



下载APP

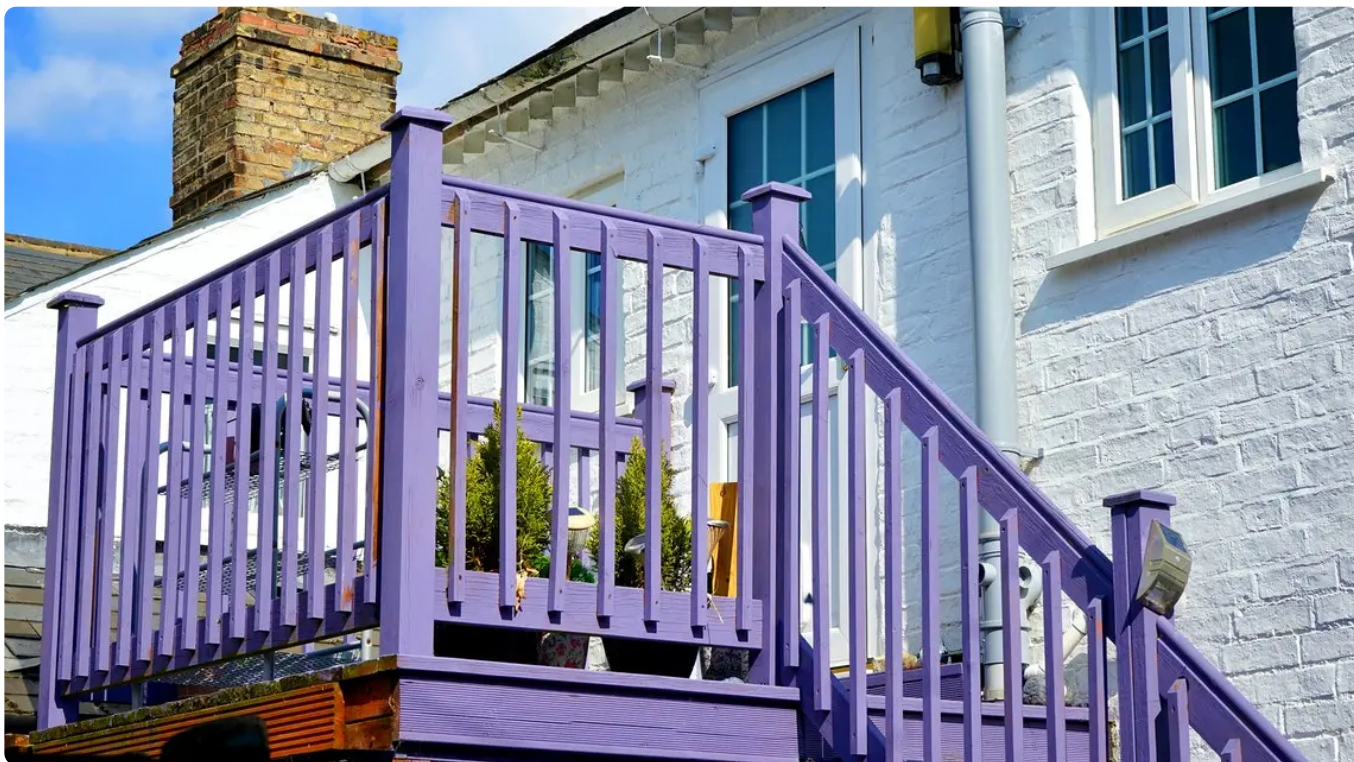


12 | 物理机上程序运行的硬件环境是怎么样？

2021-09-03 宫文学

《手把手带你写一门编程语言》

课程介绍 >

**讲述：宫文学**

时长 20:14 大小 18.54M



你好，我是宫文学。

在经过了儿节课的努力以后，我们的语言运行引擎，从 AST 解释器升级成了 TypeScript 版的虚拟机，又升级成了 C 语言版的虚拟机。这个过程中，我们的语言的性能在不断地提升。并且，我们的关注点，也越来越从高层的语法语义处理层面，往底层技术方向靠拢了。

