30 | 目标代码的生成和优化（二）：如何适应各种硬件架构？

2019-11-01 宫文学

讲述：宫文学
时长 16:56 大小 15.52M

前一讲，我带你了解了指令选择和寄存器分配，本节课我们继续讲解目标代码生成的，第三个需要考虑的因素：**指令重排序（Instruction Scheduling）**。

我们可以通过重新排列指令，让代码的整体执行效率加快。那你可能会问了：就算重新排序了，每一条指令还是要执行啊？怎么就会变快了呢？

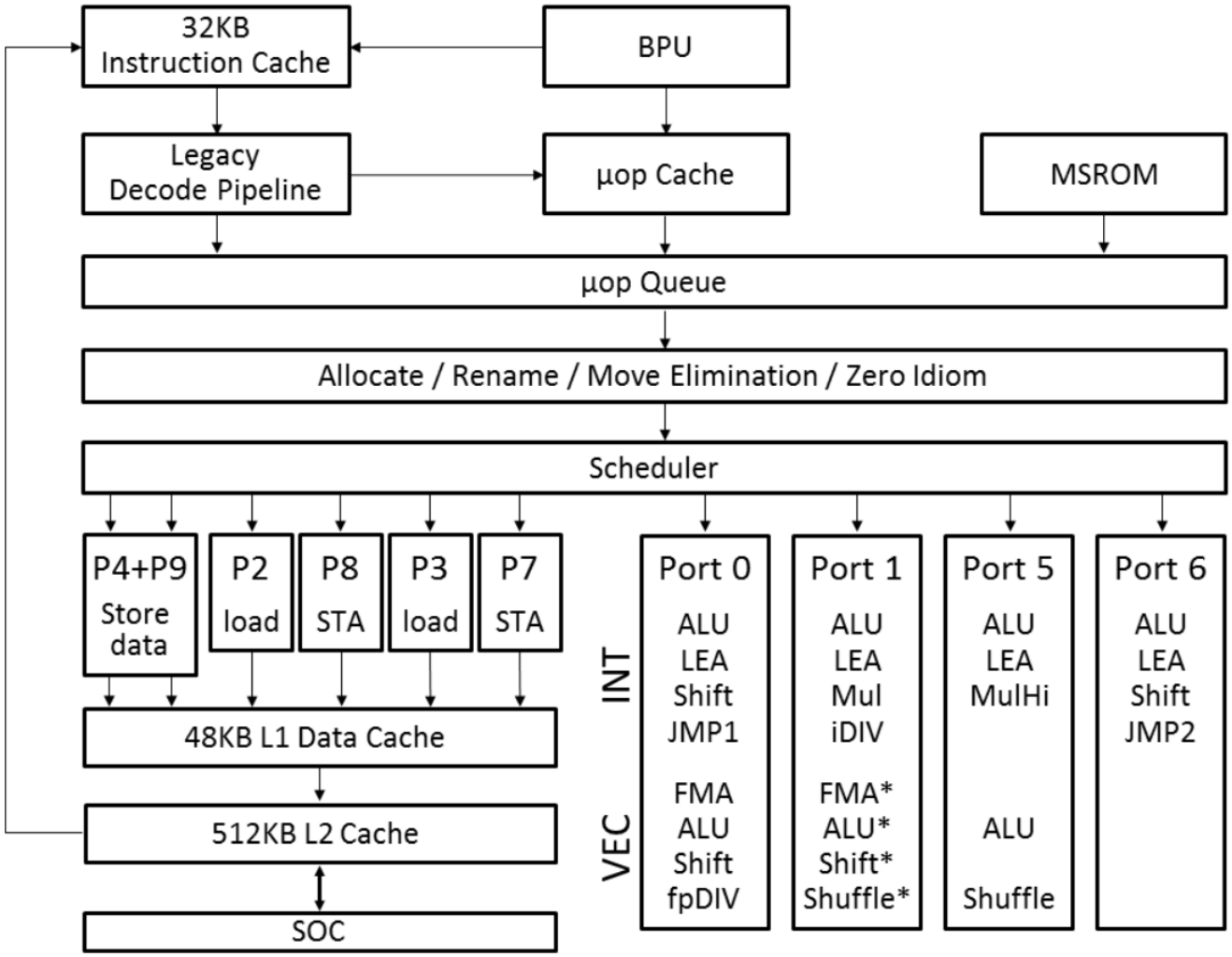
别着急，本节课我就带你探究其中的原理和算法，来了解这个问题。而且，我还会带你了解 LLVM 是怎么把指令选择、寄存器分配、指令重排序这三项工作组织成一个完整流程，完成目标代码生成的任务的。这样，你会对编译器后端的代码生成过程形成完整的认知，为正式做一些后端工作打下良好的基础。

首先，我们来看看指令重排序的问题。

指令重排序

如果你有上面的疑问，其实是很正常的。因为我们通常会把 CPU 看做一个整体，把 CPU 执行指令的过程想象成，依此检票进站的过程，改变不同乘客的次序，并不会加快检票的速度。所以，我们会自然而然地认为改变顺序并不会改变总时间。

