现代微处理器
90 分钟指南！

简短、轻松、快节奏地介绍现代处理器微架构的主要设计方面。

今天的机器人非常原始，只能理解
一些简单的指令，如“向左走”、“向右走”和“造车”。

— 约翰·斯拉德克

作者Jason Robert Carey Patterson， 最后更新于 2016 年 8 月（原文 2001 年 2 月）

目录

警告：本文旨在非正式且有趣！

好的，所以你是一名CS毕业生，你做了一个硬件课程作为学位的一部分，但也许那是几年前的事了，从那时起你还没有真正跟上处理器设计的细节。

特别是，您可能不知道最近迅速发展的一些关键主题……

- 流水线（超标量、OOO、VLIW、分支预测、预测）
- 多核和同步多线程（SMT，超线程）
- SIMD 矢量指令（MMX/SSE/AVX、AltiVec、NEON）
- 缓存和内存层次结构

不要害怕！本文将帮助您*快速*上手。很快，您将像专业人士一样讨论有序与无序、超线程、多核和缓存组织的更精细点。

但要做好准备——这篇文章简短而中肯。它没有出拳，节奏非常激烈（真的）。让我们进入它...

不仅仅是兆赫兹

必须澄清的第一个问题是时钟速度和处理器性能之间的差异。*它们不是一回事。*看看几年前（1990 年代后期）处理器的结果……

		SPECint95	*规格fp95*
195兆赫	MIPS R10000	11.0	17.0
400兆赫	阿尔法 21164	12.3	17.2
300兆赫	UltraSPARC	12.1	15.5
300兆赫	奔腾二代	11.6	8.8
300兆赫	PowerPC G3	14.8	11.4
135兆赫	电源2	6.2	17.6

表 1 – 大约 1997 年的处理器性能。

200 MHzMIPS R10000，300 MHz UltraSPARC和400 MHzAlpha 21164在运行大多数程序时的速度大致相同，但它们的时钟速度相差两倍。300 MHz奔腾 II在许多方面的速度也大致相同，但对于科学数字运算等浮点代码，它的速度大约是它的一半。对于普通整数代码，相同300 MHz的 PowerPC G3比其他PowerPC G3快一些，但仍然比浮点数的前三名慢得多。在另一个极端，只有135 MHz

的IBM POWER2处理器在浮点速度上与400 MHz Alpha 21164相当，但对于普通整数程序来说，速度只有400 MHz的一半。

这怎么可能？显然，它不仅仅是时钟速度 - 它完全取决于每个时钟周期中完成了多少工作。这导致...

流水线和指令级并行性

指令在处理器内一个接一个地执行，对吧？好吧，这很容易理解，但事实并非如此。事实上，自 1980 年代中期以来，这种情况就没有发生过。相反，多个指令同时*部分执行*。

考虑一条指令是如何执行的——首先获取指令，然后解码，然后由适当的功能单元执行，最后将结果写入到位。使用这种方案，一个简单的处理器可能需要每条指令4 个周期 （CPI = 4）...

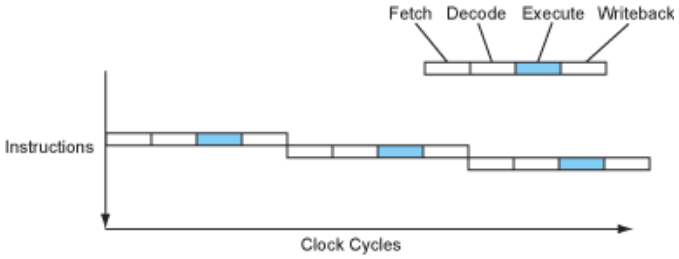


图 1 – 顺序处理器的指令流。

现代处理器在*管道*中重叠这些阶段，就像装配线一样。当一条指令正在执行时，下一条指令正在被解码，而之后的一条指令正在被获取.....

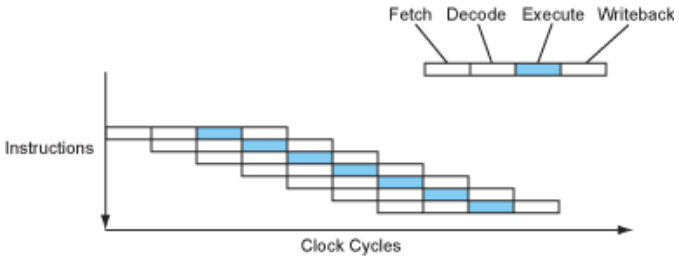


图 2 – 流水线处理器的指令流。

现在处理器每个周期完成 1 条指令（CPI = 1）。这是四倍的加速，根本不改变时钟速度。还不错吧？

从硬件的角度来看，每个流水线级都由一些组合逻辑组成，并可能访问寄存器集和/或某种形式的高速高速缓存。管道级由闩锁隔开。公共时钟信号在每个级之间同步锁存器，以便所有锁存器同时捕获流水线级产生的结果。实际上，时钟将指令"泵送"到管道中。

在每个时钟周期开始时，部分处理指令的数据和控制信息保存在流水线锁存器中，该信息构成下一个流水线级逻辑电路的输入。在时钟周期内，信号通过级的组合逻辑传播，及时产生输出，以便在时钟周期结束时被下一个流水线锁存器捕获...

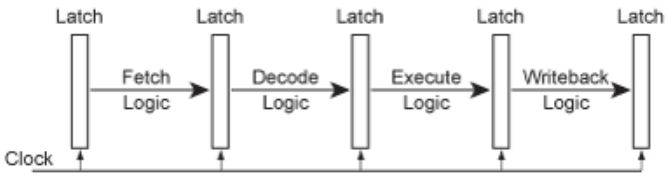


图 3 – 流水线微架构。

由于每条指令的结果在执行阶段完成后可用，因此下一条指令应该能够立即使用该值，而不是等待该结果在写回阶段提交到其目标寄存器。为了允许这一点，添加了称为*旁路*的转发线，沿着管道向后.....

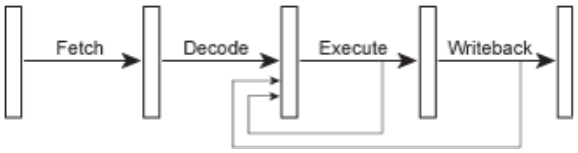


图 4 – 带有旁路的流水线微架构。

虽然流水线阶段看起来很简单，但重要的是要记住，特别是执行阶段实际上由几组不同的逻辑组（几组门）组成，为处理器必须能够*执行*的每种类型的操作组成不同的*功能单元*……

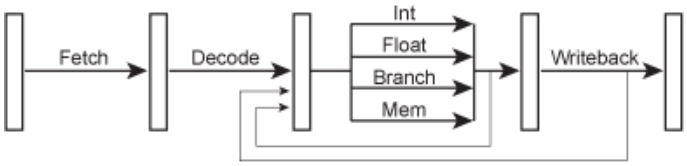


图 5 – 更详细的流水线微体系结构。

早期的RISC处理器，如IBM的801研究原型，MIPS R2000（基于斯坦福MIPS机器）和最初的SPARC（源自伯克利RISC项目），都实现了一个简单的5级流水线，与上面显示的没有什么不同。与此同时，主流的80386、68030和VAX CISC处理器在很大程度上是按顺序工作的——流水线RISC要容易得多，因为它减少了指令集，这意味着指令大多是简单的寄存器到寄存器操作，不像x86、68k或VAX的复杂*指令集*。因此，以20 MHz运行的流水线 SPARC 比以33 MHz 运行的顺序 386 要快得多。从那时起，每个处理器都已流水线化，至少在某种程度上是这样。对原始RISC研究项目的良好总结可以在David Patterson的1985年CACM文章中找到。

更深的管道 – 超级流水线

由于时钟速度受到（除其他外）流水线中最长、最慢级的长度的限制，因此可以*细分*构成每个级的逻辑门，尤其是较长的逻辑门，将流水线转换为具有更多较短级的更深的超级流水线。那么整个处理器就可以以*更高的时钟速度运行！*当然，现在每条指令需要更多周期才能完成（延迟），但处理器仍然会每周期完成1 条指令（吞吐量），并且每秒会有更多的周期，所以处理器每秒会完成更多的指令（实际性能）……

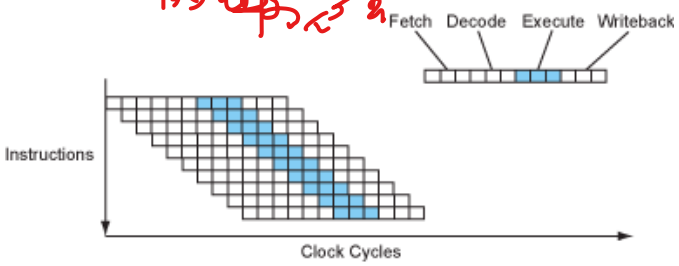


图 6 – 超流水线处理器的指令流。

Alpha 架构师特别喜欢这个想法，这就是为什么早期的 Alpha 拥有深管道并在他们的时代以如此高的时钟速度运行的原因。今天，现代处理器努力将每个流水线阶段的门延迟数量减少到少数几个，大约12-25个门深（不是全部！）加上另外3-5个门用于锁存器本身，而且大多数都有相当深的流水线……