虽然我们现在的语言特性还不够丰富，但我还是想先带你继续往下钻。我们的目标是先把技术栈钻透，然后再在各个层次上扩大战果。



所以，在接下来的几节课里，我们会把程序编译成汇编代码，然后再生成二进制的可执行程序。在这个过程中，你会把很多过去比较模糊的底层机制搞清楚，我也会带你去除一些

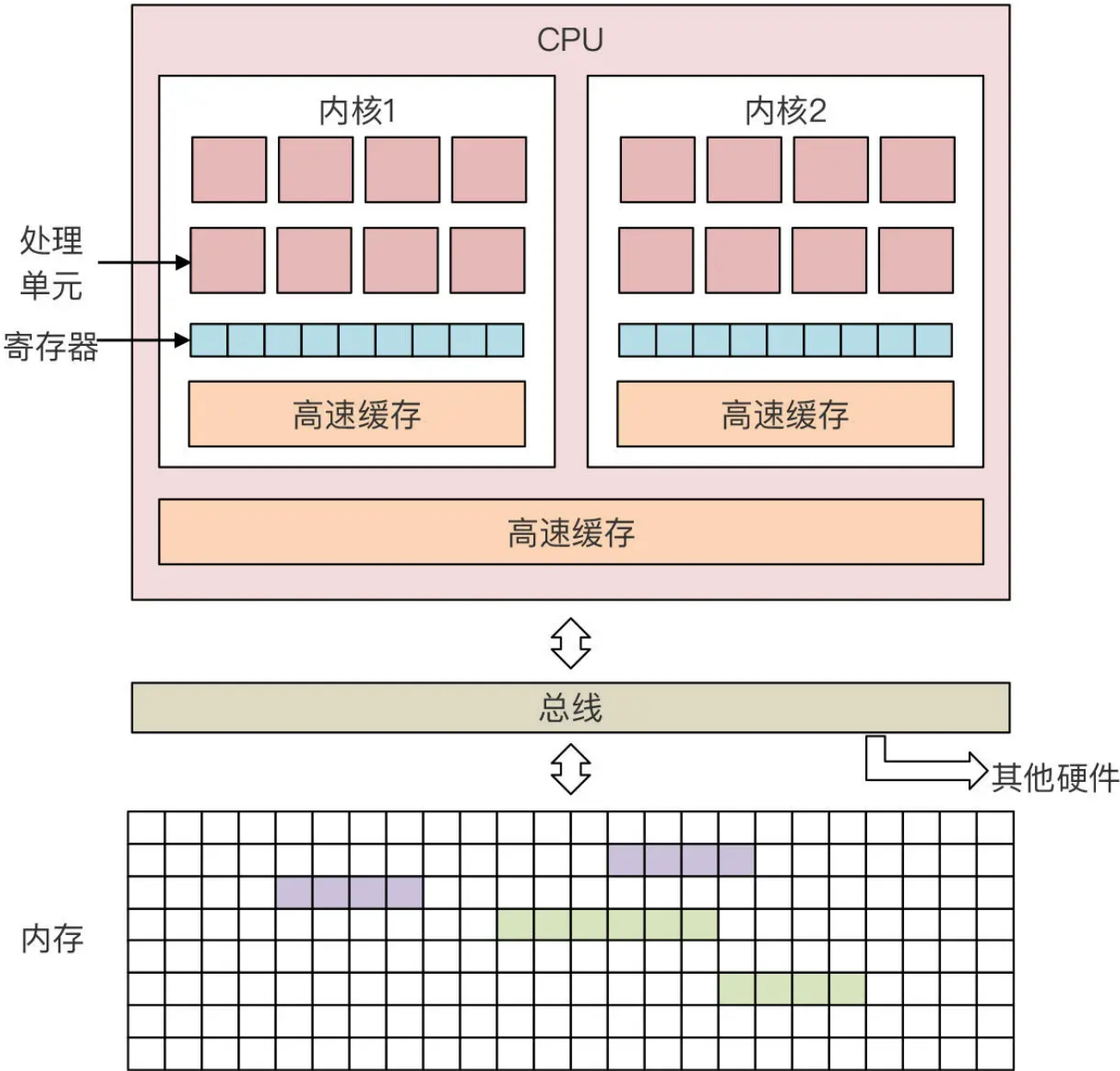
知识点的神秘面纱，让你不再畏惧它们。

在此之前，为了让你编译后的程序能够在计算机上跑起来，**你必须把物理计算机上程序的运行机制搞清楚，特别是要搞清楚应用程序、操作系统和底层硬件的互动关系。**这里面的一些知识点，通常很多程序员都理解得似是而非，不是太透彻。而理解了这些程序运行机制，除了能够让我们的语言在计算机上顺利地运行，还能够帮助你胜任一些系统级软件的开发任务。

