11 指令选择

徐辉，xuh@fudan.edu.cn

本章学习目标：

- 掌握基础的 AArch64 指令和函数调用规约
- 掌握指令选择图和解法

本章学习如何将 IR 代码翻译为 AArch64 指令集的汇编代码，目标指令集版本是 ARM-v8A。我们先介绍单条 IR 指令对应的 AArch64 指令，然后介绍针对整段 IR 代码翻译的指令选择问题和解法。我们暂不考虑具体可用的寄存器编号和数目，均使用 w0...wn（32 位寄存器）或 x0...xn（64 位寄存器）表示。

11.1 AArch64 指令集架构

ARM-v8a 是精简指令集，其主要特点是访存与运算由不同的指令分别完成。代码 11.2展示了一段简单的 hello world 汇编代码。这段代码先通过 `str` 指令将返回地址寄存器 `x30` 保存到栈上，然后通过 `adrp` 或取字符串`"Hello World!"` 字符串所在的内存页地址，加上偏移量后获得字符串的地址，最后通过 `bl` 调用 `puts` 函数，恢复寄存器 `x30` 后返回。

```
    .text
    .global main
main:
    str     x30, [sp, -16]!
    adrp    x0, .LC0
    add     x0, x0, :lo12:.LC0
    bl      puts
    ldr     x30, [sp], 16
    ret
.LC0:
    .string "Hello World!"
```

代码 11.1: ARM-v8A 指令：Hello World 程序

11.1.1 寻址模式

在 ARMv8-A 架构中，根据程序的变量作用域和存储位置，可以将寻址需求划分为以下两大类：全局变量的寻址和局部变量的寻址。全局变量通常位于静态存储区（如数据段或只读段），它们的地址在程序运行时通常是固定的。一般需要先通过 `adrp` 获取其内存页的地址（4KB 对齐），然后加上其低 12 位地址。

局部变量通常分配在栈上，包括以下寻址方式：

- **立即偏移寻址模式**：使用基地址寄存器和一个立即数偏移值计算目标地址，如 `ldr x2, [x1]` 表示加载内存地址 `x1` 中的数据到 `x2` 中，`ldr x2, [x1, #10]` 表示加载地址 `x1+10` 中的数据到 x2 中。
- **寄存器偏移寻址模式**：偏移量由另一个寄存器的值提供，如 `ldr x2, [x1, x0]` 表示加载地址 `x1+x0` 中的数据到 x2 中。另外，也可以结合移位操作实现更复杂的偏移，如 `ldr x2, [x1, x0, lsl #3]` 表示加载地址 `x1+x0*8` 的数值到 `x2` 中。

- **预索引寻址模式**：访存前先调整基地址，如 `ldr x2, [x1, #10]!` 表示先更新 `x1` 的值为 `x1+10`，再将加载地址 `x1` 中的数据到 `x2` 中。
- **后索引寻址模式**：访存前再调整基地址，如 `ldr x2, [x1], #10` 先加载地址 `x1` 中的数据到 `x2` 中，再将 `x1` 更新为 `x1+10`。
- **PC 相对寻址模式**：地址是当前程序计数器的值加上一个偏移量，如 `ldr x2, label`。
- **栈寻址模式**：通过栈指针访问数据：如 `ldr x2, [sp, #8]` 加载地址 `sp+8` 中的数据到 `x1`。

11.1.2 立即数支持

由于 ARMv8-A 指令是 32 位定长编码的，对于参数是立即数的情况支持比较有限。大多数算术数据处理指令支持 12 位无符号整数（范围为 0-4095）。如果立即数过大，可通过移位机制扩展表示。例如，加载立即数 65539 可以通过以下两条指令实现。其中 `movk` 指令表示保持寄存器低位不变。另外，如果一个整数的低 12 为为 0，且大小不超过 2^{24}，一般也可以直接使用。

```
mov x8, 3 ; 将\texttt{x8}设置为3
movk x8, 1, lsl 16 ; 将\texttt{x8}寄存器16-31位设置为1
```

代码 11.2: ARM-v8A 指令：立即数示例

逻辑运算的立即数支持与算数运算指令不同，它采用了一种特殊的编码机制，允许表示复杂的位模式，如连续的 1 位掩码（0xFF）和周期性掩码（如 0xAAAAAAAA）。

