的地址存放在一个临时变量中，只需生成一条加载指令即可。**cpp-riscv 框架同上**。

```
%t1 = load i32* %t2   ====>   ldr v1, [v2] # arm64
```

   大家只需要模仿完成 StoreInst 的翻译即可。

内存分配指令 cpp-arm 框架代码中已经完成了对于 AllocaInst 的翻译，具体思路就是为指令的目的操作数在**栈**内分配空间，将其相对于 FP 寄存器的偏移存在符号表中。对于 cpp-riscv 框架，需要处理一下每条 alloca 指令的结果对应的地址并存放到任意一个虚拟寄存器中。

二元运算指令 对 ArithmeticInst 的翻译应该是最简单的, 按照下面示例进行即可。需要注意的一点是，中间代码中二元运算指令的两个操作数都可以是立即数，但这一点在汇编指令中是不被允许的。对这种情况，同学们需要根据汇编指令的规则，提前将其中的某个操作数加载到寄存器中。需要注意的是，当第二个源操作数是立即数时，其数值范围有一定限制。

```
%t3 = mul nsw i32 %t1, %t2   ====>   mul v3, v1, v2  # arm64
```

```
%t3 = mul nsw i32 %t1, %t2   ====>   mulw v3, v1, v2  #riscv
```

控制流指令

1. BrUncondInst

   对于 BrUncondInst，同学们只需要生成一条无条件跳转指令即可:

```
br label %B6   ====>   b .L6  # arm64
```

```
br label %B6   ====>   j .L6  #riscv
```

2. BrCondInst

   对于 BrCondInst，首先明确在中间代码中该指令一定位于 CmpInst 之后，对 CmpInst 的翻译比较简单，相信大家都能完成。对 BrCondInst，同学们首先需要在目标代码生成器中添加成员以记录前一条 CmpInst 的条件码，从而在遇到 BrCondInst 时生成对应的条件跳转指令跳转到 True Branch，之后需要生成一条无条件跳转指令跳转到 False Branch。

```
%t2 = icmp ne i32 %t1, 0                    cmp v1, #0
br i1 %t2, label %B3, label %B4   ====>     bne .L3
                                            b   .L4  # arm64
```

```
%t2 = icmp ne i32 %t1, 0                    bne v1, x0, .L3
br i1 %t2, label %B3, label %B4   ====>     j   .L4  #riscv

```

3. RetInst

   对于 RetInst 同学们需要考虑的情况比较多。首先，当函数有返回值时，我们需要生成 MOV 指令，将返回值保存在 R0 寄存器中；其次，我们需要生成 MOV 指令来恢复栈帧；如果该函数保存了被调用者保存寄存器，我们还需要生成 POP 指令恢复这些寄存器；最后再生成跳转指令来返回到 Caller。

```
ret i32 0   ====>   mov w0, #0
                    # 恢复 fp(x29), lr(x30) 并移动 sp
                    ldp x29, x30, [sp], #stack_size
                    ret    # arm64
```

   对于 cpp-riscv 框架，插入恢复 ra，s0 等寄存器以及回收栈空间的任务会在寄存器分配结束后进行，同学们可以自行探索一下为什么要这样做。

```
ret i32 0   ====>   li a0 0
                    ld ra, offset1(sp)
                    ld s0, offset2(sp)
                    ......
                    addi sp, sp, #stack_size
                    jr ra    #riscv
```

函数定义 在目标代码中，在函数开头需要进行一些准备工作。首先需要生成 PUSH 指令保存函数中修改的被调用者保存寄存器，之后生成 MOV 指令令 FP 寄存器指向新的栈底，之后需要生成 SUB 指令为局部变量分配栈内空间。分配栈空间时需要注意，一定要在完成寄存器分配后再确定实际的函数栈空间，因为有可能会有某些虚拟寄存器被溢出到栈中。一种思路是不在目标代码生成时插入 SUB 指令，而是在后续打印目标代码时直接将该条指令打印出来，因为在打印时已经可以获取到实际的栈内空间大小；另一种思路是先记录操作数还没有确定的指令，在指令的操作数确定后进行设置[1]。至此，就可以继续转换函数中对应的其他指令了。

```
define i32 @main(){...    ====>    main:
                                     # 压入 fp, lr 并分配栈空间
```