今天这节课，我想先带你透彻了解程序运行的硬件环境，以及硬件架构跟我们实现计算机语言的关系。在下节课，我则会带你透彻了解程序运行的软件环境。

硬件环境和程序的运行机制

其实，我们现在用的计算机、手机、物联网等大部分智能设备，它们的硬件架构都是差不多的，基本遵循下面这张图所展示的架构。而这张图上画出来的部分，都是需要我们在实现一门计算机语言的时候需要了解的。



首先我们从整体向部分逐个击破，**先来看看计算机的总体架构和程序运行的原理。**

对于计算机，我们最关心的是两个硬件，一个是 CPU，一个是内存。它们通过计算机的总线连接在一起，这样 CPU 就可以读取内存中的数据 and 程序，并把数据写回内存。而 CPU 内部还会细分成更多的成分，包括高速缓存、寄存器和各种处理单元。

那在这种硬件环境下，程序是怎么运行起来的呢？通常，CPU 上会有个寄存器，叫做 **PC 计数器**。通过 PC 计数器的值，CPU 能够计算出下一条需要执行的代码的地址，然后读取这个代码并执行（根据不同的 CPU 架构，PC 计数器中的值可能不是直接的内存地址，而需要进行一点转换和计算）。通常情况下，程序都是顺序执行的。但当遇到跳转指令时，PC 计数器就会指向新的代码地址，从新的地址开始执行代码。


除了跳转指令会改变 PC 计数器的值，CPU 的异常机制也会改变 PC 计数器的值，跳转到异常处理程序，处理完毕之后再回来。CPU 的异常机制是 CPU 架构设计的一个重要组成部分。典型的异常是由硬件触发的中断。我们每次敲打键盘，都会触发一个中断，处理完毕以后会再接着运行原来的程序。也有的时候，中断可以由软件触发，比如当你 Debug 程序的时候，你可以控制着程序一条一条代码的执行，这也是利用了中断机制。

了解了程序总体的运行原理后，我们再通过一段代码的执行过程，深入了解一下其中的机理，并了解各个硬件成分是如何协同工作的。

一段代码的执行过程

这段示例代码是三条汇编代码，你先看一下：

```
1 movl    -4(%rbp), %eax
2 addl    $10, %eax
3 movl    %eax, -r(%rbp)
```

 复制代码

这三条代码都采用了统一的格式：**操作码的助记符、源操作数和目标操作数**。注意，不同 CPU 的指令集和不同的汇编器，会采用不同的格式，我这里只是举个例子。

这三条代码的意思也很简单，我来解释一下：

第 1 条是一个 `movl` 指令，`movl` 能够把一个整数从一个地方拷贝到另一个地方。这里是从一个内存地址取出一个值，放到 `%eax` 寄存器，而这个内存地址是 `%rbp` 寄存器的值减去 4；

第 2 条是一个 `addl` 指令，它把常数 10 加到 `%eax` 寄存器上；

第 3 条又是一个 `movl` 指令，这次是把 `%eax` 寄存器的值又写回第一行的那个内存地址。

理解了这三条代码的意思以后，我们来看看具体执行的时候都发生了些什么。

第一步，CPU 读入第一行代码。

我们这三条代码都是存在内存里的。CPU 会根据 PC 计数器的值，从内存里把第一条代码读进 CPU 里。

这里你要注意，我们刚才使用了汇编代码来表示程序，但内存里保存的，实际上是机器码。汇编代码通过汇编器可以转换成机器码。

在设计 CPU 的指令集的时候，我们会设计机器码的格式。比如，下图是我在 RISC-V 手册中找到的一张图，描述了 RISC-V 指令的几种编码方式。你能看到，每条指令占用 32 位，也就是一个整数的长度。其中 `opcode` 的意思是操作码，占用低 7 位，`rs` 是源寄存器，`rd` 的意思是目的寄存器，`imm` 是立即数，也就是常数。

31	25 24	20 19	15 14	12 11	7 6	0	
funct7	rs2	rs1	funct3	rd	opcode		R-type
imm[11:0]		rs1	funct3	rd	opcode		I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode		S-type
imm[31:12]				rd	opcode		U-type

来源：The RISC-V Instruction Set Manual Volume I: Unprivileged ISA Document, Version 20190608-Base-Ratified