但当我们进入 CPU 内部，会看到 CPU 是由多个功能部件构成的。下图是 Ice Lake 微架构的 CPU 的内部构成（从Intel 公司的技术手册中获取）：



在这个结构中，一条指令执行时，要依次用到多个功能部件，分成多个阶段，虽然每条指令是顺序执行的，但每个部件的工作完成以后，就可以服务于下一条指令，从而达到并行执行的效果。这种结构叫做**流水线（pipeline）结构。**我举例子说明一下，比如典型的 RISC 指令在执行过程会分成前后共 5 个阶段。

IF：获取指令；

ID（或 RF）：指令解码和获取寄存器的值；

EX：执行指令；

ME（或 MEM）：内存访问（如果指令不涉及内存访问，这个阶段可以省略）；

WB：写回寄存器。

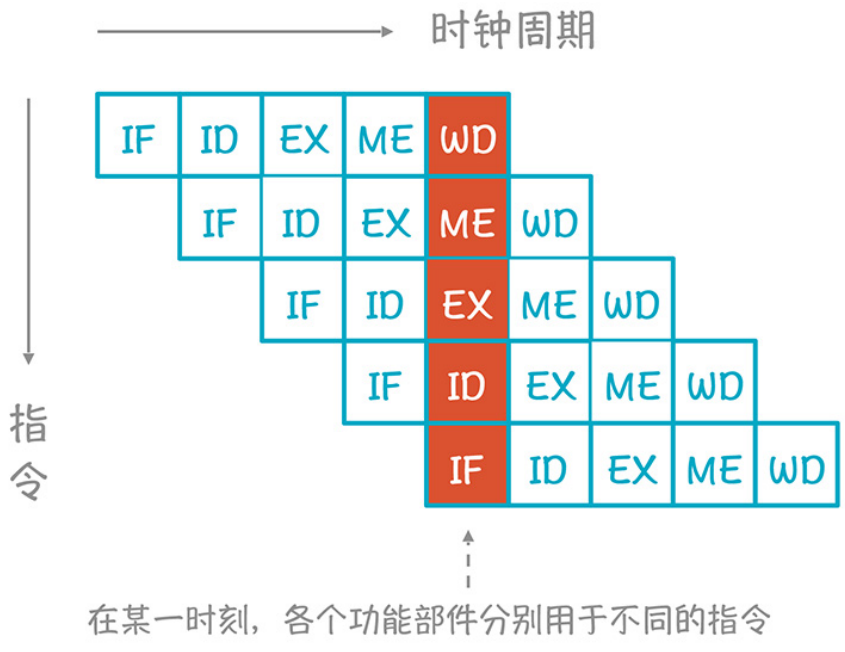
对于 CISC 指令，CPU 的流水线会根据指令的不同，分成更多个阶段，比如 7 个、10 个甚至更多。

在执行指令的阶段，不同的指令也会由不同的单元负责，我们可以把这些单元叫做执行单元，比如，Intel 的 Ice Lake 架构的 CPU 有下面这些执行单元：

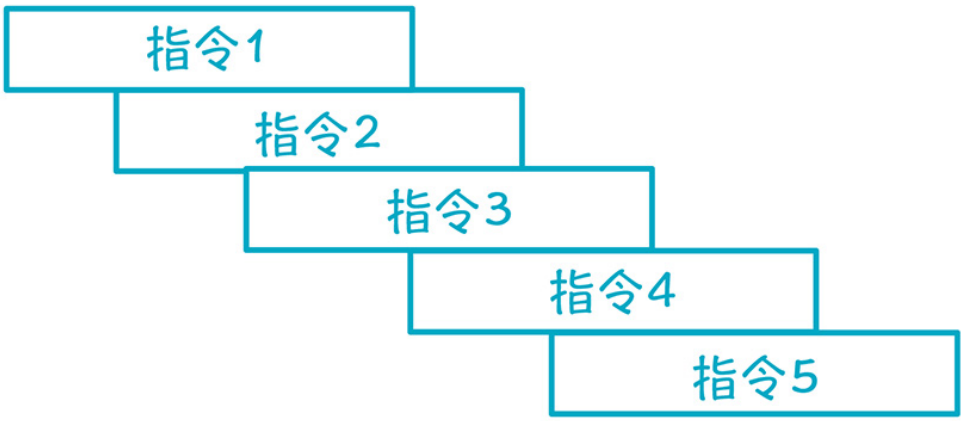
执行单元	数量	代表性的指令
ALU	4	add, and, cmp, or, test, xor, movzx, movsx, mov...
SHFT	2	sal, shl, rol, adc, sarx, adcx, adox...
Slow Int	1	mul, imul, bsr, rcl, shld, mulx, pdep...