[1]翻译 RetInst 时，也可以采取相同的思路

```
                                    stp x29, x30, [sp, #-stack_size]!
                                    mov x29, sp   # arm64
```

对于 cpp-riscv 框架，由于传参使用到的寄存器一定是物理寄存器，为了方便后续的寄存器分配，我们需要先将这些物理寄存器挪到虚拟寄存器中，其余的思路与 arm 框架一致：

```
define i32 @foo(i32 %r0){...   ====>   foo:
                                    addi sp, sp, -#stack_size
                                    sd ra, offset1(sp)  # 寄存器分配后插入 sd
                                    sd s0, offset2(sp)
                                    ......
                                    add v0, a0, x0  # v0 = COPY a0  #riscv

```

函数调用指令 函数调用指令用于进行函数调用。对于含参函数，调用者需使用寄存器传递前若干个参数：在 **AArch64** 中，前 8 个整型参数根据数据宽度通过 **X0**–**X7**（64 位）或 **W0**–**W7**（32 位）传递，在这里注意区分 w 和 x 寄存器的区别；前 8 个浮点参数通过 **V0**–**V7**（具体为 **D0**–**D7** 或 **S0**–**S7**）传递；若参数个数超过可用寄存器，则需将额外参数存入调用者栈帧的参数区。之后，调用者生成跳转指令（如 **BL**）进入被调用函数。如果函数执行结果被用到，标量返回值会保存在 **X0/W0**（整型）或 **D0/S0**（浮点）中；被调用函数负责保存和恢复 **X19**–**X28** 及 **V8**–**V15** 等被调用者保存寄存器，并在入口和出口处维护 **FP (X29)** 和 **LR (X30)** 以及栈帧。

作为对比，**RISC-V** 的调用约定中，前 8 个整型参数通过 **a0**–**a7** 传递，前 8 个浮点参数通过 **fa0**–**fa7** 传递，多余参数同样通过栈传递。返回值位于 **a0**（整型）或 **fa0**（浮点）。调用时使用 **JAL** 或 **CALL** 指令，返回地址保存在 **ra** 寄存器中，调用者和被调用者分别负责相应寄存器的保存与恢复，且栈指针 **sp** 必须保持 16 字节对齐。

```
%t1 = call i32 @foo(i32 1)   ====>    mov w0, #1   # 参数 1 (32 位)
                                      bl foo
                                      mov v1, w0   # 此时 w0 存放返回值
```

```
%t1 = call i32 @foo(i32 1)   ====>   addi a0, x0, 1
                                     call foo
                                     mov v1, a0  #riscv
```

3.3 实现寄存器分配

寄存器分配是编译器的一个重要优化技术，通过将程序变量尽可能地分配到寄存器，从而提高程序执行速度。此处我们仅讲解线性扫描寄存器分配算法 [1]，遍历每个活跃区间 (Interval)，为其分配物理寄存器。其具体分为以下三个步骤。

3.3.1 活跃区间分析

在前一步的指令选择过程中，已经为所有临时变量分配了一个虚拟寄存器。在这一步需要为每个虚拟寄存器计算活跃区间，活跃区间相交的虚拟寄存器不能分配相同的物理寄存器。活跃区间的计算主要依赖活跃变量分析这一数据流分析方法，活跃变量分析的结果可以判断变量 x 在程序点 p 处是否活跃，通俗来讲，变量 x 在点 p 处活跃指的是变量 x 在点 p 处的值在点 p 或点 p 之后仍然会被用到。变量 x 编号最小和最大的两个活跃点便是其活跃区间的端点。这一步在课程中应该已经有所讲解，具体算法可参照龙书第二版 P391。