这些指令被读入内存以后，会有一个解码的过程，也就是把操作码、源操作数、目标操作数这些信息从一条指令里拆解出来，用于后续的处理。这个解码的功能，是由 CPU 内部的一个功能单元完成的。

那么 CPU 是直接从内存中读入代码的吗？

不是的，其实 CPU 是从高速缓存中读入代码和数据的。通常代码和数据的高速缓存是分开的，分别叫做 `Instruction Cache` 和 `Data Cache`。只有高速缓存中没有这些代码或数据的

时候，才会从内存中读取。

高速缓存是内存和 CPU 之间的缓冲区。高速缓存的读写速度比内存快，能够减少 CPU 在读写内存过程中的等待时间。当 CPU 从内存里读一个数据的时候，它其实是从高速缓存中读到的；如果在高速缓存里没有，术语叫做没有命中，CPU 会把这个数据旁白的一批数据都读到高速缓存，这样再读下一个数据的时候，又可以直接从高速缓存中读取了。

高速缓存可能分多级，比如叫做 L1~L3，速度从高到底，容量则反过来，从低到高。并且，一般较低速的缓存是多个核共享的，而更高速的是每个核独享的。

那高速缓存的相关知识对我们实现计算机语言有什么帮助呢？

有一类优化技术，是提高程序数据的局部性，也就是把代码前后需要用到的数据，尽量都聚集在一起，这样便于一次性地加载到高速缓存。在读取下一个数据的时候，就不需要访问内存了，直接从告诉缓存就可以获得了，从而提高了系统的性能。这就是数据局部性的好处。

从这个角度看，你回想一下，上一节课我们就是把栈帧的数据都放在一个连续的内存块里，也是在不经意间提高了数据的局部性。

不过，高速缓存也会带来一些麻烦。比如，当两个内核都去读写同一个内存数据的时候，它们各自使用自己的高速缓存，可能就会出现数据不一致的情况。所以，如果我们在语言层面上支持并发编程的特性，就像 Java 那样，那么在生成指令时就要保证数据的一致性。如果你想具体了解一下这些技术，可以再去看看 [🔗 《编译原理实战课》](#)。

理解了高速缓存以后，我们接着继续看第一条指令的执行过程。在这条指令里，目标操作数，也就是数据加载的目的地是一个寄存器。那我们再了解一下寄存器。

寄存器是 CPU 做运算的操作区。在典型的情况下，CPU 都是把数据加载到寄存器，然后再在寄存器里做各种运算。

相比高速缓存来说，寄存器的读写速度更高，大约是内存的 100 倍。整体来说，寄存器、高速缓存和内存的读写速度是寄存器 > 高速缓存 > 内存。

在 CPU 的设计中，有些寄存器是有特定用途的，比如 PC 计数器用于计算代码地址，EFlags 寄存器用于保存一些运算结果产生的状态等。

还有一些寄存器叫做通用寄存器，它们可以被我们的代码所使用，进行加减乘除等各种计算。在把程序编译成汇编代码的时候，我们要尽量去利用这些通用寄存器来运算。但如果寄存器不够用，就需要临时保存到内存中，把寄存器的空间腾出来。

好，现在我们对寄存器也有了基本的了解了，我们接着往下分析。在第一条指令里，还有一个源操作数，是 `-4(%rbp)`，这代表了一个内存地址。CPU 需要从内存地址里获取数据。

那 CPU 是如何从内存里获取数据的呢？这个过程其实比较复杂，是由多个步骤构成的，并不是一蹴而就的。

首先，CPU 需要计算出内存地址。也就是从 `%rbp` 寄存器中取出现在的值，再减去 4，得到要访问的数据的内存地址。这个地址计算的过程，通常也是由 CPU 内部一个单独的功能模块负责的。

那是不是从这个地址读取数据就行了呢？还不行，因为这个地址可能是个逻辑地址。现代 CPU 一般都有一个 MMU 单元。MMU 是 Memory Management Unit 的缩写，也就是内存管理单元。它提供了虚拟内存管理的功能。也就是说，我们刚才计算出来的地址可能只是个逻辑地址，要经过 MMU 的翻译，才能获得物理的内存地址。