其他执行单元还有：BM、Vec ALU、Vec SHFT、Vec Add、Vec Mul、Shuffle 等。

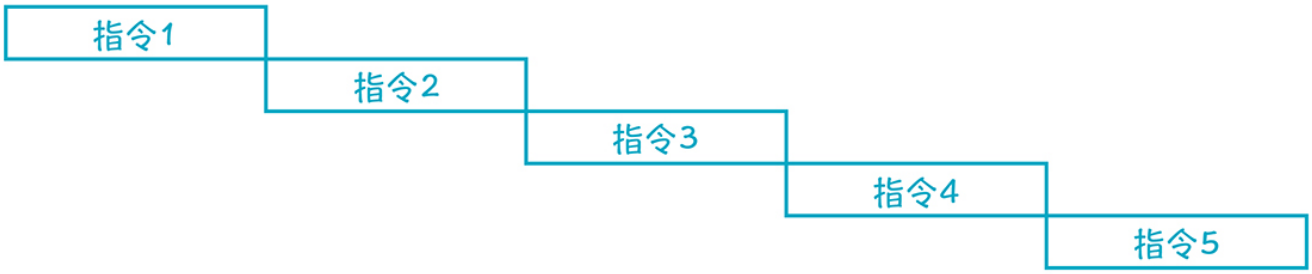
因为 CPU 内部存在着多个功能单元，所以在同一时刻，不同的功能单元其实可以服务于不同的指令，看看下面这个图；



这样的话，多条指令实质上是并行执行的，从而减少了总的执行时间，这种并行叫做**指令级并行：**



如果没有这种并行结构，或者由于指令之间存在依赖关系，无法并行，那么执行周期就会大大加长：



我们来看一个实际的例子。

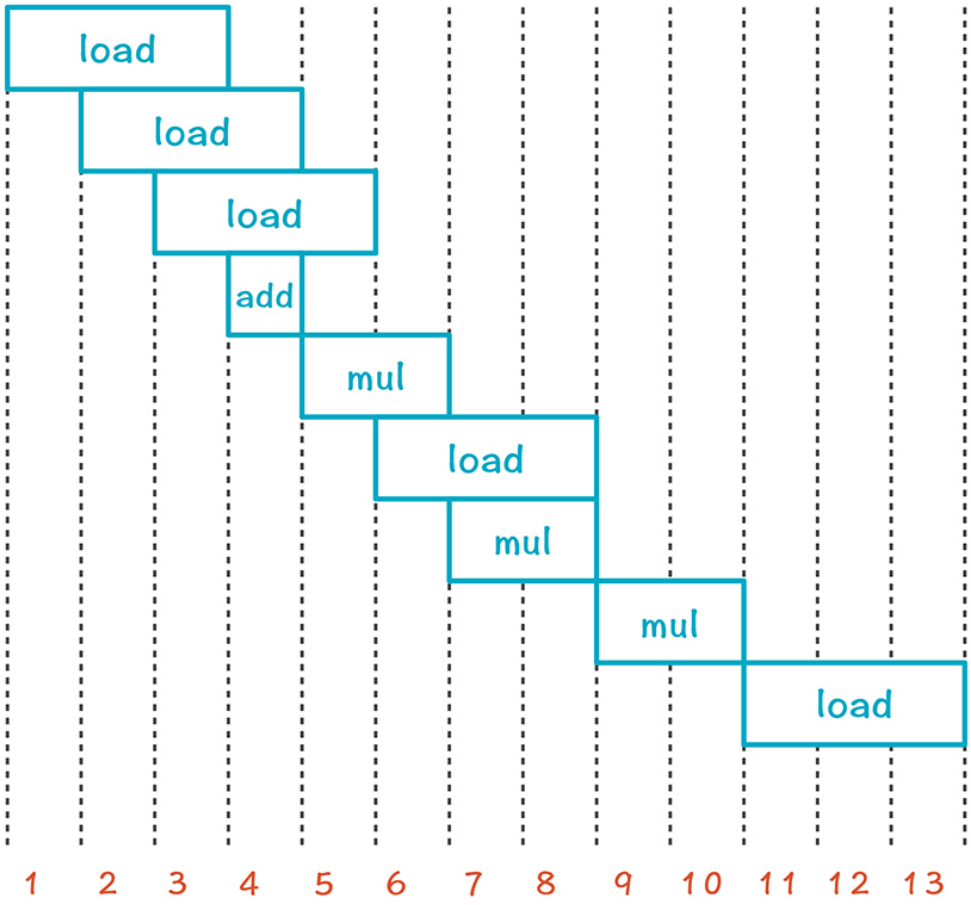
为了举例子方便，我们做个假设：假设 load 和 store 指令需要 3 个时钟周期来读写数据，add 指令需要 1 个时钟周期，mul 指令需要 2 个时钟周期。

图中橙色的编号是原来的指令顺序，绿色的数字是每条指令开始时的时钟周期，你把每条指令的时钟周期累计一下就能算出来。最后一条指令开始的时钟周期是 20，该条指令运行需要 3 个时钟周期，所以在第 22 个时钟周期执行完所有的指令。右边是重新排序后的指令，一共花了 13 个时钟周期。**这个优化力度还是很大的！**

