11 指令选择-ARM

徐辉，xuh@fudan.edu.cn

本章学习目标：

- ⋆⋆ 掌握 ARMv8-A（AArch64）指令集的基础指令
- ⋆⋆⋆ 掌握从 LLVM IR 到目标汇编代码的指令选择方法及其建模优化技术

11.1 ARMv8-A 指令集

本章介绍如何将 LLVM IR 翻译为 AArch64 汇编代码，目标体系结构为 ARMv8-A。我们首先讨论单条 LLVM IR 指令到 AArch64 指令的基本翻译方法，然后进一步介绍针对整个代码块的指令选择问题及其优化方法。为简化讨论，本章暂不考虑具体物理寄存器的分配问题，而统一使用虚拟寄存器表示操作数。其中，`w0-wn` 表示 32 位寄存器，`x0-xn` 表示 64 位寄存器。

ARMv8-A 属于典型的精简指令集（RISC）体系结构，其主要特点之一是访存操作与算术运算相互分离：数据必须先通过加载指令从内存读入寄存器，随后才能参与运算。代码 11.1 给出了一个简单的 Hello World 汇编程序。该程序首先使用 `str` 指令将返回地址寄存器 `x30` 保存到栈中；随后通过 `adrp` 与 `add` 指令构造字符串`"Hello World!"` 的地址。其中，`adrp` 用于获取目标符号所在内存页的基地址，`add` 进一步补充页内偏移。最后，程序通过 `bl` 指令调用 `puts` 函数输出字符串，并在返回前恢复寄存器 `x30`。

```
    .text                    ; 代码段
    .global main             ; 导出符号 main

main:
    str     x30, [sp, -16]! ; 保存返回地址，并为栈帧分配16字节空间
    adrp    x0, .LC0         ; 将字符串所在页的基地址加载到 x0
    add     x0, x0, :lo12:.LC0 ; 加上页内偏移，得到字符串完整地址
    bl      puts             ; 调用 puts(x0)
    ldr     x30, [sp], 16    ; 恢复返回地址，并回收栈空间
    ret                      ; 返回调用者

.LC0:
    .string "Hello World!"   ; 字符串常量
```

代码 11.1: ARM-v8A 指令示例：Hello World 程序

不同平台可能采用不同的汇编语法，本章统一采用 Linux/LLVM/GCC 工具链常见的 GNU 汇编语法。对于大多数数据处理指令，其操作数一般采用：

```
opcode destination, source1, source2
```

的形式，即结果寄存器通常写在最前面。例如：`add x0, x1, x2` 表示 `x0 = x1 + x2`。

11.1.1 寻址模式

在 ARMv8-A 架构中，变量可能位于不同的存储区域，因此需要采用不同的寻址方式。根据变量的作用域和存储位置，寻址通常可以分为全局变量寻址和局部变量寻址两类。

全局变量通常位于数据段或只读段，其地址在程序运行期间相对固定。由于 AArch64 指令长度固定为 32 位，无法在单条指令中直接编码完整的 64 位地址，因此通常采用“页基址 + 页内偏移”的方式构造全局地址。一般先通过 `adrp` 指令获取目标符号所在内存页的基地址（4KB 对齐），再通过 `add` 或 `ldr` 补充低 12 位偏移量。

```
adrp x0, g                  ; 加载全局变量 g 所在页的基地址
ldr  w1, [x0, :lo12:g]      ; 读取全局变量 g; :lo12:g表示符号g在页内的低12位偏移量
```

局部变量通常分配在栈空间中，并通过栈指针寄存器 `sp` 访问。ARMv8-A 支持多种灵活的访存寻址模式，主要包括以下几种：

- **立即偏移寻址模式**：使用基地址寄存器和立即数偏移量计算目标地址。例如：

  ```
  ldr x2, [x1]          ; 加载地址 x1 中的数据到 x2
  ldr x2, [x1, #10]     ; 加载地址 x1+10 中的数据到 x2
  ```