3.3.2 寄存器分配

得到活跃区间的信息，便可以进行寄存器分配了。下面具体介绍算法的具体流程。

算法主要涉及到了两个集合：

```
1 std::vector<Interval*> intervals;
2 std::vector<Interval*> active;
```

intervals 表示还未分配寄存器的活跃区间，其中所有的 interval 都按照开始位置进行递增排序；active 表示当前正在占用物理寄存器的活跃区间集合，其中所有的 interval 都按照结束位置进行递增排序。

算法遍历 intervals 列表，对遍历到的每一个活跃区间 i 都进行如下的处理：

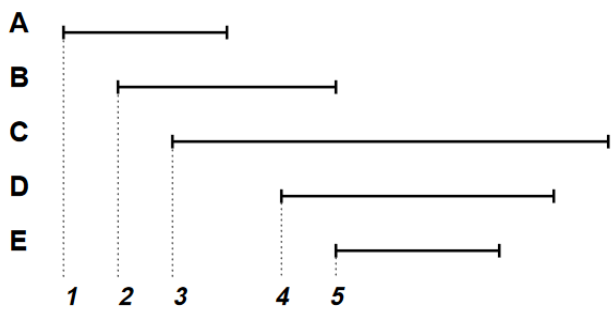
1. 遍历 active 列表，看该列表中是否存在结束时间早于区间 i 开始时间的 interval（即与活跃区间 i 不冲突），若有，则说明此时为其分配的物理寄存器可以回收，可以用于后续的分配，需要将其在 active 列表删除；

2. 判断 active 列表中 interval 的数目和可用的物理寄存器数目是否相等，

   (a) 若相等，则说明当前所有物理寄存器都被占用，需要进行寄存器溢出操作。具体为在 active 列表中最后一个 interval 和活跃区间 i 中选择一个 interval 将其溢出到栈中，选择策略就是看哪个活跃区间结束时间更晚，如果是活跃区间 i 的结束时间更晚，只需要置位其 spill 标志位即可，如果是 active 列表中的活跃区间结束时间更晚，需要置位其 spill 标志位，并将其占用的寄存器分配给区间 i，再将区间 i 插入到 active 列表中。

   (b) 若不相等，则说明当前有可用于分配的物理寄存器，为区间 i 分配物理寄存器之后，再按照活跃区间结束位置，将其插入到 active 列表中即可。

下面将给出一个寄存器分配的简单示例:



如上图所示，是一组 intervals。其中，左侧字母表示变量名，右侧线段表示变量的活跃区间。斜体数字表示线性扫描算法中的步骤。现在假设一共有 2 个物理寄存器 r1 和 r2。下表给出了一次迭代中对每个 interval 分配物理寄存器的过程。

	active intervals	available registers	spilled intervals	allocation results
初始	{}	{r1,r2}	{}	{}
step1	{A=r1}	{r2}	{}	{}
step2	{A=r1,B=r2}	{}	{}	{}
step3	{A=r1,B=r2}	{}	{C}	{}
step4	{B=r2,D=r1}	{}	{C}	{A=r1}
step5	{E=r2,D=r1}	{}	{C}	{A=r1,B=r2}
结束	{}	{r1,r2}	{C}	{A=r1,B=r2,D=r1,E=r2}

该算法在一次分配中执行分配决策 5 次（遍历每个活跃区间）。到步骤 2 结束时，active=<A，B>，与可用物理寄存器数目相等。在步骤 3 中，三个活跃区间重叠，并且已经没有空闲的物理寄存器，因此必须溢出一个变量。根据前面提到的处理方法，对比 active 列表中最后一个 interval B 和当前的 interval C 的结束时间，溢出结束时间更晚的 C。在步骤 4 中，发现 A 的结束时间早于当前 interval D 的开始时间，因此将 A 从 active 列表移出，使得寄存器 r1 可分配于 D。同理，在步骤 5 中，将 B 从 active 列表移出，使得寄存器 r2 可分配于 E。最终，这一轮分配后，C 是唯一溢出而未分配寄存器的变量。