11.1.3 主要指令

下面以翻译单条 LLVM IR 指令为目标讲解主要的 ARM-v8A 指令，见表 11.1。实际 ARMv8-A 手册中的指令有几百条。如需了解更多的指令和机制，可参考 ARM 公司提供的官方手册 [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`	为局部变量分配栈内存；sp 指针要求 16 字节对齐
`store i32 %0, i32* %a`	`str w0, [sp, 12]`	将寄存器值保存到内存地址 sp+12
`store i32 1, i32* %a`	`mov w0, 1` `str w0, [sp, 12]`	将整数保存到栈空间；str 操作数不能为立即数
`%a0 = load i32, i32* %a`	`ldr w0, [sp, 12]`	将局部变量值由内存地址 sp+12 加载到寄存器
`%g0 = load i32, i32* @g`	`adrp x8, g` `ldr w0, [x8, :lo12:g]`	将全局变量值加载到寄存器
`%r = add i32 %a, %b`	`add w0, w1, w2`	两个寄存器的值相加，结果保存到 w1
`%r = add i32 %a, 4095`	`add w0, w1, 4095`	整数范围：$x \in [0, 2^{12})$，以及 $x * 2^{12}$
`%r = sub i32 %a, %b`	`sub w0, w1, w2`	两个寄存器的值相减；亦支持减立即数，方法同加法
`%r = mul i32 %a, %b`	`mul w0, w1, w2`	不支持立即数
`%r = sdiv i32 %a, %b`	`sdiv w0, w1, w2`	不支持立即数
`%r = icmp sgt i32 %a, %b` `br i1 %r, label %bb1, label %bb2`	`cmp w0, w1` `b.le .LBB2`	比较：支持一个立即数，结果存到 CPSR 寄存器；然后条件跳转
`%r = xor i32 %a, %b`	`eor w0, w1, w1`	异或运算，支持一个立即数
`%r = and i32 %a, %b`	`and w0, w1, w1`	与运算，支持一个立即数
`%r = or i32 %a, %b`	`orr w0, w1, w1`	或运算，支持一个立即数
`call void @foo()`	`bl foo`	函数调用
`ret i32 %r`	`ldr x30, [sp], 16` `ret`	将返回地址存入 x30 寄存器，还原栈顶指针，返回

11.2 消除 phi 指令

LLVM IR 中由于转换 SSA 会引入 phi 指令，但不存在直接与 phi 匹配的 ARM 指令。以代码 11.3为例，我们可以通过两种方式消除 phi 指令：

```
bb1:
    %r1 = icmp eq i32 %a1, 0
    ; 方式一：store i32 %a1, i32* %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, i32* %a
    ; 方式二：%a3 = %a2
    br label %bb2
bb3:
    %a3 = phi i1 [%a1, %bb1], [%a2, %bb2]
    ; 方式一：%a3 = load i32, i32* %a
    %r1 = add i32 %a3, %b1
```

代码 11.3: LLVM IR 代码：phi 指令消除的例子

- **使用 store-load 替换 phi**：在 phi 指令的前驱代码块跳转指令前增加 store 指令，并将 phi 指令替换为 load 指令。以代码 11.3为例，我们分别在代码块%bb1 和%bb2 中增加对变量 a 的 store 语句，并将 phi 指令替换为对变量 a 的 load 语句。这种方法的缺点是会引入非必要的仿存操作。
- **使用伪赋值指令替换 phi**：在 phi 指令的前驱代码块直接对 phi 指令的目标寄存器进行进行赋值。以代码 11.3为例，在代码块%bb1 和%bb2 中增加对虚拟寄存器%a3 赋值的伪指令。由于 LLVM IR 并不支持这种直接赋值的指令形式，因此这种修改后的代码无法使用标准的 lli 执行。这种方法的优点是可以避免不必要的仿存操作，最大化寄存器使用。

11.3 指令选择问题

通过前面的介绍，我们可以轻松地为单条 IR 指令找到对应的汇编指令翻译方式。然而，仅考虑单条 IR 指令直接翻译为汇编代码的方式通常并非最优，因为许多汇编指令能够覆盖多条 IR 指令的功能。例如，一些复合算术运算指令（如乘法累加或减法）可以同时对应 IR 中的两条指令：乘法和加法（或减法）。本节将重点讨论指令选择问题，特别是如何为这些情况生成更加高效的汇编代码。

```
madd x0, x1, x2, x3 ; x0 = x1 * x2 + x3; mul指令本质上是该指令在x3=0时的特例
msub x0, x1, x2, x3 ; x0 = x1 * x2 - x3;
```