- **寄存器偏移寻址模式**：偏移量由另一个寄存器提供，可用于数组访问等场景。例如：

  ```
  ldr x2, [x1, x0]              ; 加载地址 x1+x0 中的数据
  ldr x2, [x1, x0, lsl #3]      ; 加载地址 x1+x0*8 中的数据
  ```

  其中，`lsl #3` 表示将寄存器 `x0` 左移 3 位，相当于乘以 8，因此该寻址模式常用于访问 8 字节元素（如 `i64` 或指针类型）构成的数组。虽然左移操作在硬件上可能发生整数回绕，但若数组下标已经大到导致地址计算溢出，则对应数组所需的内存规模通常已经超过机器实际可分配的地址空间，因此这种情况不会出现在正常的数组访问中。

- **预索引寻址模式**：在访存之前先更新基地址寄存器。例如：

  ```
  ldr x2, [x1, #10]! ; 先将x1更新为x1+10，再读取该地址中的数据到x2
  ```

  这种形式常用于栈帧调整。

- **后索引寻址模式**：先完成访存，再更新基地址寄存器。例如：

  ```
  ldr x2, [x1], #10 ; 先读取地址x1中的数据到x2，随后再将x1更新为x1+10
  ```

- **栈寻址模式**：通过栈指针寄存器 `sp` 访问局部变量。例如：

  ```
  ldr x2, [sp, #8] ; 读取地址sp+8中的数据到x2。
  ```

11.1.2 立即数支持

由于 AArch64 指令采用 32 位定长编码，因此一条指令中能够用于表示立即数的位数较为有限。不同类型的指令采用不同的立即数编码方式，其中算术运算指令和逻辑运算指令的支持方式并不相同。

大多数算术数据处理指令（如 `add`、`sub`）支持 12 位无符号立即数，即：

$$0 \leq x < 2^{12}$$

此外，这类立即数还支持额外左移 12 位，即：

$$x \times 2^{12}$$

因此，除了 `0-4095` 外，还可以直接表示如 `4096`、`8192` 等数值。例如：

```
add x0, x1, #100
add x0, x1, #4096
```

当立即数无法直接编码时，通常需要通过多条指令构造完整常数。AArch64 提供了 `movz`、`movk` 等指令用于分段构造 64 位立即数。其中：

- `movz`：将寄存器其它位清零，并写入 16 位立即数；
- `movk`：保持寄存器原有位不变，仅修改指定的 16 位字段。

例如，立即数 65539（即 `0x00010003`）可以通过如下方式构造：

```
movz x8, #3                 ; x8 = 0x0000000000000003
movk x8, #1, lsl #16        ; x8 = 0x0000000000010003
```

代码 11.2: ARM-v8A 指令：立即数示例

其中，`lsl #16` 表示将立即数左移 16 位后写入寄存器对应位置，即修改寄存器的 16-31 位。

逻辑运算指令（如 `and`、`orr`、`eor`）采用与算术指令不同的立即数编码方式。它并不是简单支持固定宽度的整数，而是支持一类特殊的位掩码模式（bitmask immediate）。这种设计来源于位运算的常见使用场景：程序通常更关注“哪些位为 1”，而不是立即数本身的数值大小。

例如，下列连续 1 位掩码可以直接编码：

```
and x0, x1, #0xFF           ; 保留低8位
and x0, x1, #0xFFFF         ; 保留低16位
```