3.3.3 生成溢出代码

在上一步寄存器分配结束之后，如果没有临时变量被溢出到栈内，那寄存器分配的工作就结束了，所有的临时变量都被存在了寄存器中；若有，就需要在操作该临时变量时插入对应的 LoadMInstruction 和 StoreMInstruction，其起到的实际效果就是将该临时变量的活跃区间进行切分，以便重新进行寄存器分配。这一步需要大家完善 LinearScan::spillAtInterval(Interval *interval) 函数。具体分为以下三个步骤：

1. 为其在栈内分配空间，获取当前在栈内相对 FP 的偏移；
2. 遍历其 USE 指令的列表，在 USE 指令前插入 LoadMInstruction，将其从栈内加载到目前的虚拟寄存器中;
3. 遍历其 DEF 指令的列表，在 DEF 指令后插入 StoreMInstruction，将其从目前的虚拟寄存器中存到栈内;

插入结束后，会迭代进行以上过程，重新计算活跃区间，进行寄存器分配，直至没有溢出情况出现。

下面给一个处理虚拟寄存器溢出到内存的简单例子。

假设有两个虚拟寄存器 vreg1 和 vreg2 需要溢出到内存中。这两个虚拟寄存器在函数中被定义（def）和使用（use）：

```
1    // 定义 vreg1
2    vreg1 = someCalculation();          //如 mov vreg1, #someValue
```

```
    // 使用 vreg1 和 vreg2
    result = someOtherCalculation(vreg1, vreg2);
```

处理溢出时，需要在对应虚拟寄存器被定义（define）后存储（store）它们的值到内存栈中，以及在对应虚拟寄存器被使用（use）前加载（load）它们的值。下面给出该例子完成溢出处理、重新分配寄存器并无新的溢出后的 Arm 汇编代码。注意：在处理溢出时添加的指令中仍然使用的是虚拟寄存器，需要重新进行寄存器分配并无溢出才能真正实现所有的物理寄存器分配。

```
    // 定义 vreg1
    mov r4, #someValue // someCalculation() 的结果
    str r4, [fp, #-4]  // 存储 vreg1 到内存中

    // 使用 vreg1 和 vreg2
    ldr r5, [fp, #-4]  // 从内存中加载 vreg1
    mov r0, r5         // 将 vreg1 的值移至 r0
    ldr r6, [fp, #-8]  // 从内存中加载 vreg2
    mov r1, r6         // 将 vreg2 的值移至 r1
    bl _someOtherCalculation // 调用 someOtherCalculation
```

3.4 更好的寄存器分配算法流程

对于 cpp-riscv 框架，我们采用类似 clang 编译器的算法流程，并且将其封装为了多后端通用的寄存器分配框架。**由于在要求中已经说明可以采用平凡的寄存器分配，所以大家在寄存器分配这一步骤中不需要在已有框架上进行，可以重新编写，所以这里只做大致介绍，供有兴趣或想参加编译比赛的同学了解。**同学们可以在LLVM Developers' Meeting了解更多。

3.4.1 Phi 指令删除

编译器通常在 LLVM-IR 上转化为 SSA 形式后进行非常多的优化，之后通过指令选择转换为 Machine-IR(仍保留 SSA 性质)，然后进行一些针对体系结构的优化，在此之后进行 phi 指令的消除，此时不再保留 SSA 形式，开始寄存器分配。Phi 指令消除的具体步骤可以参考之前在代码优化指导书中提到的 SSA book。