管道深度	*处理器*
6	超斯帕克 T1
7	PowerPC G4e
8	UltraSPARC T2 / T3，Cortex-A9
10	速龙， 蝎子
11	克雷特
12	奔腾临/II/III，速龙64/飞鸿，苹果A6
13	丹佛
14	UltraSPARC III/IV，酷睿2，苹果A7 / A8
14/19	Core i*2/i*3 Sandy/Ivy Bridge，Core i*4/i*5Haswell/Broadwell
15	皮质-A15/A57
16	PowerPC G5，Core i*1Nehalem
18	推土机/打桩机，压路机
20	奔腾 4
31	奔腾 4E 普雷斯科特

表 2 – 常见处理器的管道深度。

x86 处理器通常比 RISC（可比时代）具有更深的管道，因为它们需要做额外的工作来解码复杂的 x86 指令（稍后会详细介绍）。UltraSPARC T1/T2/T3 Niagara是深流水线趋势的最近一个例外——UltraSPARC T1只有6个，T2/T3只有8个，以保持这些核心尽可能小（稍后也会详细介绍）。

多问题 – 超标量

由于管道的执行阶段实际上是一堆不同的功能单元，每个功能单元都在执行自己的任务，因此尝试*并行*执行多个指令似乎很诱人，每个指令都在自己的*功能单元*中。为此，必须增强获取和解码/调度阶段，以便它们可以并行解码多个指令并将它们发送到"执行资源"……

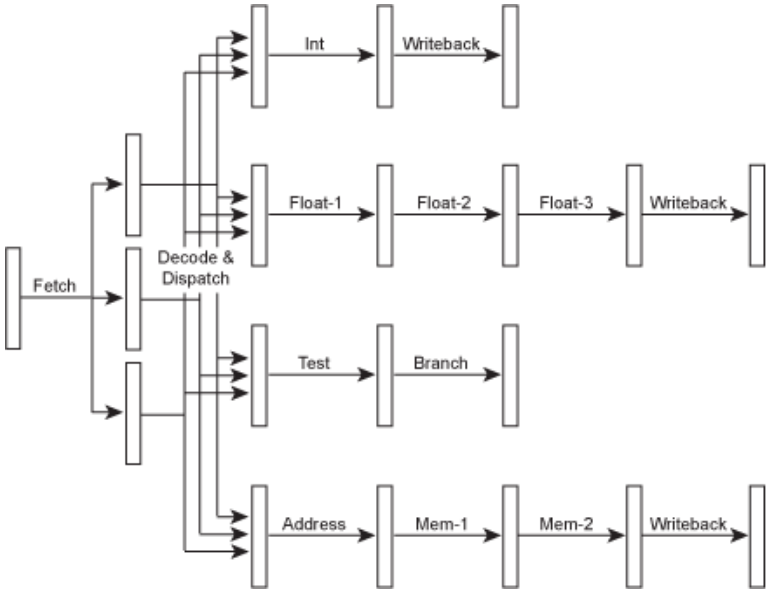


图 7 – 超标量微架构。

当然，现在每个功能单元都有独立的管道，它们甚至可以具有不同数量的阶段。这使得更简单的指令可以更快地完成，*减少延迟*（我们很快就会谈到）。由于此类处理器具有许多不同的管道深度，因此在执行*整数*指令时通常引用处理器管道的深度，这通常是可能的管道路径中最短的，内存和浮点管道暗示具有一些额外的阶段。因此，具有"10 级流水线"的处理器将使用 10 个级来执行整数指令，也许使用 12

或 13 个级来执行内存指令，也许使用 14 或 15 级来执行浮点指令。在各种管道内部和之间还有一堆旁路，但为了简单起见，这些都省略在了图中。

在上面的示例中，处理器可能每个周期发出 3 条不同的指令，例如 1 条整数、1 条浮点和 1 条内存指令。可以添加更多的功能单元，以便处理器可能能够在每个周期执行 2 条整数指令，或 2 条浮点指令，或任何目标应用程序可以最好使用的内容。

在超标量处理器上，指令流看起来像...

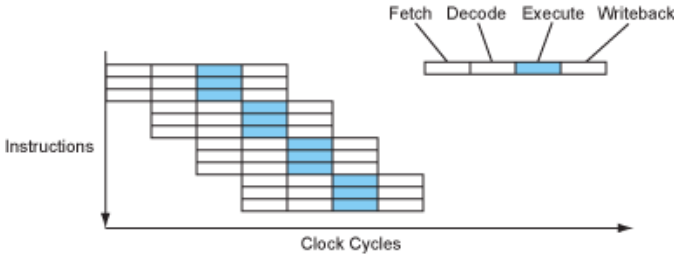


图 8 – 超标量处理器的指令流。

这太好了！现在每个周期有 3 条指令完成（CPI = 0.33，或IPC = 3，也写为ILP = 3表示*指令级并行性*）。每个周期能够发出、执行或完成的指令数称为处理器的*宽度*。

请注意，问题宽度小于功能单元的数量 - 这是典型的。必须有更多的功能单元，因为不同的代码序列具有不同的指令组合。这个想法是每个周期执行 3 条指令，但这些指令并不总是 1 个整数、1 个浮点和 1 个内存操作，因此需要 3 个以上的功能单元。

IBM POWER1处理器是PowerPC的前身，是第一个主流的超标量处理器。大多数RISC在不久之后就变成了超标量（SuperSPARC，Alpha 21064）。英特尔甚至设法构建了一个超标量x86 - 最初的奔腾 - 但是复杂的x86指令集对他们来说是一个真正的问题（稍后会详细介绍）。

当然，没有什么能阻止处理器同时存在深度流水线和多指令问题，因此它可以同时是超流水线和超标量.....

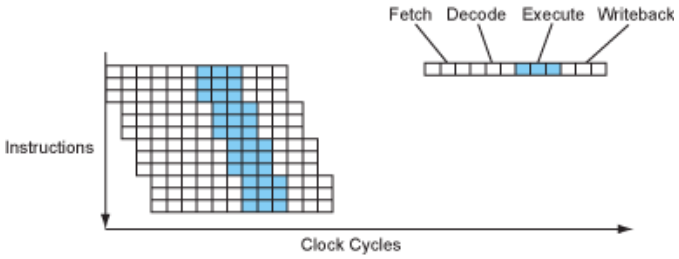


图 9 – 超流水线超标量处理器的指令流。

今天，几乎每个处理器都是超流水线超标量，所以它们简称为超标量。严格来说，超级流水线只是用更深的管道流水线。

现代处理器的宽度差异很大...

发行宽度	*处理器*
1	超斯帕克 T1
2	UltraSPARC T2/T3， Scorpion，Cortex-A9
3	奔腾临/II/III/M，奔腾4，Krait，苹果A6，Cortex-A15/A57
4	UltraSPARC III/IV，PowerPC G4e
4/8	推土机/打桩机，压路机
5	PowerPC G5
6	速龙，速龙64/飞龙，酷睿2，酷睿i*1尼哈勒姆，酷睿i*2/i*3桑迪/常春藤桥，苹果A7/A8
7	丹佛
8	Core i*4/i*5 Haswell/Broadwell

表 3 – 常见处理器的问题宽度。

每个处理器中功能单元的确切数量和类型取决于其目标市场。一些处理器有更多的浮点执行资源（IBM的POWER系列），其他处理器更偏整数（Pentium Pro/II/III/M），一些处理器将大部分资源用于SIMD矢量指令（PowerPC G4/G4e），而大多数处理器试图采取"平衡"的中间立场。

显式并行性 – VLIW

在向后兼容性不是问题的情况下，*可以将指令集*本身设计为*显式*分组要并行执行的指令。这种方法消除了在调度阶段对复杂的依赖关系检查逻辑的需求，这将使处理器更容易设计，更小，并且更容易随着时间的推移提高时钟速度（至少在理论上）。

在这种风格的处理器中，"指令*"实际上是一组*小的子指令，因此指令本身很长，通常是128 位或更多，因此得名 VLIW – *非常长的指令字*。每条指令都包含多个并行操作的信息。

VLIW处理器的指令流很像超标量，除了解码/调度阶段要简单得多，并且只发生在每组子指令上……

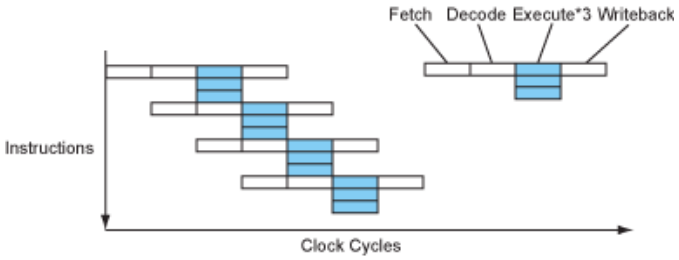


图 10 – VLIW 处理器的指令流。

除了简化调度逻辑之外，VLIW 处理器很像超标量处理器。从编译器的角度来看尤其如此（稍后会详细介绍）。

然而，值得注意的是，大多数VLIW设计*不是互锁*的。这意味着它们不会检查指令之间的依赖关系，并且除了在缓存未命中时使整个处理器停止之外，通常无法停止指令。因此，编译器需要在相关指令之间插入适当数量的循环，即使没有指令来填补空白，必要时也可以使用*nops*（no-operations，发音为"no ops"）。这在一定程度上使编译器复杂化，因为它正在执行超标量处理器通常在运行时执行的操作，但是编译器中的额外代码最少，并且节省了处理器芯片上的宝贵资源。