代码 11.4: ARM-v8A 指令：复合算数运算

下面我们将重点讨论单个代码块的指令选择问题。由于一个函数可以划分为多个代码块，因此可以独立处理各代码块的指令选择问题，从而简化指令翻译问题的复杂度。

11.3.1 指令选择图

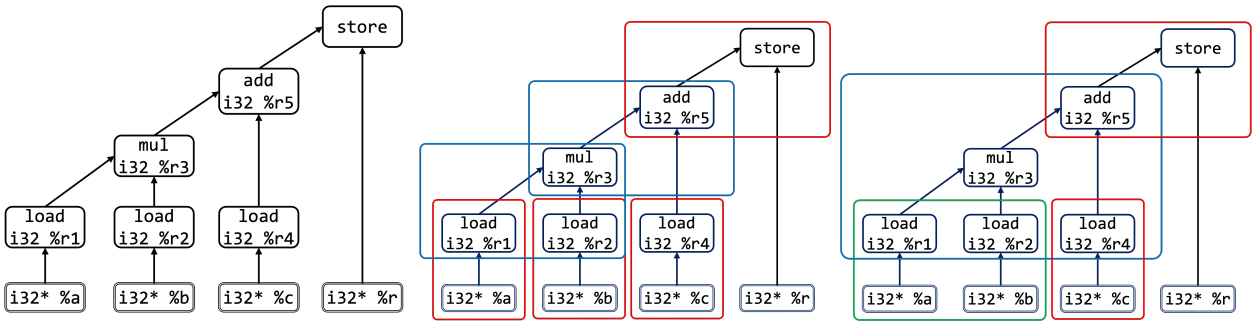
我们将该单个代码块内的 IR 表示为指令选择图，从而对该问题进行建模。

定义 1 (指令选择图)**.** 指令选择图是一个有向无环图，包括两种类型的节点：指令节点和数据存储节点；其中的边表示指令运行所需的参数。

以代码 11.5为例，其指令选择图可表示为图 11.1a，指令执行顺序只需满足拓扑排序即可。

```
%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 11.5: 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

```
%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, i32* %aj
store i32 %aj1, i32* %ai
%t1 = load i32, i32* %t
store i32 %t1, i32* %aj
%ai1 = load i32, i32* %ai
store i32 %ai1, i32* %t
```

代码 11.6: LLVM IR 代码：内存同步问题

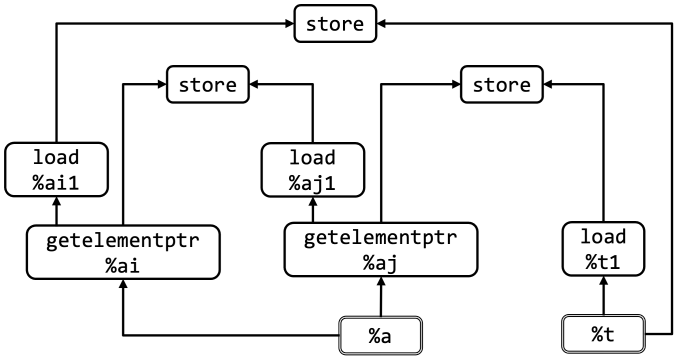
值得注意的是，当代码块中出现 store 等涉及内存同步的指令时，需要对代码块进行分割，为指令前后的 IR 分别绘制指令选择子图，从而保证汇编代码和 IR 语义的一致性。以代码 11.6为例，图 11.2a为直接绘制的指令选择图，其中存在多种满足拓扑排序的 store-load 和 store-store 顺序，且语义不相同。以 store 指令为边界进行分割后可以构造三个存在顺序关系的指令选择子图：图 11.2b、11.2c和 11.2d。

11.3.2 铺树问题

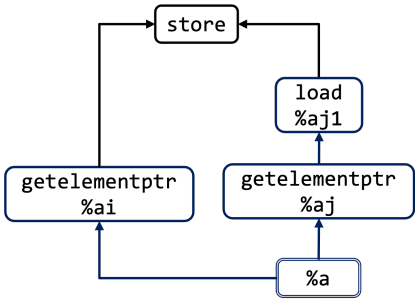
通过指令选择图，我们可以将指令选择问题转化成铺树（图）问题，即如何翻译汇编指令，使其可以覆盖指令选择图上的所有节点，同时使得目标汇编代码指令数最少、运行时间最优。以图 11.1a为例，该图至少包含图 11.1b和图 11.1c中的两种铺树方案。其对应的汇编代码分别为代码 11.7和代码 11.8。我们很容易看出代码 11.8对应的指令数更少。如果 `ldr` 和 `ldp` 指令的性能开销以及 `mul` 和 `madd` 指令的性能开销都相同，则明显图 11.1c对应的铺树方案更优。

铺树问题是一个 NP-hard 问题，可以通过贪心法或动态规划求解。一种常用的贪心法称为 Maximal Munch，即每步选择覆盖节点最多的方案进行铺树。

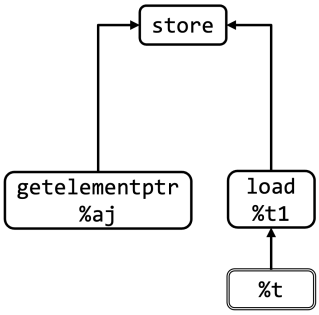
```
ldr w1, [sp, .a]
ldr w2, [sp, .b]
```



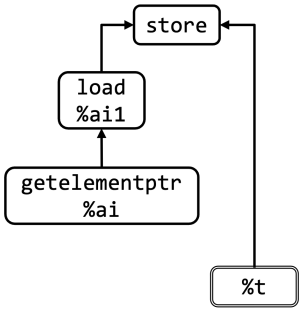
(a) 直接为代码 11.6绘制指令选择图



(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: 代码块设计内存同步问题时的指令选择子图分割

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

代码 11.7: 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 11.8: 图 11.1c对应的汇编代码

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 %bb4
bb5:
    %x6 = phi i32 [ %x3, %bb3 ], [ %x5, %bb4 ]
    br label %bb1
bb6:
    ret i32 %x1
}
```

代码 11.9: LLVM IR 代码

Bibliography

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