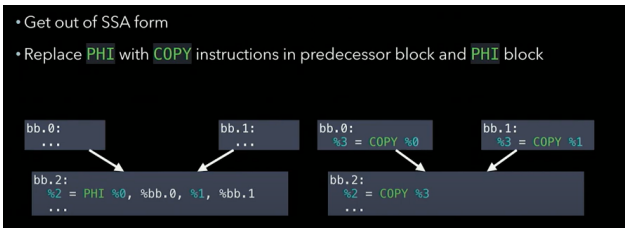


图 3.2: Phi 指令删除示例

3.4.2 分段的活跃区间计算

寄存器分配的第一件事是需要进行活跃变量分析和活跃区间计算，框架已经完成了这一部分的代码，同学们如果有兴趣可以自行查找资料，这里仅给出一个示例：

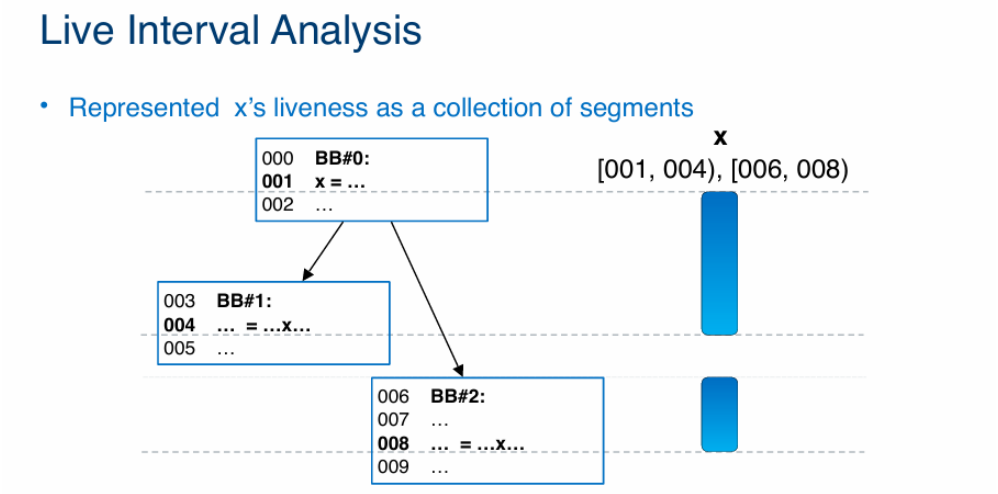


图 3.3: **活跃区间计算示例**

3.4.3 Register Coalesce

之前的 Phi 指令消除时会产生很多的 COPY 语句，但实际上如果 COPY 前后的两个寄存器活跃区间不冲突，可以将其合并为一个寄存器，从而减少一条 COPY 语句，下面给出一个示例：

图 3.4: Register Coalesce **示例，由于虚拟寄存器**%5, %6，%202 **的活跃区间并不相交，所以可以将这三个虚拟寄存器合并为一个。**

3.4.4 预处理已经存在的物理寄存器

同学们在阅读完指令选择的教程后应该注意到我们的 Machine-IR 中不仅有虚拟寄存器，还存在相当一部分的物理寄存器，所以我们在分配寄存器的时候，要先考虑虚拟寄存器区间是否与已存在的

物理寄存器区间冲突，这一步可以通过预处理实现，将之前已经算好的物理寄存器的活跃区间预先做一个标记，分配时检查是否与已有的物理寄存器区间冲突即可。

3.4.5 Greedy 分配

同学们可以参考该教程Greedy 寄存器分配算法，我们的框架并没有实现 greedy 分配中的 split 操作，如果有兴趣的同学可以选择自行实现，如果成功实现可以额外加分。（不能超过代码优化额外加分的上限）

3.4.6 图着色分配

有兴趣的同学可以可以参考虎书 11 节或图着色寄存器分配算法选择自行实现。

4 多后端框架实验流程说明

本实验提供 SysY 多后端编译器框架，详细的目录结构与各个阶段的职责，请同学们参考仓库中的 `backend/README.md`，其中对后端流水线（指令选择、帧降低、寄存器分配、栈降低）以及 RISC-V / AArch64 两个 Target 的实现入口都有比较完整的说明。