仔细看一下左边前两条指令，这两条指令的意思是：先加载数据到寄存器，然后做一个加法。但加载需要 3 个时钟周期，所以 add 指令无法执行，只能干等着。

右列的前三条都是 load 指令，它们之间没有数据依赖关系，我们可以每个时钟周期启动一个，到了第四个时钟周期，每一条指令的数据已经加载完毕，所以就可以执行加法运算了。

我们可以把右边的内容画成下面的样子，你能看到，很多指令在时钟周期上是重叠的，**这就是指令级并行的特点。**



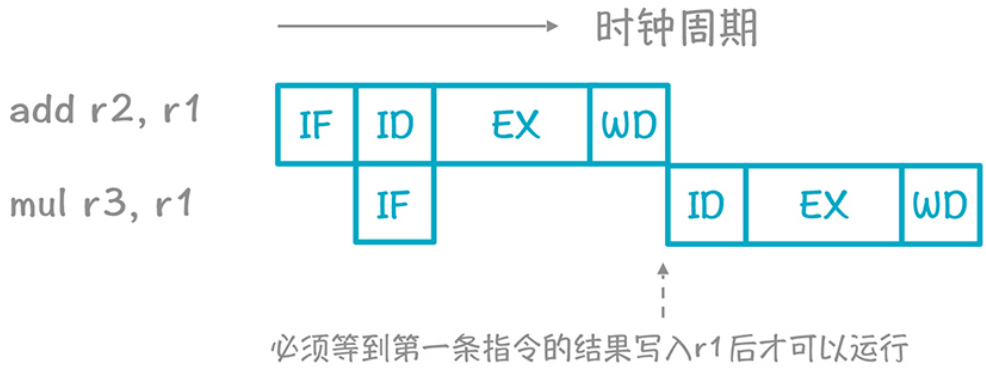
当然了，不是所有的指令都可以并行，最后的 3 条指令就是顺序执行的，导致无法并行的原因有几个：

数据依赖约束

如果后一条指令要用到前一条指令的结果，那必须排在它后面，比如下面两条指令：add 和 mul。

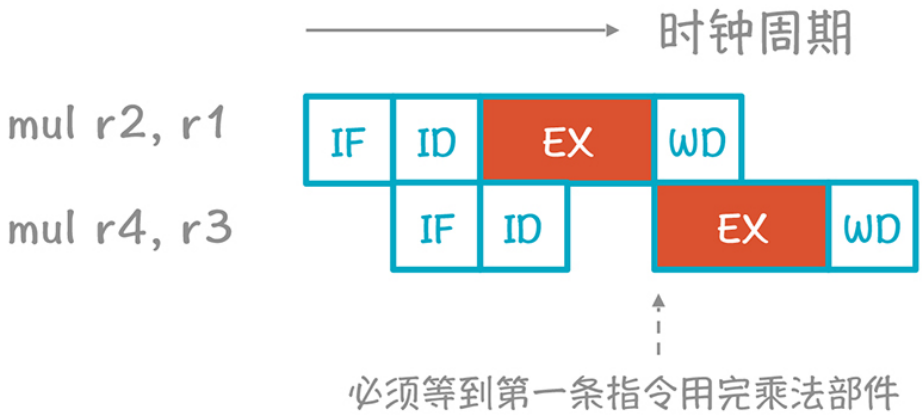
对于第二条指令来说，除了获取指令的阶段（IF）可以和第一条指令并行以外，其他阶段需要等第一条指令的结果写入 r1，第二条指令才可以使用 r1 的值继续运行。


```
1 add r2, r1
2 mul r3, r1
```



功能部件约束

如果只有一个乘法计算器，那么一次只能执行一条乘法运算。



指令流出约束

指令流出部件一次流出 n 条指令。

寄存器约束

寄存器数量有限，指令并行时使用的寄存器不可以超标。

后三者也可以合并成为一类，称作资源约束。

在数据依赖约束中，如果有因为使用同一个存储位置，而导致不能并行的，可以用重命名变量的方式消除，这类约束被叫做伪约束。而先写再写，以及先读后写是伪约束的两种呈现方式：

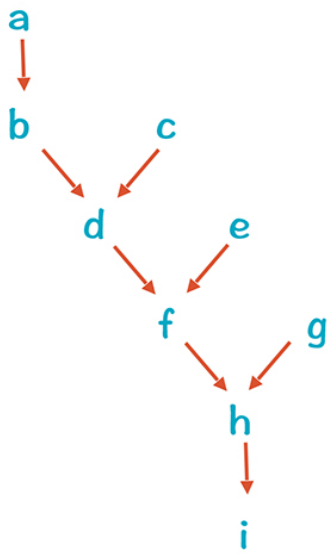
先写再写：如果指令 A 写一个寄存器或内存位置，B 也写同一个位置，就不能改变 A 和 B 的执行顺序，不过我们可以修改程序，让 A 和 B 写不同的位置。