上述指令常用于整数截断、类型转换以及提取某些字段。

此外，ARMv8-A 还支持周期性重复的位模式。例如：

```
orr x0, xzr, #0xAAAAAAAAAAAAAAAA
```

该指令会生成如下二进制模式：

```
1010101010101010...
```

需要注意的是，这里的 `0xAAAAAAAAAAAAAAAA` 并不是作为普通 64 位整数直接编码到指令中，而是属于 ARMv8-A 逻辑立即数支持的一类特殊位掩码模式。ARM 会使用专门的压缩编码方式描述位模式的重复结构，并由硬件自动生成完整的 64 位掩码。例如，`0xAAAAAAAAAAAAAAAA` 和 `0x3333333333333333` 都属于这种可直接编码的周期性位模式。这种模式主要用于高效支持位掩码生成、SIMD 向量计算以及各类底层位运算优化。

“‘latex

11.1.3 主要指令

下面以单条 LLVM IR 指令的翻译为例，介绍常用的 ARMv8-A（AArch64）指令，见表 11.1 。实际 ARMv8-A 架构包含数百条指令，并提供了丰富的寻址模式与硬件机制。本节仅介绍编译器后端代码生成过程中最常见的一部分指令。如需进一步了解完整指令系统，可参考 ARM 官方手册 [1]。

表 11.1: LLVM IR 及其对应的 ARM-v8A 指令

IR 指令	ARM-v8A 指令	说明
`%a = alloca i32`	`sub sp, sp, 16`	为局部变量分配栈空间；AArch64 要求栈指针保持 16 字节对齐
`store i32 %0, ptr %a`	`str w0, [sp, 12]`	将寄存器中的 32 位整数保存到栈空间
`store i32 1, ptr %a`	`mov w0, #1` `str w0, [sp, 12]`	先构造立即数，再写入内存；`str` 不支持立即数操作数
`%a0 = load i32, ptr %a`	`ldr w0, [sp, 12]`	将栈空间中的 32 位整数加载到寄存器
`%g0 = load i32, ptr @g`	`adrp x8, g` `ldr w0, [x8, :lo12:g]`	先加载全局变量所在内存页地址，再加低 12 位偏移访问全局变量
`%r = add i32 %a, %b`	`add w0, w1, w2`	两个寄存器中的整数相加，结果保存到 `w0`
`%r = add i32 %a, 4095`	`add w0, w1, #4095`	算术指令支持 12 位立即数，以及左移 12 位后的形式
`%r = sub i32 %a, %b`	`sub w0, w1, w2`	两个寄存器中的整数相减；也支持立即数减法
`%r = mul i32 %a, %b`	`mul w0, w1, w2`	整数乘法；不支持立即数操作数
`%r = sdiv i32 %a, %b`	`sdiv w0, w1, w2`	有符号整数除法；不支持立即数操作数
`%r = icmp sgt i32 %a, %b` `br i1 %r, label %bb1, label %bb2`	`cmp w0, w1` `b.le .LBB2`	比较结果会更新条件标志位（NZCV），随后执行条件跳转
`%r = xor i32 %a, %b`	`eor w0, w1, w2`	按位异或运算；支持逻辑立即数
`%r = and i32 %a, %b`	`and w0, w1, w2`	按位与运算；支持逻辑立即数
`%r = or i32 %a, %b`	`orr w0, w1, w2`	按位或运算；支持逻辑立即数
`call void @foo()`	`bl foo`	函数调用，并将返回地址保存到 `x30` 寄存器
`ret i32 %r`	`ldr x30, [sp], 16` `ret`	从栈中恢复返回地址寄存器 `x30`，还原栈顶指针并返回

本节中的映射关系主要用于帮助理解 LLVM IR 与目标汇编之间的基本对应关系。需要注意的是，LLVM IR 中的单条指令通常并不一定严格对应一条 ARM 汇编指令，可能存在“一对多”或“多对一”的情况。

一方面，一条 IR 指令可能需要多条 ARM 指令共同完成。例如，ARMv8-A 中的许多运算指令并不支持立即数操作数，而较大的立即数通常也无法直接编码到单条指令中，因此编译器需要先构造常数，再执行对应运算。例如：

```
mul w0, w1, w2
sdiv w0, w1, w2
```