需要特别注意的是：**后端目标架构的选择是通过编译命令行参数控制的**，而不是通过切换不同的工程或 Makefile。当前框架中，核心可执行文件统一为 `bin/compiler`，通过是否指定 `-march` 以及其取值来选择后端：

- **RISC-V 后端**：省略 `-march` 参数（默认即为 RISC-V），或使用框架约定的 RISC-V 相关参数；
- **AArch64 后端**：在命令行中显式添加 `-march aarch64`，即可选择 AArch64 后端。

4.1 编译与测试流程

首先使用 Make 进行整体编译，

```
make -j    # 如果你没有修改词法/语法文件 (parser/lexer)，直接 make -j 即可
```

假设根目录下有一个示例程序 `example.sy`，可以分别使用 RISC-V 和 AArch64 后端生成汇编并运行：

RISC-V 后端示例 （默认后端，无 `-march`）

```
# 生成 RISC-V 汇编
./bin/compiler -S -o example.riscv.s example.sy -O1

# 使用 RISC-V 交叉编译工具链生成可执行文件
riscv64-unknown-elf-gcc example.riscv.s -c -o tmp.o
riscv64-unknown-elf-gcc -static tmp.o -L./lib -lsysy_riscv

./a.out  # 运行程序
echo $?  # 查看 main 函数的返回值
```

AArch64 后端示例 （通过 `-march aarch64` 选择 ARMv8 后端）

```
# 生成 AArch64 汇编
./bin/compiler -S -o example.aarch64.s example.sy -O1 -march aarch64

# 使用 AArch64 交叉编译工具链生成可执行文件
aarch64-linux-gnu-gcc example.aarch64.s -c -o tmp.o
aarch64-linux-gnu-gcc -static tmp.o -L./lib -lsysy_aarch

./a.out  # 运行程序 (在支持 AArch64 的环境或通过 qemu-aarch64)
echo $?  # 查看 main 函数的返回值
```

框架同时提供了一个统一的测试脚本 `test.py`，用于批量运行 SysY 测试用例：

```
# 测试 RISC-V 后端
python3 test.py --group Basic    --stage riscv --opt [1, 2, 3]
# 测试 AArch64 后端
python3 test.py --group Advanced --stage arm   --opt [1, 2, 3]
```

4.2 ARM 示例（仅作示意）

下面给出一段基于 Armv8-a 结构 (同时兼容 32 位、64 位 Arm) 而不是 AArch64/ARMv8 的示例，用于说明“中间代码 → 目标代码”以及“寄存器分配前后”的形态变化。**注意：本示例汇编代码仅用于演示，不要求与当前框架实际生成的指令序列完全一致。**

以如下 SysY 程序为例：

```
int main()
{
    int a;
    int b;
    int min;
    a = 1 + 2 + 3;
    b = 2 + 3 + 4;
    if (a < b)
        min = a;
    else
        min = b;
    return min;
}
```

在还未进行寄存器分配、暂时使用“虚拟寄存器”表示的时候，目标代码可能类似如下（仅示意）：

```
    .arch armv8-a
```

```
    .arch_extension crc
    .arm
    .global main
    .type main , %function
main:
.L17:
    push {fp}
    mov fp, sp
    sub sp, sp, #12
    ldr v26, =1
    add v4, v26, #2
    add v5, v4, #3
    str v5, [fp, #-12]
    ldr v27, =2
    add v7, v27, #3
    add v8, v7, #4
    str v8, [fp, #-8]
    ldr v9, [fp, #-12]
    ldr v10, [fp, #-8]
    cmp v9, v10
    blt .L21
    b .L24
.L21:
    ldr v13, [fp, #-12]
    str v13, [fp, #-4]
    b .L23
.L22:
    ldr v15, [fp, #-8]
    str v15, [fp, #-4]
    b .L23
.L23:
    ldr v16, [fp, #-4]
    mov r0, v16
    add sp, sp, #12
    pop {fp}
    bx lr
.L24:
    b .L22
```