要实现完整的虚拟内存管理功能，还需要操作系统的支持，这个我们在下一节课还会探讨。

那现在，CPU 终于得到了物理内存的地址。那么它会先从高速缓存中读数据，如果高速缓存中没有这个数据，才从内存加载。

你看，一个简单的内存访问功能，竟然涉及到这么多的细节。

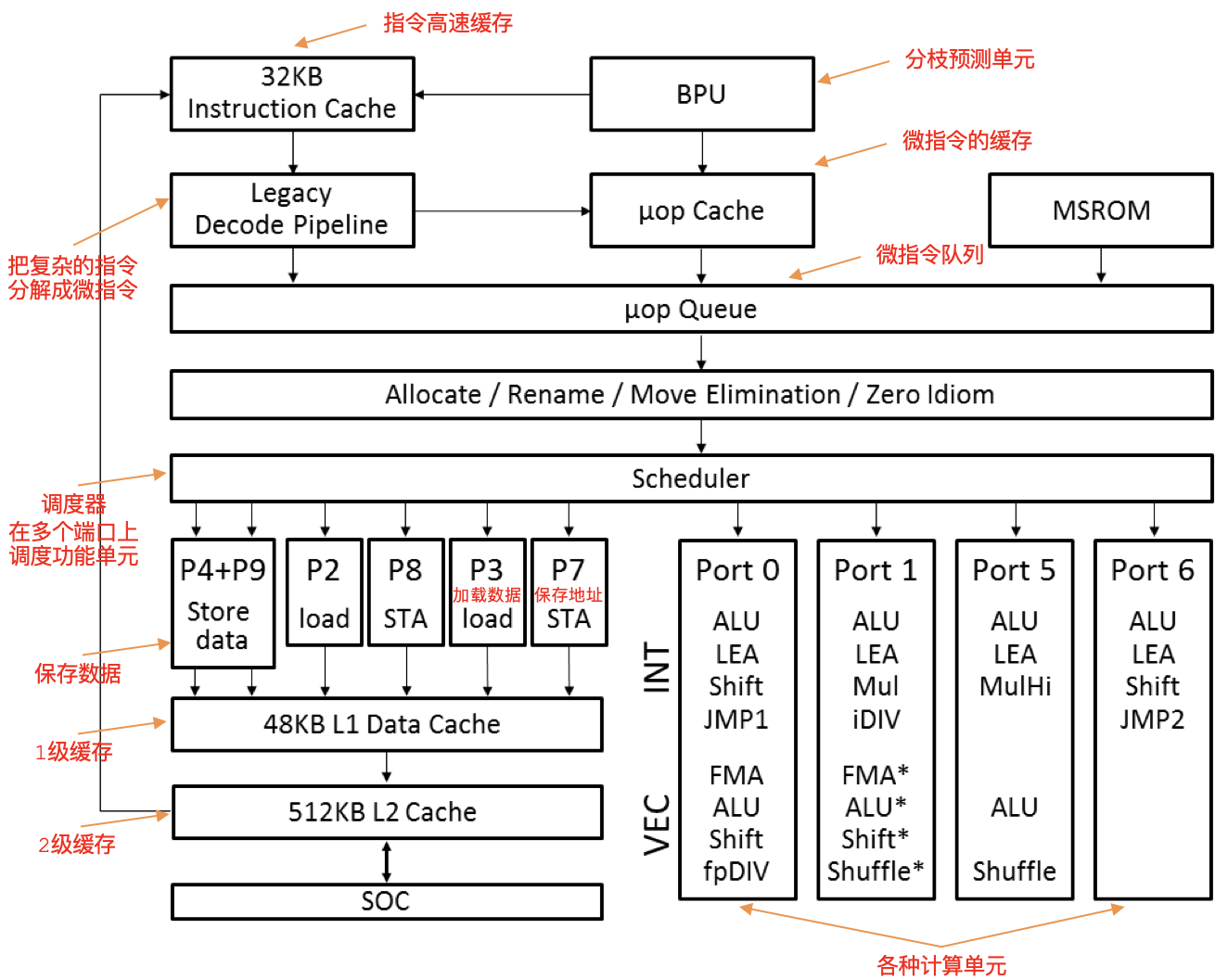
解析完毕第一条指令之后，你大致也能理解第二条、第三条指令是如何执行的了。其中第二条指令，是做了一个加法运算，在这个过程中，会用到 CPU 内部的另一个功能单元：**ALU，也就是算术逻辑运算单元。**