目前还没有VLIW设计作为主流CPU在商业上取得成功，但是英特尔的IA-64架构仍在以Itanium处理器的形式生产，曾经打算取代x86。英特尔选择将IA-64称为"EPIC"设计，用于"显式并行指令计算"，但它本质上是一个具有巧妙分组（允许长期兼容性）和预测（见下文）的VLIW。图形处理器（GPU）中的可编程着色器有时是VLIW设计，许多数字信号处理器（DSP）也是如此，还有Transmeta（请参阅即将推出的x86部分）。

指令依赖关系和延迟

流水线和多个问题可以走多远？如果 5级管道快5 倍，为什么不建造20 级超级管道呢？如果4 期超标量很好，为什么不选择8 期呢？就此而言，为什么不构建一个具有50 级流水线的处理器，每个周期发出 20 条指令？

好吧，请考虑以下两个说明...

```
a = b * c;
d = a + 1;
```

第二条指令*取决于*第一条指令——在第一条指令完成计算结果之前，处理器无法执行第二条指令。这是一个严重的问题，因为相互依赖的指令无法并行执行。因此，在这种情况下，多个问题是不可能的。

如果第一条指令是简单的整数加法，那么这在流水线*单问题*处理器中可能仍然可以，因为整数加法很快，并且第一条指令的结果将及时可用，以将其反馈到下一条指令中（使用旁路）。但是，在乘法的情况下，需要几个周期才能完成，当第二条指令在一个周期后到达执行阶段时，第一个指令的结果将不可用。因此，处理器将需要停止第二条指令的执行，直到其数据可用，从而在管道中插入一个*气泡*，在那里没有工作完成。

从指令到达执行阶段到其结果可供其他指令使用之间的周期数称为指令的*延迟*。管道越深，阶段越多，因此延迟越长。因此，非常深的管道并不比短管道有效多少，因为由于所有这些令人讨厌的指令相互依赖，深管道只会充满气泡。

从编译器的角度来看，现代处理器中的典型延迟范围从整数运算的单个周期到浮点加法的大约3-6个周期，乘法的相同或可能稍长，再到整数除法的十几个周期。

内存加载的延迟特别麻烦，部分原因是它们往往发生在代码序列的早期，这使得很难用有用的指令来填充它们的延迟，同样重要的是因为它们有点不可预测——加载延迟变化很大，具体取决于访问是否是缓存命中（我们稍后会谈到缓存）。

> 当“延迟”一词用于相关但不同的含义时，可能会令人困惑。在这里，我说的是编译器看到的延迟。一些硬件工程师可能会将延迟视为执行所需的周期数（管道阶段数）。因此，硬件工程师可能会说简单整数管道中的指令延迟为 5，但吞吐量为 1，而从编译器的角度来看，它们的延迟为 1，因为它们的结果可用于下一个周期。编译器视图更为常见，甚至在硬件手册中也普遍使用。

分支和分支预测

流水线的另一个关键问题是分支。请考虑以下代码序列...

```
if (a > 7) {
    b = c;
} else {
    b = d;
}
```

...编译成类似...

```
    cmp a, 7    ; a > 7 ?
    ble L1
    mov c, b    ; b = c
    br L2
L1: mov d, b    ; b = d
L2: ...
```

现在考虑一个执行此代码序列的流水线处理器。当第 2 行的条件分支到达管道中的执行阶段时，处理器必须已经获取并解码了接下来的几条指令。但是*哪些*指令呢？它应该获取并解码*if*分支（第 3 行和第 4 行）还是*else*分支（第 5 行）？在条件分支进入执行阶段之前，它不会真正知道，而是在可能需要几个周期的深度管道处理器中。而且它不能只是等待 - 处理器平均每六条指令遇到一个分支，如果它在每个分支上等待几个周期，那么首先使用流水线获得的大部分性能都将丢失。

所以处理器必须做出*猜测*。然后，处理器将获取它猜测的路径，并*推测性地*开始执行这些指令。当然，在知道分支的结果之前，它将无法实际提交（写回）这些指令。更糟糕的是，如果猜测错误，指令将不得不取消，这些周期将被浪费。但如果猜测正确，处理器将能够继续全速运行。

关键问题是处理器*应该如何*进行猜测。两个选择浮现在脑海中。首先，*编译器*可能能够标记分支以告诉处理器要走哪条路。这称为*静态分支预测*。如果指令格式中有一点来编码预测，那将是理想的，但对于较旧的体系结构，这不是一个选项，因此可以使用约定来代替，例如预测采用向后分支，而预测不采用前向分支。然而，更重要的是，这种方法要求编译器非常聪明，以便做出正确的猜测，这对于循环来说很容易，但对于其他分支来说可能很难。

另一种选择是让处理器在*运行时*进行猜测。通常，这是通过使用片上分支预测表来完成的，该表包含最近分支的地址以及指示上次是否占用每个分支的位。实际上，大多数处理器实际上使用两位，因此单个未采取的事件不会逆转通常采取的预测（对于环路后边缘很重要）。当然，这个动态分支预测表占用了处理器芯片上的宝贵空间，但分支预测非常重要，非常值得。

不幸的是，即使是最好的分支预测技术有时也是错误的，并且对于深度管道，可能需要取消许多指令。这被称为*错误预测惩罚*。奔腾Pro/II/III就是一个很好的例子——它有一个12级的流水线，因此错误地预测了10-15个周期的惩罚。即使有一个聪明的动态分支预测器，可以正确预测90%的时间，这种高错误预测的惩罚意味着奔腾Pro/II/III大约30%的性能由于预测错误而损失。换句话说，三分之一的时间奔腾Pro/II/III并没有做有用的工作，而是在说"哎呀，错误的方式"。

现代处理器将越来越多的硬件用于分支预测，试图进一步提高预测精度，并降低成本。许多人不仅孤立地记录每个分支的方向，而且在通往它的几个分支的上下文中记录，这被称为*两级自适应*预测器。有些保留更全局的分支历史记录，而不是每个分支的单独历史记录，以尝试检测分支之间的任何相关性，即使它们在代码中相对较远。这称为*gshare*或*gselect*预测因子。最先进的现代处理器通常会实现*多个*分支预测器，并根据哪个预测因子似乎最适合每个分支在它们之间进行选择！

尽管如此，即使是最好的现代处理器，拥有最好、最聪明的分支预测器，也只能达到大约 95% 的预测准确率，并且由于分支预测错误，仍然会损失相当多的性能。底线很简单——非常深的管道自然会受到*收益递减*的影响，因为管道越深，你必须试图预测的未来越远，你就越有可能出错，当你犯错时，错误预测的惩罚就越大。

使用谓词消除分支

条件分支是如此成问题，以至于完全消除它们会很好。显然，如果语句不能从编程语言中消除，*那么*结果的分支怎么可能被消除呢？答案在于某些分支的使用方式。

再次考虑上面的例子。在五个指令中，两个是分支，其中一个是无条件分支。如果可以以某种方式标记*mov*指令以告诉它们仅在某些情况下执行，则可以简化代码.....

```
cmp a, 7        ; a > 7 ?
mov c, b        ; b = c
cmovle d, b     ; if le, then b = d
```

在这里，引入了一条名为*cmovle*的新指令，用于"小于或等于则有条件的移动"。此指令通过正常执行来工作，但仅在其条件为 true 时才提交自身。这称为谓词指令，因为它的执行由谓*词*控制（真/假测试）。

鉴于这个新的谓词移动指令，代码中已经删除了两条指令，并且都是昂贵的分支。此外，通过聪明地总是做第一个*mov*然后在必要时覆盖它，代码的并行性也得到了提高——第 1 行和第 2 行现在可以并行执行，从而实现 50% 的加速（2 个周期而不是 3 个周期）。然而，最重要的是，已经消除了分支预测错误并遭受大量错误预测惩罚的可能性。

当然，如果*if*和*else*情况下的代码块更长，那么使用谓词意味着执行比使用分支更多的指令，因为处理器正在通过代码有效地执行*两条路径*。是否值得执行更多指令以避免分支是一个棘手的决定 - 对于非常小或非常大的块，决策很简单，但对于中型块，优化器必须考虑复杂的权衡。

Alpha 架构从一开始就有一个有条件的移动指令。MIPS，SPARC和x86后来添加了它。在IA-64中，英特尔全力以赴，几乎每条指令都是基于预测的，希望大大减少内部循环中的分支问题，尤其是那些分支

不可预测的问题，如编译器和操作系统内核。有趣的是，许多手机和平板电脑中使用的ARM架构是第一个具有完全谓词指令集的架构。考虑到早期的ARM处理器只有较短的管道，因此错误预测的惩罚相对较小，这一点更加有趣。

指令调度，寄存器重命名和OOO

如果分支和长延迟指令会在管道中引起气泡，那么也许这些空循环可用于执行其他工作。为此，必须对程序中的指令*重新排序*，以便在一条指令等待时，可以执行其他指令。例如，可以在程序的更下方找到其他几个指令，并将它们放在前面的乘法示例中的两个指令之间。

有两种方法可以做到这一点。一种方法是在运行时在硬件中进行重新排序。在处理器中进行动态指令调度（重新排序）意味着必须增强调度逻辑以查看指令组并尽可能无序地*调度*它们，以使用处理器的功能单元。毫不奇怪，这被称为*无序执行*，或者简称为 OOO（有时写为 OoO 或 OOE）。