先读后写：如果 A 必须在 B 写某个位置之前读某个位置，那么不能改变 A 和 B 的执行顺序。除非能够通过重命名让它们使用不同的位置。

以上就是指令重排序的原理，掌握这个原理你就明白为什么重排序可以提升性能了，**不过明白原理之后，我们还有能够用算法实现出来才行。**

用算法排序的关键点，是要找出代码之间的数据依赖关系。下图展现了示例中各行代码之间的数据依赖，可以叫做**数据的依赖图（dependence graph）。**它的边代表了值的流动，比如 a 行加载了一个数据到 r1，b 行利用 r1 来做计算，所以 b 行依赖 a 行，这个图也可以叫做优先图（precedence graph），因为 a 比 b 优先，b 比 d 优先。

序号	开始	指令
a	1	load a, r1
b	4	add r1, r1
c	5	load b, r2
d	8	mul r2, r1
e	10	load c, r2
f	13	mul r2, r1
g	15	Load d, r2
h	18	mul r2, r1
i	20	store r1, a



我们可以给图中的每个节点再加上两个属性，利用这两个属性，就可以对指令进行排序了：

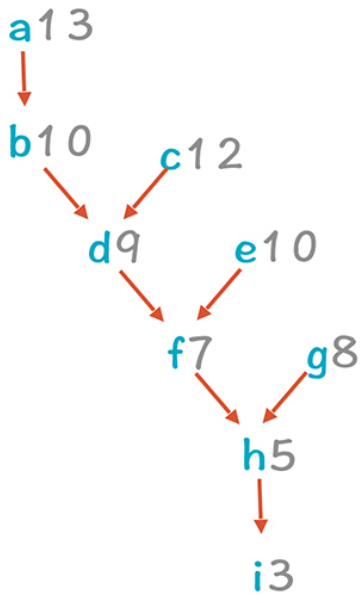
一是操作类型，因为这涉及它所需要的功能单元。

二是时延属性，也就是每条指令所需的时钟周期。

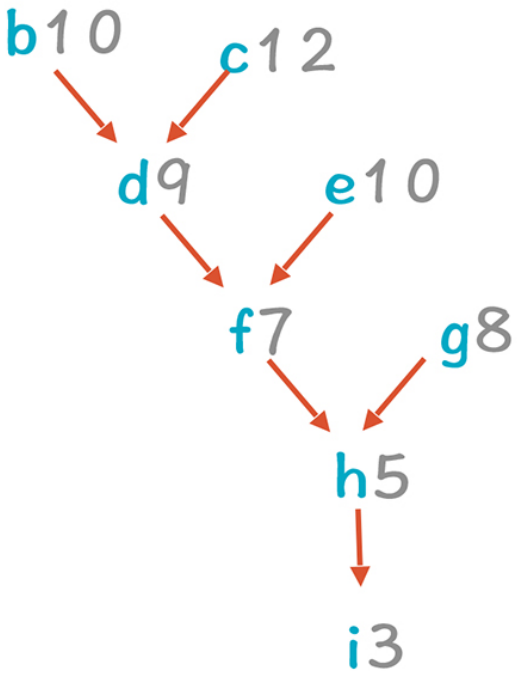
图中的 a、c、e、g 是叶子，它们没有依赖任何其他的节点，所以尽量排在前面。b、d、ef、h 必须出现在各自所依赖的节点后面。而根节点 i，总是要排在最后面。

根据时延属性，我们计算出了每个节点的累计时延（每个节点的累计时延等于父节点的累计时延加上本节点的时延）。其中 a-b-d-f-h-i 路径是关键路径，代码执行的最少时间就是这条路径所花的时钟周期之和。

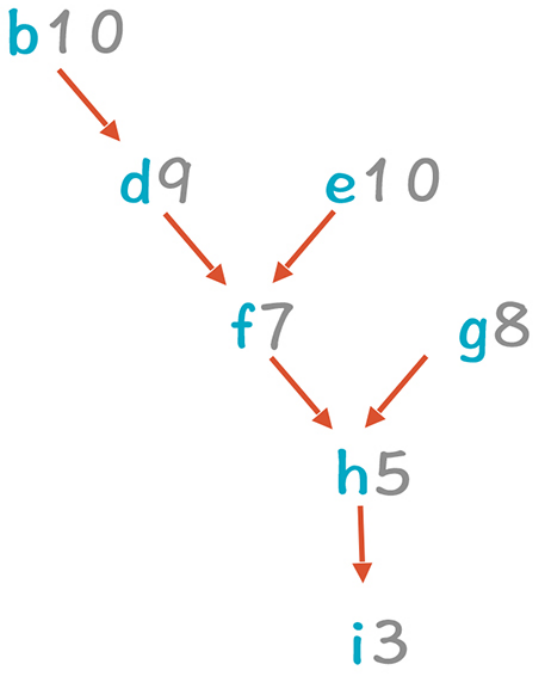
序号	开始	指令
a	1	load a, r1
b	4	add r1, r1
c	5	load b, r2
d	8	mul r2, r1
e	10	load c, r2
f	13	mul r2, r1
g	15	Load d, r2
h	18	mul r2, r1
i	20	store r1, a