到这里为止，我们已经提到了计算机内部的多个功能单元了，所以我们再把 CPU 内部的功能单元和流水线功能给总结一下。

CPU 内部的功能单元和流水线

对于 CPU 内部的结构，我们已经了解了高速缓存和寄存器。除此之外，CPU 内部还包含了很多的功能单元，每个单元负责不同的功能。比如，有的单元负责获取指令，有的单元负责对指令译码，有的单元负责真正的运算，有的单元负责读取数据，有的单元负责写入数据，等等。

在阅读 CPU 的手册的时候，你会看到关于这个 CPU 的内部结构的一些信息，这个内部结构也被叫做**微架构**。你可以多看看这些图，即使你不能完全理解其中每个单元的含义，这也会帮助你理解 CPU 到底是如何运作的。下面这张图是我从 Intel 的手册中看到的 Ice Lake 型号的 CPU 的微架构的示意图：



来源：图2-1，Intel® 64 and IA-32 Architectures Optimization Reference Manual

我稍微解释一下这个微架构。你会看到，在图的左上角，指令高速缓存中的指令会被解码，解码后变成**微指令**。这里就涉及到了 X86 设计上的一些细节。X86 使用的指令属于复杂指令集（CISC），CISC 会针对特定的功能来设计一些指令，所以指令的执行效率会比较高，就像我们为了某个应用目的专门写一个程序来处理那样。

但复杂指令集也有坏处，就是指令的条数太多了，导致硬件设计会变得复杂，也不容易利用我们下面将要讲到的流水线的优势。所以，其实现代使用 CISC 的 CPU，在内部设计上也借鉴了 RISC 的优点，把复杂的指令拆解成了简单的指令，或者叫做微指令，也就是图中的 uop。

微指令会排成队列去执行任务，它们会到达一个调度器，由调度器调度不同的处理单元去完成不同的任务。调度器通过不同的端口（Port）来调度任务，不同的功能单元则在端口上接收任务。有的单元负责保存数据，有的单元负责加载数据，这些单元都会接到高速缓存上。还有几个端口是专门做计算的。不同的计算任务又分别由不同的计算单元承担，比如 ALU 是做算术运算的，LEA 是做地址运算的，FMA 是做浮点数运算的，等等。

不过，不同的 CPU，其内部功能单元的划分是不同的。但总的来说，在执行一条指令的时候，CPU 内部实际是多个单元按顺序去处理的，这被叫做**指令流水线**。不同 CPU 的流水线设计是不同的，有的分 5 个步骤，有的分成 8 个、10 个甚至更多个步骤。

采用流水线技术最大的好处，就是我们不用等一条指令完全执行完毕，才去执行第二条指令。假设每条指令需要用到 5 个功能单元，分成 5 个步骤。那么在第一条指令的第一个步骤执行完毕以后，第一个功能单元就空出来了，就可以处理第二条指令了。总的来说，相当于有 5 条指令在并行运行。

当然了，实际上的执行过程并没有这么理想，因为不同的指令会用到不同的功能单元。比如上面示例程序的三条指令中，addl 指令用到了 ALU 单元，而其他两条指令就没用到。而且，每个功能单元所需要的时钟周期也是不同的。所以，各条指令在执行过程中就会出现等待的情况。

在编译技术中，专门有一种叫做指令重排序的技术，通过重新排列指令的顺序，在不影响计算结果的情况下，能够让 CPU 的流水线发挥更大的工作效率，如果你想了解更多关于这方面的知识，可以看看 [🔗 《编译原理之美》](#)。

在现代 CPU 的设计中，在硬件层面也提供了乱序执行的功能，这样也可以减少指令的互相等待，提高运行效率。但当 CPU 提前执行后面的一些指令的时候，有可能会产生错误。比如，如果某个指令后面跟着一个分支跳转指令，那这个时候到底要执行哪个分支，要由前面的指令执行结果来决定。

所以你能看到，前面我给的 CPU 微架构图里就会有一个分支预测单元（BPU），尽量争取能准确地预测接下来的分支。如果预测失败，那么 CPU 就会把流水线清空，把已经完成的计算的结果废弃掉。

流水线技术可以看做是指令级的一种并行技术，是一种微观的并行技术。不过，计算机系统还可以通过多处理器、多核和超线程技术来支持并行，现在我们就来介绍一下这部分的内容。