如果处理器要无序执行指令，则需要记住这些指令之间的依赖关系。通过不处理原始架构定义的寄存器，而是使用一组*重命名*的寄存器，可以更轻松地执行此操作。例如，将寄存器存储到内存中，然后将其他一些存储器加载到同一寄存器中，表示不同的*值*，不需要进入同一物理寄存器。此外，如果将这些不同的指令映射到不同的物理寄存器，则可以并行执行，这就是OOO执行的重点。因此，处理器必须随时保持动态指令及其使用的物理寄存器的映射。此过程称为*寄存器重命名*。作为额外的好处，可以使用一组可能更大的真实寄存器，以尝试从代码中提取更多的并行性。

所有这些依赖关系分析、寄存器重命名和 OOO 执行都给处理器增加了许多复杂的逻辑，使其更难设计、芯片面积更大、更耗电。额外的逻辑特别耗电，因为这些晶体管*总是*在工作，不像功能单元至少花费一些时间空闲（甚至可能断电）。另一方面，乱序执行的优势在于，软件不需要重新编译即可获得新处理器设计的一些好处，尽管通常不是全部。

解决整个问题的另一种方法是让*编译器*通过重新排列指令来优化代码。这称为*静态*或编译时指令调度。然后，可以将重新排列的指令流馈送到具有更简单*的顺序*多问题逻辑的处理器，依靠编译器以最佳指令流"勺子馈送"处理器。避免复杂的OOO逻辑应该使处理器更容易设计，更少耗电和更小，这意味着更多的内核或额外的缓存可以放置在相同数量的芯片区域（稍后会详细介绍）。

与 OOO 硬件相比，编译器方法还具有其他一些优势 - 它可以比硬件更深入地查看程序，并且可以推测多个路径而不仅仅是一条路径，如果分支不可预测，这是一个大问题。另一方面，不能期望编译器是通灵的，所以它不一定总是让一切都完美。如果没有 OOO 硬件，当编译器无法预测缓存未命中等情况时，管道将停止。

大多数早期的超标量都是有序设计（SuperSPARC，hyperSPARC，
UltraSPARC，Alpha 21064&21164，最初的奔腾）。早期OOO设计的例子包
括MIPS R10000，Alpha 21264，在某种程度上还有整个POWER/PowerPC系列（及其预订站）。今天，几乎所有的高性能处理器都是无序设计，UltraSPARC III/IV，POWER6和Denver除外。大多数低功耗、低性能的处理器（如Cortex-A7/A53和 Atom）都是按顺序设计，因为 OOO 逻辑消耗大量功耗，而性能提升相对较小。

布莱尼亚克辩论

必须问的一个问题是，昂贵的无序逻辑是否真的有必要，或者编译器是否可以在没有它的情况下足够好地完成指令调度任务。这在历史上被称为*布莱尼亚克*与*速度恶魔*的辩论。这种简单（且有趣）的设计风格分类首次出现在Linley Gwennap的1993年微处理器报告社论中，并由Dileep Bhandarkar的Alpha Implementations & Architecture一书广为人知。

*Brainiac*设计处于智能机器的末端，许多 OOO 硬件试图从代码中挤出指令级并行性的每一滴，即使花费数百万个逻辑晶体管和多年的设计工作来做到这一点。相比之下，*速度恶魔*的设计更简单、更小，依赖于智能编译器，并愿意牺牲一点指令级并行性来换取简单性带来的其他好处。从历史上看，速度恶魔的设计倾向于以更高的时钟速度运行，正是因为它们更简单，因此被称为"速度恶魔"，但今天情况不再如此，因为时钟速度主要受到功率和热量问题的限制。

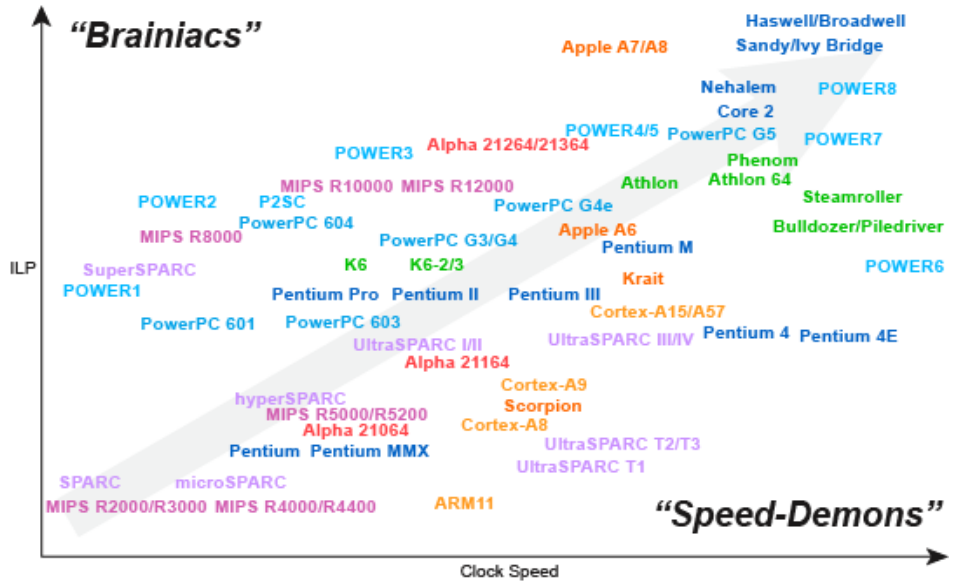
显然，OOO硬件应该能够提取更多的指令级并行性，因为在运行时将知道无法提前预测的事情 - 特别是缓存未命中。另一方面，更简单的顺序设计将更小，功耗更低，这意味着您可以将更多小的顺序内核与

更少、更大的无序内核放置在同一芯片上。您更愿意拥有哪个：4 个强大的 brainiac 核心，还是 8 个更简单的顺序核心?

究竟哪个是更重要的因素，目前尚有争议。总的来说，过去似乎 OOO 执行的好处和成本都被夸大了。在成本方面，调度和寄存器重命名逻辑的适当流水线使OOO处理器在1990年代后期实现了与更简单设计竞争的时钟速度，并且近年来巧妙的工程设计大大降低了OOO执行的功率开销，主要留下芯片面积成本。这证明了处理器架构师的一些杰出工程设计。

然而，不幸的是，OOO执行在动态提取额外的指令级并行性方面的有效性令人失望，只看到了相对较小的改进，可能比等效的顺序设计提高了20-40%左右。引用 Andy Glew 的话，他是乱序执行的先驱，也是Pentium Pro/II/III 的主要架构师之一：“OOO 的肮脏小秘密是，我们通常根本不是 OOO”。乱序执行也无法提供最初希望的“重新编译独立性”程度，即使在激进的 OOO 处理器上，重新编译仍然会产生很大的加速。

当谈到脑残的争论时，许多供应商走上了一条路，然后改变了主意，转向了另一边.....



显示世系： ◉ **无** ○ **x86** ○ **MIPS** ○ **SPARC** ○ **POWER/PowerPC** ○ **Alpha** ○ **ARM x86 & ARM** ○

图 11 – 布莱尼亚克 vs 速度恶魔。

例如，DEC在前两代阿尔法中主要变成了速度恶魔，然后在第三代改为brainiac。MIPS也做了类似的事情。另一方面，Sun在他们的第一个超标量SPARC中进行了脑残，然后在最近的设计中转向了速度恶魔。多年来（直到最近），POWER/PowerPC阵营也逐渐远离了brainiac设计，尽管所有POWER/PowerPC设计中的预订站确实在不同功能单元之间提供了一定程度的OOO执行，即使每个功能单元队列中的指令严格按顺序执行。相比之下，ARM处理器已经显示出向更聪明的设计的持续发展，它们来自低功耗，低性能的嵌入式世界，但仍以移动为中心，因此无法将时钟速度推得太高。

英特尔一直是最值得关注的。由于 x86 架构的限制，现代 x86 处理器别无选择，只能至少有点脑残（稍后会详细介绍），而奔腾 Pro全心全意地接受了这种情绪。然后，与AMD的竞争接踵而至，达到1 GHz，AMD在2000年3月以一杆之差获胜。英特尔不惜一切代价将重点转移到时钟速度上，使奔腾 4尽可能像解耦的 x86 微架构的速度恶魔，牺牲了一些 ILP，并使用深度20级流水线通过 2 GHz 然后是 3 GHz，后来的修订版具有惊人的31 级流水线，最高可达3.8 GHz。与此同时，对于IA-64Itanium（上面未显示），英特尔再次坚定地押注于智能编译器方法，其简单的设计完全依赖于静态的编译时调度。面对IA-64 的失败、奔腾 4 的巨大功率和散热问题，以及 AMD 时钟较慢的 Athlon 处理器在2 GHz范围内实际上在实际代码上优于奔腾4的事实，英特尔随后再次扭转了其立场，并恢复了旧的奔腾 Pro/II/IIIbrainiac 设计，以生产奔腾 M及其核心继任者， 取得了巨大的成功。

The Power Wall & The ILP Wall

奔腾 4 严重的功率和热量问题表明时钟速度存在限制。事实证明，功耗的增长速度*甚至比*时钟速度还要快——对于任何给定级别的芯片技术，将处理器的时钟速度提高 20%，通常会使其功耗增加更多，甚至可能增加 50%，因为晶体管不仅开关频率提高 20%，而且电压通常也需要增加， 为了更快地驱动信号通过电路以可靠地满足较短的时序要求，当然，假设电路以更高的速度工作。虽然功率随时钟频率线性增加，但它随着电压*的平方*而增加，在非常高的时钟速度（f*V*V）下会产生一种"三重打击"。

情况变得更糟，因为除了正常的开关功率外，还有少量*的漏*电功率，因为即使晶体管关闭，流过它的电流也不会完全减少到零。就像良好的有用电流一样，这种漏电流也会随着电压的增加而上升。如果这还不够糟糕，泄漏通常会随着温度的升高而上升，这是由于硅内更热，更有能量的电子的运动增加。

最终结果是，今天，将现代处理器的时钟速度提高相对适度的 30% 可以消耗多达两倍的功率，并产生双倍的热量.....