均只能使用寄存器作为输入操作数。

另一方面，一条 ARM 指令也可能同时对应多条 IR 指令。例如，ARMv8-A 提供了乘加指令：

```
madd w0, w1, w2, w3
```

其语义为 `w0 = w1 * w2 + w3`，因此可以同时对应 LLVM IR 中的乘法与加法两条指令。编译器还会主动利用 ARMv8-A 提供的特殊指令与寻址模式减少指令数量。例如，乘以 2 的幂通常会被优化为移位操作：

```
lsl w0, w1, #3      ; 等价于 w0 = w1 * 8
```

类似地，数组访问中的地址计算也常使用带缩放的寄存器偏移寻址模式：

```
ldr x0, [x1, x2, lsl #3]
```

其中，`x1` 为数组基地址，`x2` 为数组下标，`lsl #3` 表示元素大小为 8 字节。

11.2 消除 phi 指令

LLVM IR 在转换为 SSA（Static Single Assignment）形式后，会引入大量 phi 指令用于表示控制流汇合点上的变量选择。然而，ARM 等目标体系结构并不存在与 phi 指令直接对应的机器指令，因此在代码生成阶段必须消除 phi 指令。以代码 11.3 为例，可以通过两种方式完成 phi 指令的消除。

```
bb1:
%r1 = icmp eq i32 %a1, 0
; 方式一: store i32 %a1, ptr %a
; 方式二: %a3 = %a1
br i1 %r1, label %bb2, label %bb3

bb2:
%a2 = add i32 %a1, %b1
; 方式一: store i32 %a2, ptr %a
; 方式二: %a3 = %a2
br label %bb3

bb3:
%a3 = phi i32 [%a1, %bb1], [%a2, %bb2]
; 方式一: %a3 = load i32, ptr %a
%r1 = add i32 %a3, %b1
```

代码 11.3: LLVM IR 代码：消除 phi 指令的例子

方式一：使用 store-load 替换 phi ：该方法通过内存中转的方式实现 phi 指令的语义。在 phi 指令所有前驱基本块的跳转指令之前插入 store 指令，将对应的源操作数写入同一内存位置；随后在包含 phi 指令的基本块开头使用 load 指令读取该值，并以读取结果替代原有 phi 指令。

以代码 11.3 为例，在基本块 %bb1 和 %bb2 中分别插入：

```
store i32 %a1, ptr %a
store i32 %a2, ptr %a
```

同时将基本块 %bb3 中的 phi 指令替换为：

```
%a3 = load i32, ptr %a
```

这种方式与现有 LLVM IR 兼容可以直接使用 LLVM 解释器执行。缺点是会引入额外的访存开销。

方式二：使用伪赋值指令替换 phi ：在每个前驱基本块中直接对 phi 指令的目标寄存器进行赋值。具体而言，可将：

```
%a3 = phi i32 [%a1, %bb1], [%a2, %bb2]
```

转换为在前驱基本块中的伪赋值操作：

```
; bb1
%a3 = %a1
; bb2
%a3 = %a2
```

该方法本质上模拟了后端汇编中的寄存器移动操作。由于数据始终保存在寄存器中，因此能够避免额外的访存开销。缺点是 LLVM IR 本身并不支持这种“寄存器直接赋值”的语法形式，因此这种修改后的 IR 无法被标准 LLVM 解释器直接执行。

11.3 指令选择问题

通过前面的介绍，我们可以较为直接地为单条 LLVM IR 指令找到对应的 ARM 汇编翻译方式。例如，IR 中的加法、减法或乘法指令通常都能映射为对应的 ARM 算术指令。然而，仅采用“单条 IR 指令对应单条汇编指令”的翻译策略通常并不是最优的，因为现代处理器提供了大量能够同时完成多个操作的复合指令。如果忽略这些复合指令的使用，生成的目标代码往往会包含更多指令数量和额外的寄存器读写，从而影响程序执行效率。