经过线性扫描等寄存器分配算法，将虚拟寄存器映射到具体物理寄存器后，代码形态可能变为：

```
    .arch armv8-a
    .arch_extension crc
```

```
    .arm
    .global main
    .type main , %function
main:
.L17:
    push {r4, r5, fp}
    mov fp, sp
    sub sp, sp, #12
    ldr r4, =1
    add r4, r4, #2
    add r4, r4, #3
    str r4, [fp, #-12]
    ldr r4, =2
    add r4, r4, #3
    add r4, r4, #4
    str r4, [fp, #-8]
    ldr r4, [fp, #-12]
    ldr r5, [fp, #-8]
    cmp r4, r5
    blt .L21
    b .L24
.L21:
    ldr r5, [fp, #-12]
    str r5, [fp, #-4]
    b .L23
.L22:
    ldr r5, [fp, #-8]
    str r5, [fp, #-4]
    b .L23
.L23:
    ldr r5, [fp, #-4]
    mov r0, r5
    add sp, sp, #12
    pop {r4, r5, fp}
    bx lr
.L24:
    b .L22
```

再次强调：**上述 ARM 汇编只是帮助理解后端各阶段职责的示例代码，实际框架在不同版本、不同优化级别下生成的汇编序列可能与示例有差异，同学们不需要做到逐条指令完全一致，只需满足语义正确、遵循调用约定和框架接口要求即可。**

5 评分标准

5.1 基本要求（满分 7 分）

基本要求得到满分分数需正确实现如下 SysY 特性：

1. 数据类型：int
2. 变量声明、常量声明，常量、变量的初始化
3. 语句：赋值（=）、表达式语句、语句块、if、while、return
4. 表达式：算术运算（+、-、*、/、%，其中 +、-都可以是单目运算符）、关系运算（==，>，<，>=，<=，!=）和逻辑运算（&&（与）、||（或）、!（非））
5. 函数、语句块（函数声明、函数调用；变量、常量作用域（函数、语句块中变量、常量声明的处理），break、continue 语句）
6. 注释
7. 输入输出（实现连接 SysY 运行时库，参见文档《SysY 运行时库》）

具体评分标准：基本要求一共有 100 个测试用例，其中 100 个测试用例平分 6 分，即每个测试用例的分值为 0.06 分。**如果你通过了所有的基本要求测试用例，额外再获得 1 分**。如果你线下检查时未能成功回答出助教的问题，我们会根据你的回答情况在测试用例得分的基础上扣除一定分数。**需要开启所有得到分数的中端优化，否则后端的实现并不算完整，会扣除一定的分数**。

5.2 进阶要求（满分 2 分）

进阶要求得到满分分数需正确实现如下 SysY 特性：

1. 数组：数组（一维、二维、…）的声明和数组元素访问
2. 浮点数：浮点数常量识别、变量声明、存储、运算

具体评分标准：进阶要求一共有 100 个测试用例，其中 100 个测试用例平分 1 分，即每个测试用例的分值为 0.01 分，对于该项分数我们会以 0.05 为精度向下取整（即如果你的得分为 0.99 分，我们会将你的分数记为 0.95 分）。**如果你通过了 90% 以上的进阶要求测试用例，额外获得 0.5 分，如果你通过了所有的进阶要求测试用例，额外再获得 0.5 分**。如果你线下检查时未能成功回答出助教的问题，我们会根据你的回答情况在测试用例得分的基础上扣除一定分数。**需要开启所有得到分数的中端优化，否则后端的实现并不算完整，会扣除一定的分数**。

参考文献

[1] MASSIMILIANO POLETTO. Linear scan register allocation. https://c9x.me/compile/bib/linearscan.pdf.