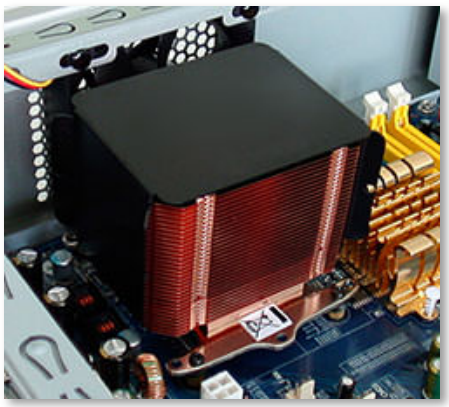


图12 –卸下前风扇的现代台式机处理器的散热器。

在某种程度上，功率的增加是可以的，但是在某个时候，目前大约150-200瓦，功率和热量问题变得无法管理，因为根本不可能以任何实际的方式为硅芯片提供那么多的功率和冷却，即使电路实际上可以以更高的时钟速度运行。这称为*电源墙*。

过于关注时钟速度的处理器，如奔腾4，IBM的POWER6和最近的AMD推土机，很快就撞上了电源墙，发现自己无法将时钟速度推到最初希望的高度，导致它们被时钟速度较慢但*更智能*的处理器击败，这些处理器利用了更多的指令级并行性。

因此，纯粹追求时钟速度并不是最好的策略。当然，对于笔记本电脑、平板电脑和手机等便携式移动设备来说更是如此，由于电池容量的限制和有限的、通常是无风扇的冷却，"专业"笔记本电脑的功率墙要快得多，"专业"笔记本电脑约为 50W，超轻型笔记本电脑约为 15W，平板电脑为 10W，手机的功率低于 5W。

那么，如果主要追求时钟速度是一个问题，那么纯粹的大脑是正确的方法吗？可悲的是，没有。追求越来越多的ILP也有一定的局限性，因为不幸的是，由于负载延迟，缓存未命中，分支和指令之间的依赖关系的组合，普通程序在其中没有很多细粒度的并行性。这种可用指令级并行性的限制称为*ILP 墙*。

过于关注ILP的处理器，如早期的POWER处理器SuperSPARC和MIPS R10000，很快发现它们提取额外指令级并行性的能力只是适度的，而额外的复杂性严重阻碍了它们达到快速时钟速度的能力，导致这些处理器被*更笨*但更高频率的处理器击败，这些处理器并不那么专注于ILP。

4 问题超标量处理器希望在每个周期都有4 条独立的指令可用，并满足其所有依赖关系和延迟。实际上，这几乎是不可能的，尤其是在负载延迟为 3 或4 个周期的情况下。目前，主流单线程应用程序的实际指令级并行性最多限制为每个周期约2-3条指令。事实上，运行 SPECint 基准测试的现代处理器的平均 ILP 每个周期不到 2 条指令，并且 SPEC 基准测试比大多数大型实际应用程序"容易"。某些类型的应用程序确实表现出更多的并行性，例如科学代码，但这些通常不代表主流应用程序。还有一些类型的代码，例如"指针追逐"，即使每个周期维持1 条指令也是极其困难的。对于这些程序，关键问题是内存系统，还有另一堵墙，*内存墙*（我们稍后会谈到）。

x86呢?

那么x86在这一切中的位置在哪里，英特尔和AMD如何在所有这些发展中保持竞争力，尽管架构现在已经超过35年了？

虽然最初的奔腾，一个超标量x86，是一个了不起的工程，很明显最大的问题是复杂而混乱的x86指令集。复杂的寻址模式和最少数量的寄存器意味着由于潜在的依赖关系，很少有指令可以并行执行。为了让x86阵营与RISC架构竞争，他们需要找到一种方法来"绕过"x86指令集。

该解决方案由NexGen和Intel的工程师独立发明（大约在同一时间），是将x86指令*动态解码*为简单的，类似RISC的微指令，然后可以通过快速，RISC风格的寄存器重命名OOO超标量内核执行。微指令通常称为*μops*（发音为"micro-ops"）。大多数 x86 指令解码为 1、2 或3 μops，而更复杂的指令需要更大的数字。

对于这些*解耦*的超标量 x86 处理器，寄存器重命名绝对至关重要，因为 x86 架构在32位模式下只有 8 个寄存器（64 位模式增加了额外的 8 个寄存器）。这与RISC架构有很大不同，RISC架构通过重命名提供更多寄存器的效果不大。尽管如此，通过巧妙的寄存器重命名，完整的RISC技巧可供x86世界使用，除了高级静态指令调度（因为μops隐藏在x86层后面，因此编译器不太可见）和使用大型寄存器集以避免内存访问。

基本方案的工作原理是这样的...

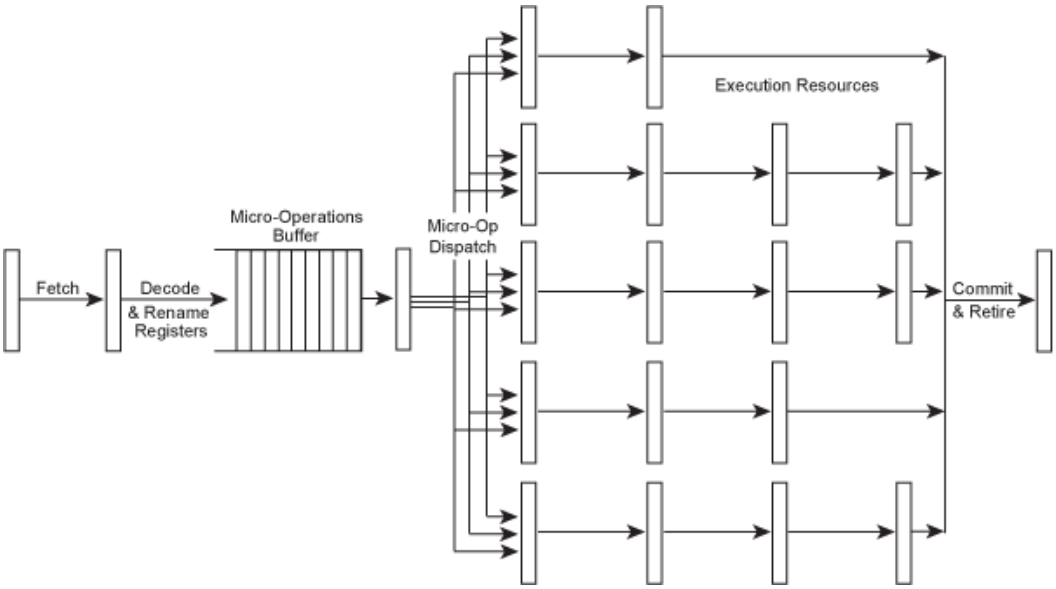


图 13 – "RISCy x86"解耦微架构。

NexGen的Nx586和英特尔的Pentium Pro（也称为P6）是第一个采用解耦x86微架构设计的处理器，今天所有现代x86处理器都使用这种技术。当然，它们的核心流水线、功能单元等的确切设计都有所不同，就像各种RISC处理器一样，但是从x86转换为内部μop类似RISC的指令的基本思想是它们的共同点。

一些较新的x86处理器甚至将转换后的μops存储在一个小缓冲区中，甚至是专用的"L0"μop指令缓存中，以避免在循环期间一遍又一遍地重新翻译相同的x86指令，从而节省时间和功耗。这就是为什么，例如，Core i*2/i*3Sandy/Ivy Bridge的流水线深度在前面的超流水线部分中显示为14/19级 - 当处理器从其L0 μop缓存运行时为14级（这是常见情况），但当从L1指令缓存运行时为19级，并且必须解码x86指令并将其转换为μops。

x86指令获取和解码与内部类似RISC的μop指令调度和执行的解耦也使得定义现代x86处理器的*宽度*有点棘手，而且变得更加不清楚，因为在内部，这些处理器通常会在可能的情况下将μop分组或"融合"成公共对，以便于跟踪（例如加载和添加或比较和分支）。例如，Core i*4/i*5 Haswell/Broadwell等处理器每个周期最多可以解码5个x86指令，每个周期最多产生4个融合μops，然后将其存储在L0 μop缓存中，每个周期最多从中获取4个融合μops，然后寄存器重命名并放入重新排序缓冲区中， 每个周期最多从中向功能单元发出8 个未融合的单个 μop，在那里它们沿着各种管道向下移动，直到它们完成，因此每个周期最多可以提交和停用4 个融合的 μop。那么，哈斯韦尔/布罗德韦尔的宽度是多少呢？它本质上是一个 8问题处理器，因为如果它们以正确的方式配对/融合，每个周期最多可以获取、发出和完

成8 个未融合的 μop（未融合的 μop 是简单 RISC 指令的最直接等价物），但即使是专家也不同意这种设计的宽度， 由于4 问题在融合 μops 方面也是有效的，这是处理器在跟踪目的方面主要"考虑"的内容，如果根据原始 x86 指令进行思考，5-issue也是有效的。当然，这个宽度标记难题主要是学术性的，因为无论如何，没有处理器在运行真实世界的代码时可能真正维持如此高水平的ILP。