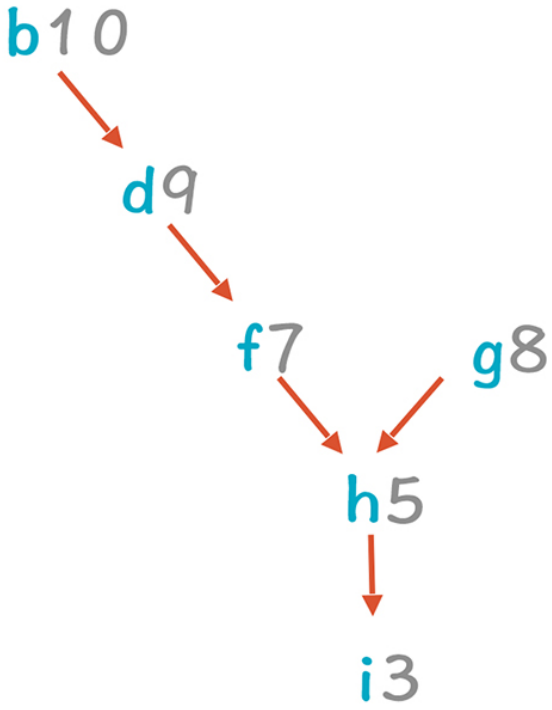
因为 a 在关键路径上，所以首先考虑把 a 节点排在第 1 行。



剩下的树中，c-d-f-h-i 变成了关键路径，因为 c 的累计时延最大。c 节点可以排在第 2 行。



b 和 e 的累计时延都是最长的，但由于 b 必须在 a 执行完毕后，才会开始执行，所以最好等够 a 的 3 个时钟周期，否则还是会空等，所以先不考虑 b，而是把 e 放到第 3 行。



继续按照这个方式排，最后可以获得 a-c-e-b-d-g-f-h-i 的指令序列。不过这个代码其实还可以继续优化：也就是发现并消除其中的伪约束。

c 和 e 都向 r2 里写了值，而 d 使用的是 c 写入的值。如果修改变量名称，比如让 e 使用 r3 寄存器，我们就可以去掉 e 跟 d，以及 e 与 c 之间伪约束，让 e 就可以排在 c 和 d 之前。同理，也可以让 g 使用 r4 寄存器，使得 g 可以排在 e 和 f 的前面。当然了，在这个示例中，这种改变并没有减少总的时间消耗，因为关键路径上的依赖没有变化，它们都使用了 r1 寄存器。但在别的情况下，就有可能带来更大的优化。

我们刚才其实是采用了一种最常见的算法，List Scheduling 算法，**大致分为 4 步：**

1. 把变量重命名来消除伪约束（可选步骤）。

2. 创建依赖图。

3. 为每行代码计算优先值（计算方法可以有很多，比如我们示例中基于最长时延的方法就是一种）。

4. 迭代处理代码并排序。

除了 List Scheduling 算法以外，还有其他的算法，这里我就不展开了。不过，讲到算法时，我们需要考虑算法的复杂度。前一讲讲算法时，我没有提这个问题，是想在这里集中讲一下。

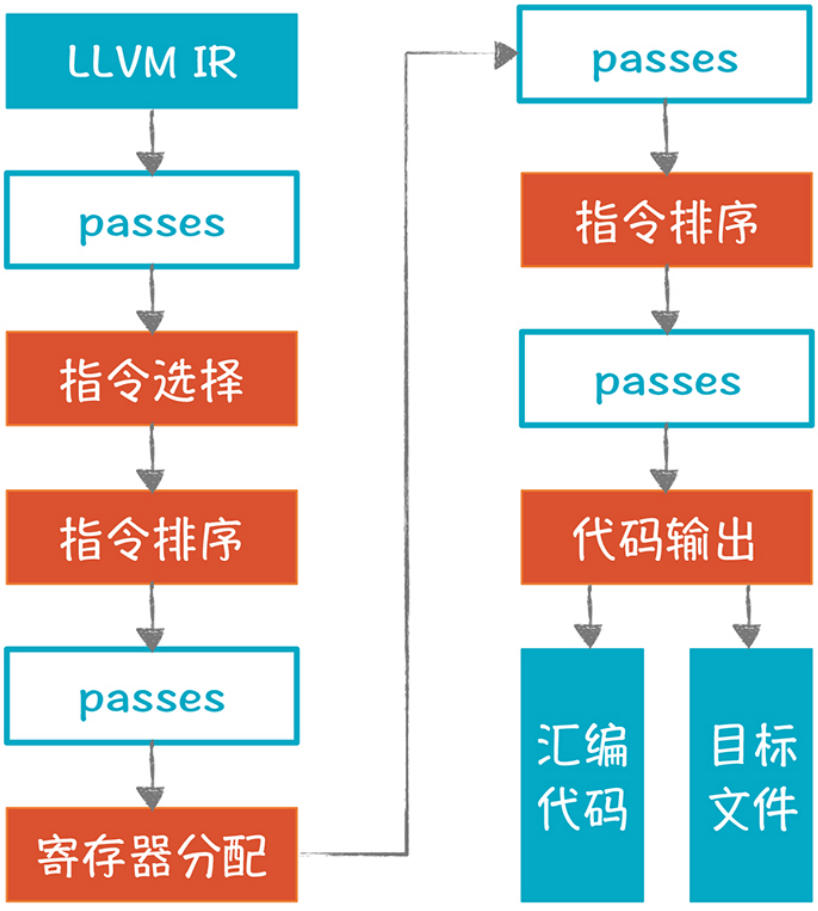
这两节课中，关于指令选择、寄存器分配和指令重排序的算法，其难度（时间复杂度）都是“NP- 完全”的。“NP- 完全”是什么意思呢？也就是这类问题找不到一个随规模（代码行数）计算量增长比较慢的算法（多项式时间算法）来找到最优解。反之，有可能计算量会随着代码行数呈指数级上升。因此，编译原理中的一些难度最高的算法，都在代码生成这一环。