例如，ARM-v8A 提供了乘法累加和乘法减法等复合算术指令，它们能够在一次指令执行过程中同时完成乘法与加法（或减法）运算。例如：

```
%t1 = mul i32 %a, %b
%t2 = add i32 %t1, %c
```

可以直接翻译为：

```
madd w0, w1, w2, w3
```

而无需先生成：

```
mul w4, w1, w2
add w0, w4, w3
```

上述过程体现的正是指令选择问题：即如何从 IR 表达的计算语义中，选择最适合目标体系结构的机器指令组合，使生成代码在指令数目和执行效率等方面达到更优效果。从更一般的角度来看，指令选择并不是简单的“翻译”问题，而是一个“模式匹配”问题。编译器需要识别 IR 中能够被目标机器复杂指令覆盖的计算模式，并使用代价更低的机器指令进行替换。例如：

- 将“乘法 + 加法”识别为 `madd`
- 将“乘法 + 减法”识别为 `msub`
- 将数组地址计算识别为 ARM 的地址偏移寻址模式

因此，高质量的指令选择算法往往需要在 IR 上进行模式分析与匹配，以最大程度利用目标体系结构提供的复杂指令能力。为了降低问题复杂度，下面我们将重点讨论单个基本块上的指令选择问题。由于函数控制流可以被划分为多个相互独立的基本块，因此编译器通常可以分别对每个基本块进行指令选择。

11.3.1 指令选择图

为了系统化地研究单个基本块上的指令选择问题，我们通常将基本块中的 LLVM IR 表示为一种特殊的有向图结构，从而将“生成目标机器指令”的过程转化为图上的模式匹配与覆盖问题。

定义 1（指令选择图）**.** 指令选择图（Instruction Selection Graph）是一个有向无环图（Directed Acyclic Graph, DAG），图中包含两类节点：

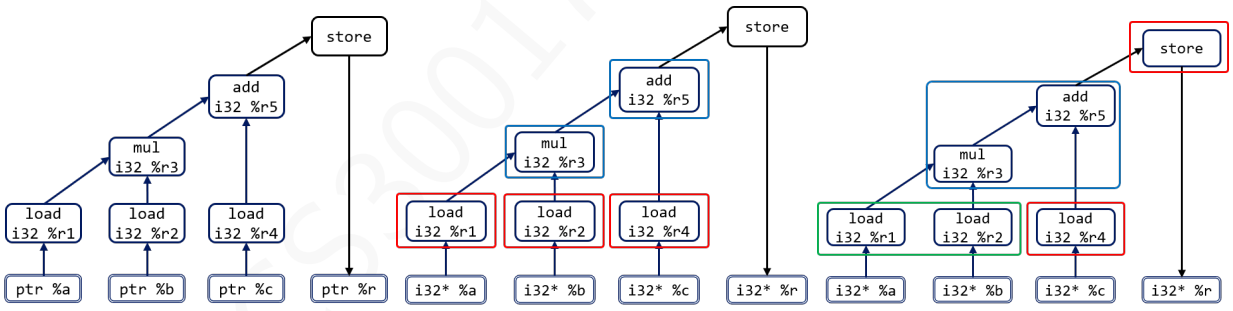
- **指令节点**：表示 IR 中的运算操作，例如 `add`、`mul`、`load` 等；
- **数据存储节点**：表示变量、寄存器或内存中的数据值。

图中的有向边表示数据依赖关系，即某条指令执行时需要使用的输入操作数。

由于 LLVM IR 处于 SSA 形式，每个虚拟寄存器只会被定义一次，因此普通算术数据依赖天然构成有向无环图。以代码 11.4 为例，该代码首先从内存中读取变量 `a` 和 `b`，随后执行乘法运算；接着读取变量 `c`，并将其与乘法结果相加；最后将结果写回内存。根据这些数据依赖关系，可以构建出图 11.1a 所示的指令选择图。若某条指令的输出被另一条指令使用，则在两者之间建立一条有向边。只要最终生成的机器指令序列满足该 DAG 的拓扑顺序，就能够保证程序语义正确。

```
%r1 = load i32 %a;
%r2 = load i32 %b;
%r3 = mul i32 %r1, %r2;
%r4 = load i32 %c;
%r5 = add i32 %r3, %r4;
store i32 %r5, %r;
```