RISC风格的x86组中最有趣的成员之一是Transmeta Crusoe处理器，它将x86指令转换为内部VLIW形式，而不是内部超标量，并使用*软件*在运行时进行翻译，就像Java虚拟机一样。这种方法允许处理器本身成为一个简单的VLIW，没有解耦x86设计的复杂x86解码和寄存器重命名硬件，也没有任何超标量调度或OOO逻辑。与硬件转换相比，基于软件的x86转换确实降低了系统的性能（硬件转换作为额外的管道阶段发生，因此在性能方面几乎是免费的），但结果是一个非常精简的芯片，运行快速，凉爽，功耗极低。一个600 MHzCrusoe处理器可以与当时运行的500 MHz奔腾III相媲美，运行在低功耗模式（300 MHz时钟速度）下，同时只使用一小部分功率并产生一小部分热量。这使其成为笔记本电脑和手持式计算机的理想选择，其中电池寿命至关重要。当然，今天，专门为低功耗设计的x86处理器变体，如奔腾M及其酷睿后代，已经使Transmeta风格的基于软件的方法变得没有必要，尽管目前在NVIDIA的丹佛ARM处理器中使用了一种非常相似的方法，再次追求以极低的功耗实现高性能。

线程 – SMT、超线程和多核

如前所述，通过超标量执行来利用指令级并行性的方法被严重削弱了，因为大多数普通程序都没有很多细粒度并行性。正因为如此，即使是最激进的大脑 OOO 超标量处理器，再加上一个聪明而积极的编译器来喂它，在运行大多数主流的现实世界软件时，由于负载延迟、缓存未命中、分支和指令之间的依赖关系的组合，每个周期几乎永远不会超过平均约2-3条指令。在同一周期内发出许多指令最多只发生在几个周期的短时间内，由执行低ILP代码的许多周期隔开，因此峰值性能甚至无法实现。

如果在正在执行的程序中没有其他独立指令可用，则还有另一个潜在的独立指令来源 - 其他正在运行的程序或同一程序中的其他线程。*同时多线程* （SMT） 是一种处理器设计技术，它正是利用这种类型的线程级并行性。

再一次，这个想法是用有用的指令填充管道中的那些空气泡，但这次不是使用来自同一代码中更下方的指令（这很难获得），指令来自同时运行的*多个线程*，所有这些都在一个*处理器内核*上。因此，SMT 处理器在系统的其余部分看来就像是多个独立的处理器，就像真正的多处理器系统一样。

当然，真正的多处理器系统也会同时执行多个线程，但每个处理器中只有一个线程。*对于多核*处理器也是如此，它们将两个或多个处理器内核放在单个芯片上，但在其他方面与传统的多处理器系统没有什么不同。相比之下，SMT 处理器仅使用一个*物理*处理器内核向系统提供两个或多个*逻辑*处理器。这使得SMT在芯片空间、制造成本、功耗和散热方面比多核处理器效率高得多。当然，没有什么能阻止多核实现，其中每个内核都是SMT设计。

从硬件的角度来看，实现 SMT 需要复制处理器中存储每个线程"执行状态"的所有部分，例如程序计数器、架构上可见的寄存器（但不是重命名寄存器）、TLB 中保存的内存映射等。幸运的是，这些部件仅占整个处理器硬件的一小部分。真正大而复杂的部分，如解码器和调度逻辑、功能单元和缓存，都在线程之间共享。

当然，处理器还必须跟踪在任何给定时间点哪些指令和哪些重命名寄存器属于哪些线程，但事实证明，这只会增加核心逻辑的复杂性。因此，对于相对便宜的设计成本，即内核中的逻辑增加约10%，并且晶体管总数和最终生产成本的增加几乎可以忽略不计，处理器可以同时执行多个线程，有望大幅提高每个周期的功能单元利用率和指令，从而增加整体性能。

SMT 处理器的指令流看起来像...

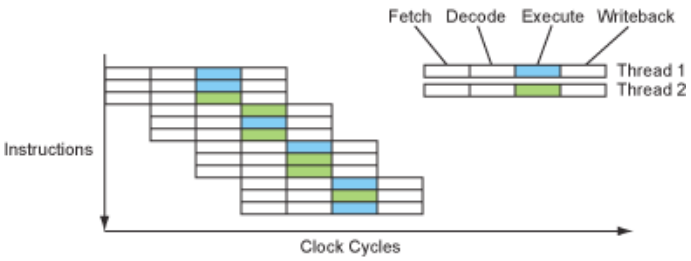


图 14 – SMT 处理器的指令流。

这真的很棒！现在我们可以通过运行多个线程来填补这些气泡，我们可以证明添加比单线程处理器通常可行的功能单元更多的功能单元是合理的，并且真正进入了多指令问题的城镇。在某些情况下，这甚至可能具有提高单线程性能的副作用（例如，对于特别对 ILP 友好的代码）。

所以20期我们来了，对吧？不幸的是，答案是否定的。

SMT 性能是一项棘手的业务。首先，SMT 的整个思想是围绕以下假设构建的：要么许多程序同时执行（而不仅仅是闲置），要么只有一个程序正在运行，则它有很多*线程同时执行*。现有多处理器系统的经验表明，这并不总是正确的。在实践中，至少对于台式机，笔记本电脑，平板电脑，手机和小型服务器，很少有几个不同的程序同时主动执行的情况，因此通常归结为机器当前用于的一项任务。

一些应用程序，如数据库系统、图像和视频处理、音频处理、3D 图形渲染和科学代码，确实具有明显的高级（粗粒度）并行性，并且易于利用，但不幸的是，即使这些应用程序中的许多也没有被编写为使用多个线程来利用多个处理器。此外，许多易于并行化的应用程序，因为它们本质上是"令人尴尬的并行"，主要受到内存带宽的限制，而不是处理器（图像处理，音频处理，简单的科学代码），因此添加第二个线程或处理器对它们没有多大帮助，除非内存带宽也显着增加（我们很快就会进入内存系统）。更糟糕的是，许多其他类型的软件，如网络浏览器、多媒体设计工具、语言解释器、硬件模拟等，目前根本没有以并行的方式编写，或者肯定不足以有效利用多个处理器。

最重要的是，与真正的多处理器（或多核）相比，SMT 设计中的线程都*只共享*一个处理器内核和一组缓存，这一事实具有重大的性能缺点。在 SMT 处理器的管道中，如果一个线程仅饱和其他线程所需的一个功能单元，则它实际上会阻止所有其他线程，即使它们只需要相对较少地使用该单元。因此，平衡线程的进度变得至关重要，SMT 最有效的用途是用于具有高度可变代码混合的应用程序，因此线程不会经常竞争相同的硬件资源。此外，线程之间对缓存空间的竞争可能会产生比只让一个线程拥有所有可用缓存空间更糟糕的结果 - 特别是对于关键工作集对缓存大小高度敏感的软件，例如硬件模拟器/仿真器，虚拟机和高质量视频编码（具有较大的运动估计窗口）。

最重要的是，如果不小心，甚至对某些应用程序也非常谨慎，SMT 性能实际上可能比单线程性能和线程之间的传统上下文切换更*差*。另一方面，主要受内存延迟（但不是内存带宽）限制的应用程序，如数据库系统、3D 图形渲染和大量通用代码，从 SMT *中受益匪浅*，因为它提供了一种在加载延迟和缓存未命中期间使用空闲时间的有效方法（我们将在后面介绍缓存）。因此，SMT呈现出非常复杂且特定于应用的性能图景。这也使它成为营销的艰巨挑战 - 有时几乎与两个"真正的"处理器一样快，有时更像两个真正蹩脚的处理器，有时甚至比一个处理器更糟糕，对吧？

奔腾 4是第一个使用 SMT 的处理器，英特尔称之为"超线程"。它的设计允许 2 个同时线程（尽管奔腾 4的早期版本由于错误而禁用了 SMT 功能）。奔腾 4上 SMT 的加速比从大约 10% 到 +30% 不等，具体取决于应用。随后的英特尔设计在过渡回奔腾M和酷睿2的大脑设计以及过渡到多核的过程中避开了SMT。许多其他SMT设计也在同一时间被取消（Alpha 21464，UltraSPARC V），有一段时间SMT似乎失宠了，直到它最终卷土重来，POWER5是一种2线程SMT设计以及多核（每个内核2线程乘以每个芯片2个内核等于每个芯片4个线程）。英特尔的酷睿i系列也是2线程SMT，因此典型的四核酷睿i处理器是8线程芯片。Sun在线程级并行性方面是最激进的，UltraSPARC T1Niagara提供了8个简单的顺序内核，每个内核都有4线程SMT，单个芯片上总共有32个线程。随后在UltraSPARC T2中增加到每个内核8个线程，然后在UltraSPARC T3中每个芯片增加到16个内核，高达128个线程！

更多内核还是更宽的核心？

鉴于 SMT 能够将线程级并行性转换为指令级并行性，再加上对于特别对 ILP 友好的代码具有更好的单线程性能优势，您现在可能会问，当同样宽（总计）的 SMT 设计会更胜一筹时，为什么有人会构建多核处理器。

不幸的是，事情并没有那么简单。事实证明，非常宽的超标量设计在芯片面积和时钟速度方面都非常糟糕。一个关键问题是复杂的多议题调度逻辑大致按问题宽度的*平方*扩展，因为所有n条候选指令都需要与其他候选指令进行比较。应用排序限制或"问题规则"可以减少这种情况，一些聪明的工程也可以，但它仍然是n^2 的顺序。也就是说，5期处理器的调度逻辑比4 期设计大 50% 以上，其中 6期大两倍以上，7期大小 3 倍以上，8 期比 4 期大4 倍以上（宽度仅为 2 倍），依此类推。此外，非常宽的超标量设计需要高度多端口的寄存器文件和缓存，以服务于所有这些同时访问。这两个因素不仅增加了尺寸，而且还在电路设计层面大幅增加了长距离布线的数量，从而严重限制了时钟速度。因此，单个 10 问题

内核实际上比两个5问题内核更大*且更慢*，由于电路设计限制，我们实现20 问题SMT 设计的梦想并不真正可行。

然而，由于SMT和多核的优势在很大程度上取决于目标应用的性质，因此对于不同程度的SMT和多核，广泛的设计可能仍然有意义。让我们探索一些可能性...