当然了，找最优解太难，我们可以退而求其次，找一个次优解。就比如我们用地图软件导航的时候，没必要要求导航路径每次都是找到最短的。这时，就会有比较简单的算法，计算量不会随规模增长太快，但结果还比较理想。**我们这两讲的算法都是这个性质的。**

到目前为止，我带你了解了目标代码生成的三大考虑因素：指令选择、寄存器分配和指令重排序。现在，我们来看看目标代码生成，在 LLVM 中是如何实现的，这样，你能从概念过渡到实操，从而把知识点掌握得更加扎实。

LLVM 的实现

LLVM 的后端需要多个处理步骤来生成目标代码：



图中橙色的部分是重要的步骤，它本身包含了多个 Pass，所以也叫做超级 Pass。图中蓝框的 Pass，是用来做一些额外的优化处理（关于 LLVM 的 Pass 机制，我在 27 讲提到过，如果你忘记了，可以回顾一下）。

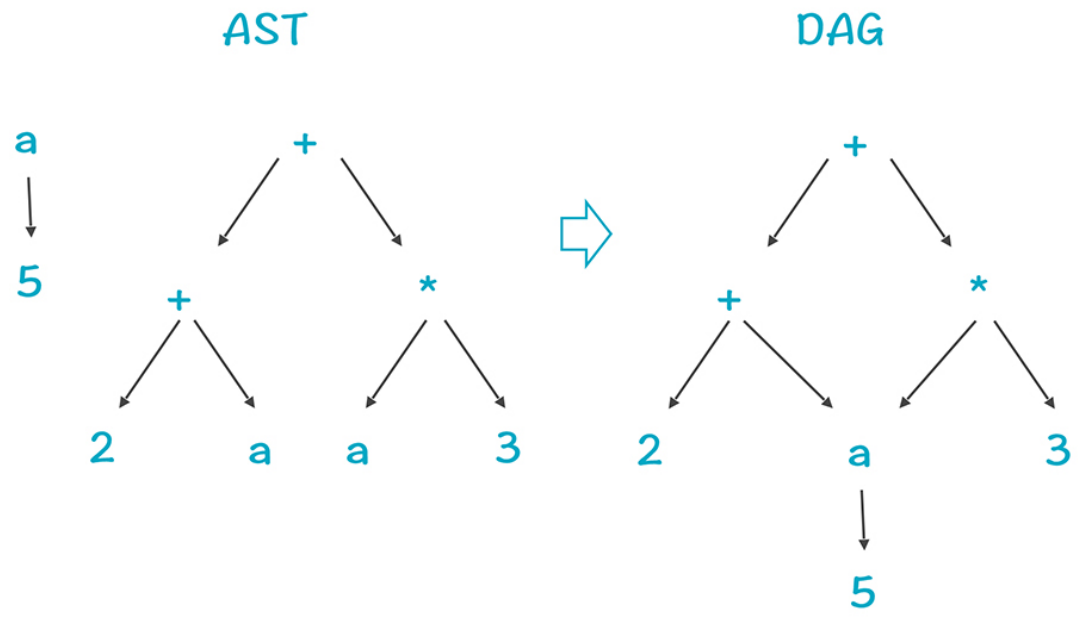
接下来，我来讲解一下 LLVM 生成目标代码的关键步骤。

指令选择

LLVM 的指令选择算法是基于 DAG（有向无边图）的树模式匹配，与前一讲基于 AST 的算法有一些区别，但总思路是一致的（具体算法描述可以参见这篇论文）。这个算法是 Near-Optimal（接近 Optimal）的，能够在线性的时间内完成指令的选择，并且它特别关注产生的代码的尺寸，要求尺寸足够小。