代码 11.4: LLVM IR 代码



(a) 指令选择图　(b) 指令选择方案 1　(c) 指令选择方案 2

图 11.1: 指令选择问题举例

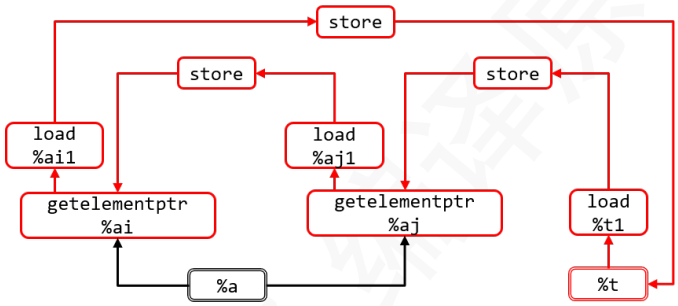
值得注意的是，当代码块中出现 `store` 等涉及内存写入的指令时，不能简单地仅依据寄存器数据依赖将整个基本块构造成一个指令选择 DAG。虽然 SSA 形式下的普通算术数据依赖天然构成无环图，但访存操作之间还存在额外的内存依赖关系。在某些情况下，这些内存依赖可能破坏原有 DAG 结构，从而无法仅通过普通数据流图正确表达程序语义。若忽略这些内存依赖关系，则后续拓扑排序与指令重排过程中可能生成违反原始 LLVM IR 语义的代码。

以代码11.5 为例，该代码实现了三个内存位置之间的数据交换操作，其语义高度依赖于 `store` 指令之间的执行顺序。若任意改变这些内存访问顺序，都可能导致读取到被覆盖后的值，从而破坏程序语义。因此，如果直接为整个基本块构建图，则不能有效表达内存读写之间的顺序约束，从而在进行拓扑排序时，可能得到错误的程序。

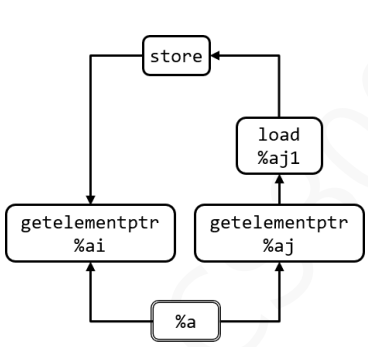
```
%ai = getelementptr [10 x i32], [10 x i32]* %a, i32 0, i32 i
%aj = getelementptr [10 x i32], [10 x i32]* %a, i32 0, i32 j
%aj1 = load i32, ptr %aj
store i32 %aj1, ptr %ai
%t1 = load i32, ptr %t
store i32 %t1, ptr %aj
%ai1 = load i32, ptr %ai
store i32 %ai1, ptr %t
```

代码 11.5: LLVM IR 代码：内存同步问题

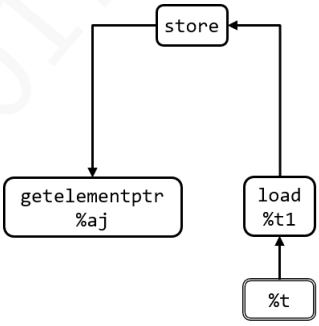
为解决这一问题，可以将 `store` 等具有副作用的指令作为边界，对基本块进行划分，将整个基本块拆分为多个具有顺序关系的指令选择子图。例如，图 11.2a可以拆分为图11.2b、11.2c和11.2d。