如今，“典型”的SMT设计意味着宽执行内核和OOO执行逻辑，包括多个解码器，大型复杂的超标量调度逻辑等。因此，就芯片面积而言，典型SMT内核的尺寸相当大。在相同的芯片空间下，可以安装*几个*更简单的单问题顺序内核（带或不带基本SMT）。事实上，多达六个小而简单的内核可能适合一个现代OOO超标量SMT设计的芯片区域！

现在，鉴于指令级并行性和线程级并行性都受到收益递减（以不同的方式）的影响，并且记住SMT本质上是一种将TLP转换为ILP的方法，但也要记住宽超标量设计在芯片面积（以及设计复杂性和功耗）方面非常非线性地扩展，显而易见的问题是最佳点在哪里？磁芯应该有多宽才能在 ILP 和 TLP 之间达到良好的平衡？目前，正在探索许多不同的方法.....

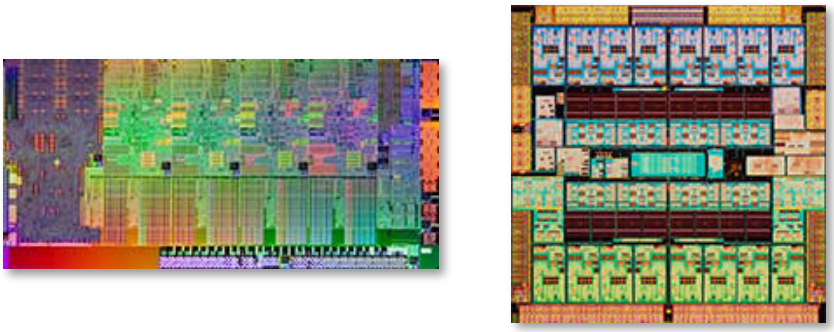


图 15 – 极端设计：Core i*2“桑迪桥”与UltraSPARC T3“尼亚加拉 3”。

在一个极端，我们有像英特尔的酷睿 i*2Sandy Bridge（左上）这样的处理器，由4 个大的、宽的、6 个问题、乱序、激进的 brainiac 内核组成（沿着顶部，下面是共享的 L3 缓存），每个运行2 个线程，总共8 个“快速”线程。另一方面，Sun/Oracle 的UltraSPARC T3Niagara 3（右上）包含 16 个更小、更简单的 2 个按顺序发行的内核（顶部和底部，中间有共享的 L2 缓存），每个内核运行 8 个线程，总共有 128 个线程，尽管这些线程比 Sandy Bridge 的线程慢得多。这两款芯片都属于同一个时代——2011年初。两者都包含大约10亿个晶体管，并且大约按比例绘制（假设晶体管密度相似）。请注意，简单的有序内核实际上要小得多！

哪种方法更好？唉，这里没有简单的答案——再一次，这将在很大程度上取决于应用程序。对于具有大量活动但内存延迟受限线程的应用程序（数据库系统、3D 图形渲染），更简单的内核会更好，因为大/宽内核无论如何都会花费大部分时间等待内存。然而，对于大多数应用程序来说，根本没有足够的线程活动来使其可行，并且单个线程的性能更为重要，因此具有更少但更大、更宽、更 brainiac 内核的设计更合适（至少对于今天的应用程序）。

当然，在这两个极端之间还有一系列尚未充分探索的选择。例如，IBM的POWER7是同一代，也有大约10亿个晶体管，并使用它们采取8核，4线程SMT设计，具有适度但不过于激进的OOO执行硬件。AMD 的 Bulldozer 设计采用了一种更不寻常的方法，每*对*内核都有一个共享的 SMT 式前端，后端提供非共享的多核式整数执行单元，但共享的 SMT 式浮点单元，模糊了 SMT 和多核之间的界限。

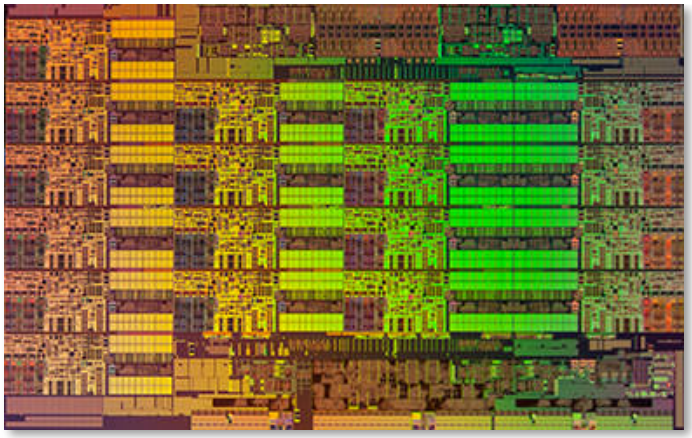


图 16 – 拥有 18 个大脑核心的至强哈斯威尔，两全其美?

今天（2015年初），由于摩尔定律，现在*有数十亿个*晶体管可用，即使是激进的大脑设计也可以有相当多的内核 - 英特尔的Xeon Haswell，酷睿i*4Haswell的服务器版本，使用57亿个晶体管提供18个内核（高于Xeon Sandy Bridge的8个），每个内核都是一个非常激进的brainiac 8问题设计（高于Sandy Bridge的6问题）， 每个内核仍然具有2 线程SMT，而 IBM 的 POWER8 使用44 亿个晶体管来转向比 POWER7 更聪明的内核设计，同时提供12 个内核（从 POWER7 的 8 个增加到 8 个），每个内核都有8 线程SMT（从 POWER7 的 4 个增加到 4 个）。当然，如此大的大脑核心设计是否*有效利用了*所有这些晶体管是一个单独的问题。

考虑到小内核的多核单位面积性能效率，但大内核的最大直接单线程性能，也许在未来我们可能会看到*不对称*设计，有一个或两个大而宽的brainiac内核加上大量更小，更窄，更简单的内核。在许多方面，这样的设计是最有意义的 - 高度并行程序将受益于许多小内核而不是几个大内核，但单线程，顺序程序需要至少一个大型，宽，brainiac内核的强大功能，即使它确实需要四倍的面积来提供两倍的单线程性能。

IBM的Cell处理器（用于索尼PlayStation 3）可以说是第一个这样的设计，但不幸的是，它遭受了严重的可编程性问题，因为Cell中的小而简单的内核与大型主内核的指令集不兼容，并且只能有限，笨拙地访问主内存，使它们更像专用协处理器而不是通用CPU内核。一些现代ARM设计也使用非对称方法，几个大内核与一个或几个更小，更简单的"伴侣"内核配对，不是为了获得最大的多核性能，但是如果手机或平板电脑只是少量使用，那么大，耗电的内核可以断电，以增加电池寿命，ARM称之为"大"。小"。

当然，有了所有这些晶体管，将其他辅助功能集成到主CPU芯片中也可能是有意义的，例如I / O和网络（通常是主板芯片组的一部分），专用视频编码/解码硬件（通常是图形系统的一部分），甚至是整个低端GPU（图形处理单元）。在减少芯片数量、物理空间或成本比主 CPU 芯片上的更多内核和用于其他目的的独立专用芯片的性能优势更重要的情况下，这种集成特别有吸引力，使其成为手机、平板电脑和小型低性能笔记本电脑的理想选择。这种异构设计称为*片上系统*或SoC...

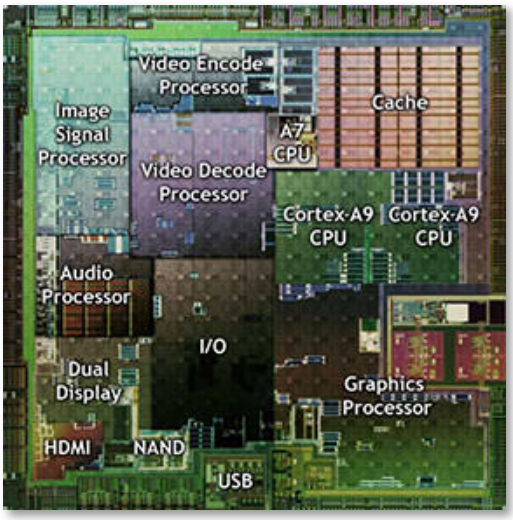


图 17 – 典型的 SoC：NVIDIA Tegra 2。

数据并行性 – SIMD 矢量指令

除了指令级并行性和线程级并行性之外，许多程序中还有另一个并行性来源——数据并行性。与其寻找并行执行指令组的方法，不如寻找使一*条*指令并行应用于一组*数据值*的方法。

这有时称为 SIMD 并行性（单指令，多数据）。更常见的是，它被称为*矢量处理*。超级计算机过去经常使用矢量处理，具有很长的矢量，因为在超级计算机上运行的科学程序类型非常适合矢量处理。

然而，今天，矢量超级计算机早已让位于多处理器设计，其中每个处理单元都是一个商品CPU。那么为什么要恢复矢量处理呢?