并行和并发的硬件支持

在一台计算机中，你可以安装多颗 CPU，从而支持多个程序并行执行。每颗 CPU 都拥有自己的一套完整的寄存器，各自读取指令并执行，这种技术通常用于服务器。

现代的 CPU 还可以更进一步，在一颗 CPU 中支持多个内核，同样可以并行地运行多个程序。我们现在的 PC 电脑和智能手机的 CPU，基本上都是多核的。

再进一步，有的 CPU 还支持超线程（Hyper Threading）技术，在一个内核上也可以运行两个或多个程序并行执行，每个程序都有一套相互独立的寄存器，基本上互不干扰。

不过，就算一颗 CPU 只有一个内核，在同一时刻只支持运行一个程序，我们仍然可以通过时间片轮转的技术，让这颗 CPU 运行多个任务。而我们前面提到的中断机制，就可以用来定期停止一个正在执行的任务，让 CPU 去执行另一个任务，这种机制叫做并发，而不是并行。

现代语言都支持并行和并发，上述这些硬件机制，就是实现并行和并发的基础。我们下一节课会在操作系统和语言层面讨论更多有关并发和多任务的话题。

课程小结

好了，对程序运行的硬件环境的介绍，我们就先到这里。

今天这节课，我们梳理了在物理机上的程序的运行原理，弄清楚这些原理对于我们实现计算机语言非常关键。这里我再跟你强调以下几个要点：

从硬件层面，我们主要关注 CPU 和内存两个硬件。在我们未来生成的汇编代码里，CPU 的寄存器和内存地址会被作为指令的操作数。

总的来说，物理机上的程序的运行原理是：我们只要把机器码放在内存里，并把 PC 计数器指向代码的地址，CPU 就可以执行这些代码了。所以，在编译程序的时候，我们只需要生成顺序排列的汇编码，再编译成顺序排列的机器码，再把这些机器码加载到内存里，并将 PC 技术器指向这段代码的地址，就可以保证程序能够正常执行了。

这里我们要注意，分支（或跳转）指令会修改 PC 计数器的值，让程序跳到另外的地方执行代码。而硬件中断等 CPU 提供的异常机制也会强行打断程序，跳转到其他地方去执行。这些异常机制是实现 BIOS、操作系统的底层调度机制等功能的基础，非常重要。

另外我们还讨论了 CPU 内部的高速缓存、功能单元和流水线机制。这些 CPU 架构的知识会影响到我们如何编译程序。比如：

指令重排序算法会利用 CPU 的流水线机制提高指令级并发的性能；

实现数据的局部化会提升高速缓存的命中率，减少内存读写；

由于高速缓存中的数据 and 内存中的数据可能存在不一致的情况，因此我们在为并发程序生成代码的时候，要提供一定的机制来保证数据的一致性。

最后，我特别强调，**如果你想要深入了解计算机硬件架构的知识，一定要养成阅读 CPU 手册的习惯**，这些手册你都很容易在厂商的官网上找到。

在后面的课程中，我们也会在自己动手生成汇编代码的过程中，加深对这些知识点的理解。

思考题

在讨论程序的运行机制以及后面有关汇编语言的课程中，我都会不断跟你强调要养成查阅手册的习惯。很多看似难以理解的问题，其实一查手册你就明白了。


那今天的思考题，我其实就是让你练习一下查手册。请你找一下，Intel CPU 或其他 CPU 中，各级高速缓存访问速度是多少？我在这节课的末尾，放了一些手册的链接，你一定要下载查阅一下。

感谢你和我一起学习，也欢迎你把这节课分享给更多对物理计算机的程序运行机制感兴趣的朋友。我是宫文学，我们下节课见。

资源链接

1. [Intel® 64 and IA-32 Architectures Optimization Reference Manual](#)。这本手册描述了 CPU 的很多内部细节。要为 Intel CPU 做编译优化的话，一定需要阅读这本手册。
2. [Intel® 64 and IA-32 architectures software developer's manual volume 1: Basic architecture](#)。这是 Intel CPU 开发者手册的第一卷，属于必读的内容。特别要读一下第 3 章：基础执行环境。
3. 这里有下载 Intel CPU [各种手册的目录](#)。

分享给需要的人，Ta 订阅后你可得 **20 元现金奖励**

 赞 0  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 11 | 基于C语言的虚拟机（二）：性能增长10倍的秘密

下一篇 13 | 物理机上程序运行的软件环境是怎么样？

精选留言

 写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。