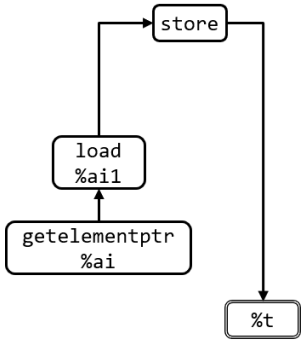
(a) 直接为代码 11.5 绘制指令选择图



(b) 指令选择子图 1



(c) 指令选择子图 2



(d) 指令选择子图 3

图 11.2: 代码块涉及内存同步问题时的指令选择子图分割

11.3.2 图覆盖问题

通过指令选择图，我们可以将指令选择问题转化为图覆盖问题，即如何使用目标机器指令覆盖指令选择图中的所有节点，并使生成代码的代价最小。这里的代价通常包括指令数量和执行时间。

以图 11.1a 为例，其至少存在图 11.1b 和图 11.1c 两种铺树方案，对应的汇编代码分别如代码 11.6 和代码 11.7 所示。

```
ldr w1, [sp, .a]
ldr w2, [sp, .b]
ldr w3, [sp, .c]
mul w3, w1, w2
add w5, w3, w4
str w5, [sp, .r]
```

代码 11.6: 图 11.1b 对应的汇编代码

```
ldp w1, w2, [sp, .a]
ldr w3, [sp, .c]
madd w5, w1, w2, w4
str w5, [sp, .r]
```

代码 11.7: 图 11.1c 对应的汇编代码

其中，方案一采用逐条指令翻译方式，而方案二使用了 ARM-v8A 的复合指令：

- `ldp` 同时完成两个连续变量加载；
- `madd` 同时完成乘法与加法。

因此，方案二的指令数量更少，寄存器使用也更少。若进一步假设 `ldr` 与 `ldp`、`mul` 与 `madd` 的执行代价相同，则图 11.1c 对应的铺树方案更优。

一般而言，铺树问题属于 NP-hard 问题，常见求解方法包括贪心算法与动态规划。其中，一种经典的贪心策略称为 Maximal Munch，其基本思想是：从 DAG 的根节点开始，每一步优先选择能够匹配且覆盖节点数最多的指令模式，并递归处理剩余未覆盖部分。例如，当编译器发现：

```
%r3 = mul i32 %r1, %r2
%r5 = add i32 %r3, %4
```

时，会优先匹配覆盖范围更大的 `madd` 模式，而不是分别选择 `mul` 与 `add` 两个较小模式。Maximal Munch 实现简单、效率较高，因此被许多编译器采用。不过，由于其属于局部贪心策略，因此并不一定能够得到全局最优解。关于其具体实现细节与优化效果，同学们可以进一步自行探索。

参考文献

[1] Arm Architecture Reference Manual for A-profile architecture. `https://developer.arm.com/documentation/ddi0487/latest/`, 2023.

练习

1) 将下列 IR 代码翻译为 ARMv8-A 汇编代码。

```
define i32 @collatz(i32 %x0) {
bb0:
    br label %bb1
bb1:
    %x1 = phi i32 [ %x0, %bb0 ], [ %x6, %bb5 ]
    %r0 = icmp ne i32 %x1, 1
    br i1 %r0, label %bb2, label %bb6
bb2:
    %x2 = srem i32 %x1, 2
    %r1 = icmp eq i32 %x2, 0
    br i1 %r1, label %bb3, label %bb4
bb3:
    %x3 = sdiv i32 %x1, 2
    br label %bb5
bb4:
    %x4 = mul i32 %x1, 3
    %x5 = add i32 %x4, 1
    br label %bb5
bb5:
    %x6 = phi i32 [ %x3, %bb3 ], [ %x5, %bb4 ]
    br label %bb1
bb6:
    ret i32 %x1
}
```

代码 11.8: LLVM IR 代码