DAG 是融合了公共子表达式的 AST，也是一种结构化的 IR。下面两行代码对应的 AST 和 DAG 分如图所示，你能看到，DAG 把 a=5 这棵子树给融合了：

```
1  a = 5
2  b = (2 + a) + (a * 3)
```



LLVM 把内存中的 IR 模型，转化成了一个体现了某个目标平台特征的 SelectionDAG，用于做指令选择。每个基本块转化成一个 DAG，DAG 中的节点通常代表指令，边代表指令

之间的数据流动。

在这个阶段之后，LLVM 会把 DAG 中的 LLVM IR 节点，全部转换成目标机器的节点，代表目标机器的指令，而不是 LLVM 的指令。

指令排序（寄存器分配之前）

基于前一步的处理结果，我们要对指令进行排序。但因为 DAG 不能反映没有依赖关系的节点之间的排序，所以 LLVM 要先把 DAG 转换成一种三地址模式，这个格式叫做 MachineInstr。这个阶段会把指令排序，并尽量发挥指令级并行的能力。

寄存器分配

接下来做寄存器的分配。LLVM 的 IR 支持无限多的寄存器，在这个环节要分配到实际的寄存器上，分配不下的就溢出到内存。

指令排序（寄存器分配之后）

分配完寄存器之后，LLVM 会再做一次指令排序。因为寄存器分配，会指定确定的寄存器，而访问不同的寄存器的时钟周期，可能是不同的。对于溢出到内存中的变量，也增加了一些指令在内存和寄存器之间传输数据。利用这些信息，LLVM 可以进一步优化指令的排序。

代码输出

做完上面的所有工作后，就可以输出目标代码了。

LLVM 在这一步把 MachineInstr 格式转换为 MCInst 格式，因为后者更有利于汇编器和链接器输出汇编代码或二进制目标代码。

在这里，我想延伸一下，和你探讨一个问题：如果现在有一个新的 CPU 架构，要实现一个崭新的后端，来支持各种语言，应该怎么做。

在我国大力促进芯片研发的背景下，这是一个值得探讨的问题，新芯片需要编译器的支持才可以呀。你要实现各种指令选择的算法、寄存器分配的算法、指令排序的算法来反映这款

CPU 的特点。

对于这个难度颇高的任务，LLVM 的 TableGen 模块会给你提供很大的帮助。这个模块能够帮助你为某个新的 CPU 架构快速生成后端。你可以用一系列配置文件定义你的 CPU 架构的特征，比如寄存器的数量、指令集等等。

一旦你定义了这些信息，TableGen 模块就能根据这些配置信息，生成各种算法，如指令选择器、指令排序器、一些优化算法等等。这就像编译器前段工具可以帮你生成词法分析器，和语法分析器一样，能够大大降低开发一个新后端的工作量，所以说，把 LLVM 研究透彻，会有助于你在这样的重大项目中发挥重要作用。

课程小结

本节课，我讲解了目标代码生成的第三个主题：指令重排序。

要理解这个主题，你首先要知道 CPU 内部是分成多个功能部件的，要知道一条指令的执行过程中，指令获取、解码、执行、访问数据都是如何发生的，这样你会知道指令级并行的原理。

其次，从算法角度，你要知道 List Scheduling 的步骤，掌握基于最大时延的优先级计算策略。有了这个基础之后，你可以进一步地研究其他算法。

我想强调的是，指令选择、寄存器分配、指令重排序这三个领域的算法，都是“NP- 完全”的，所以寻找优化的算法，是这个领域最富有挑战的任务。要研究清楚这些算法，你需要阅读相关的资料，比如本讲推荐的论文和其他该领域的经典论文。

另外，我建议你阅读 CPU 厂商的手册，因为只有手册才会提供相关 CPU 的具体信息，解答你对技术细节的一些疑惑。比如网上曾经有人提问说：为什么 mov 指令要用到 ALU 部件？这个其实看一下手册就知道了。

最后，我带你了解了 LLVM 是如何做这些后端工作的，这样可以加深你对代码生成这部分知识的了解。

一课一思

为了方便教学，本讲的示例用的时延值都比较少，这其实是不符合实际的。假设我们忽略指令获取和解码的阶段，只考虑执行和写入寄存器两个阶段，这时候 add 指令需要 3 个时钟周期（2 个执行，1 个写寄存器），mul 指令也需要 3 个时钟周期，那么会对示例代码的排序产生什么影响呢？你可以实际推演一下，这对于你理解指令重排序的算法会很有帮助。

最后，感谢你的阅读，如果这篇文章让你有所收获，也欢迎你将它分享给更多的朋友。

上一篇　29 | 目标代码的生成和优化（一）：如何适应各种硬件架构？

下一篇　31 | 内存计算：对海量数据做计算，到底可以有多快？

精选留言 (2)

写留言

Giacomo
2019-11-03

后端比前端难了好多啊

展开

沉淀的梦想

2019-11-01

不太理解，为什么文中例子的指令重排序结果是a-c-e-b-d-g-f-h-i？b,d不是存在数据依赖吗？而且add的时钟周期为2,这么排应该会导致停顿啊

展开