在许多情况下，特别是在成像、视频和多媒体应用中，程序需要对一*小组*相关值执行相同的指令，通常是短向量（简单结构或小数组）。例如，图像处理应用程序可能希望添加 8 位数字组，其中每个8 位数字表示像素的红色、绿色、蓝色或 alpha（透明度）值之一.....

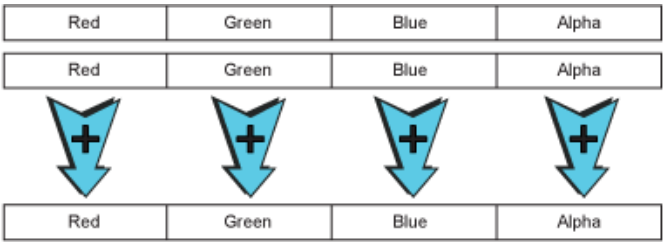


图 18 – SIMD 矢量加法运算。

这里发生的事情与32位加法完全相同的操作，只是每 8 次进位没有被传播。此外，当所有8 位都已满时，这些值可能不希望包装为零，而是在这些情况下保持为 255 作为最大值（称为饱和算术）。换句话说，每 8 次进位不会进行，而是触发全1结果。因此，上面显示的向量加法操作实际上只是一个修改后的32 位加法。

从硬件的角度来看，添加这些类型的矢量指令并不是非常困难 - 可以使用现有的寄存器，并且在许多情况下，功能单元可以与现有的整数或浮点单元共享。还可以添加其他有用的打包和解包指令，用于字节洗牌等，以及一些类似谓词的指令用于位屏蔽等。经过一些思考，一小组矢量指令可以实现一些令人印象深刻的加速。

当然，没有理由停止在32 位。如果碰巧有一些64位寄存器，架构通常具有浮点（至少），它们可用于提供64位向量，从而使并行性加倍- SPARC VIS和x86 MMX做到了这一点。如果可以定义全新的寄存器，那么它们可能会更宽 - x86 SSE增加了8个新的128位寄存器，后来在64位模式下增加到16个寄存器，然后用AVX扩展到256位，而POWER/PowerPC AltiVec从一开始就提供了一整套32个新的128位寄存器（与POWER/PowerPC更分离的设计风格保持一致， 甚至分支指令也有自己的寄存器）。扩大寄存器的另一种方法是使用配对，其中每对寄存器被 SIMD 矢量指令视为单个操作数 – ARM NEON就是这样做的，其寄存器既可用作 32个 64位寄存器，也可用作 16 个128 位寄存器。

当然，寄存器中的数据也可以以其他方式划分，而不仅仅是8位字节 - 例如用于高质量图像处理的16位整数，或用于科学数字运算的浮点值。例如，使用 AltiVec、NEONv2 和最新版本的 SSE/AVX，可以将4 向并行浮点乘加作为单个完全流水线的指令执行。

对于此类数据并行性可用且易于提取的应用，SIMD 矢量指令可以产生惊人的加速。最初的目标应用主要是图像和视频处理领域，但合适的应用还包括音频处理、语音识别、3D 图形渲染的某些部分和许多类型的科学代码。对于其他类型的软件，例如编译器和数据库系统，加速通常要小得多，甚至可能根本没有。

不幸的是，编译器在处理普通源代码时很难自动使用矢量指令，除非在微不足道的情况下。关键问题是程序员编写程序的方式倾向于序列化所有内容，这使得编译器很难证明两个给定的操作是独立的并且可以并行完成。这一领域正在慢慢取得进展，但目前程序基本上必须手动重写，以利用矢量指令（科学代码中基于数组的简单循环除外）。

但是，幸运的是，仅在您喜欢的操作系统的图形和视频/音频库中的关键位置重写少量代码，在许多应用程序中都会产生广泛的影响。如今，大多数操作系统都以这种方式增强了其关键库功能，因此几乎所有多媒体和3D图形应用程序都利用了这些高效的矢量指令。为抽象创造又一次胜利！

现在，几乎每个架构都添加了 SIMD 矢量扩展，包括 SPARC （VIS）、x86 （MMX/SSE/AVX）、POWER/PowerPC （AltiVec） 和 ARM （NEON）。然而，只有来自每种架构的相对较新的处理器才能执行其中一些新指令，这引发了向后兼容性问题，尤其是在 x86 上，其中 SIMD 矢量指令的发展有些随意（MMX、3DNow！、SSE、SSE2、SSE3、SSE4、AVX、AVX2）。

记忆与记忆墙

如前所述，延迟对于流水线处理器来说是一个大问题，延迟对于来自内存的负载尤其糟糕，内存约占所有指令的四分之一。

加载往往发生在代码序列（基本块）的开头附近，大多数其他指令取决于要加载的数据。这会导致所有其他指令停止，并且难以获得大量的指令级并行性。事情甚至比最初看起来更糟糕，因为在实践中，大多数超标量处理器仍然每个周期只能发出一个或最多两个加载指令。

内存访问的核心问题是构建快速内存系统非常困难，部分原因是光速等固定限制，当信号传输到RAM并返回时会产生延迟，更重要的是因为充电和耗尽构成存储单元的微小电容器的速度相对较慢。没有什么可以改变这些自然事实——我们必须学会绕过它们。

例如，使用CAS 延迟为 11 的现代 SDRAM，主内存的访问延迟通常是内存*系统总线*的24 个周期 – 1 将地址发送到 DIMM（内存模块），RAS 到 CAS延迟为 11 用于行访问，CAS延迟为 11 用于列访问，最后 1个周期将第一段数据发送到处理器（或电子缓存），剩余的数据块在接下来的几个总线周期中跟随。当然，这只是典型的延迟，因为内存延迟变化很大。对于任何给定的访问，我们可能会幸运地访问内存控制器已经为我们打开正确行的内存库，跳过RAS 到 CAS的延迟并节省近 50% 的延迟（1+11+1），或者我们可能不走运并访问内存控制器打开了一行但这是错误的行， 需要先收起来，等待额外的RAS预充电延迟，并增加近50%的延迟（1+11+11+11+11+1）。在多处理器系统上，可能需要更多的总线周期来支持处理器之间的高速缓存一致性。然后是处理器本身的循环，在地址发送到内存控制器之前检查各种片上缓存，然后当数据从RAM到达内存控制器并发送到相关的处理器内核时。幸运的是，这些是更快的内部CPU周期，而不是内存总线周期，但在大多数现代处理器中，它们仍然占20个CPU周期左右。

假设一个典型的800 MHz SDRAM内存系统（DDR3-1600），假设一个2.4 GHz的处理器，这使得（1+11+11+1）*2400/800 + 20 = 92个周期的*CPU时钟*访问主内存！啧，你说！而且情况会变得更糟 - 2.8 GHz 处理器需要 104 个周期，3.2 GHz 处理器需要 116 个周期，3.6 GHz 处理器需要 128 个周期，而4.0GHz处理器需要等待惊人的 140 个周期才能访问主内存！

老一代处理器甚至更糟，因为它们的内存控制器不在芯片上，而是主板上芯片组的一部分，为处理器和主板芯片组之间的地址和数据传输增加了另外 2 个总线周期——当时内存总线只有200 MHz或更低， 不是800 MHz，因此这2 个总线周期通常会在总数中再增加20+ 个 CPU周期。一些处理器试图通过提高处理器和芯片组之间的*前端总线*（FSB）的速度来缓解此问题（PowerPC G5中奔腾4，1.25 GHzDDR中的800 MHzQDR）。所有现代处理器都使用的一种更好的方法是将内存控制器直接集成到处理器芯片中，这允许将这 2*个总线周期*转换为更快的*CPU 周期*。UltraSPARC IIi和Athlon 64是第一批这样做的主流处理器，而英特尔迟到了，只将内存控制器集成到他们的CPU芯片中，从Core i系列开始。

> 请注意，尽管DDR SDRAM内存系统在时钟信号的上升沿和下降沿上传输数据（即：*以双倍数据速率*），但内存系统总线的真实时钟速度仅为其一半，并且是适用于控制信号的总线时钟速度。因此，DDR 内存系统的延迟与非 DDR系统的延迟相同，即使带宽增加了一倍（稍后将详细介绍带宽和延迟之间的差异）。

不幸的是，DDR SDRAM内存和片上内存控制器都只能做这么多，内存延迟仍然是一个主要问题。处理器和主内存之间差距大且缓慢增长的问题称为*内存墙*。它曾经是处理器架构师面临的最重要的一个问题，尽管今天这个问题已经大大缓解，因为由于功率和热量限制 - *电源墙*，处理器时钟速度不再以以前的速度攀升。

尽管如此，内存墙仍然是一个大问题。

缓存和内存层次结构

现代处理器解决了*带有缓存*的内存墙问题。高速缓存是位于处理器芯片上或附近的一种小而快速的内存类型。它的作用是保留主存储器的小块副本。当处理器请求特定的主内存时，如果数据在缓存中，缓存可以提供比主内存快得多的速度。

通常，处理器芯片本身在每个内核内部都有小而快速的"主"1 级（L1）缓存，大小通常在8-64k左右，较大的2 级（L2）缓存距离较远但仍在片上（几百 KB 到几 MB），可能还有更大更慢的 L3 缓存等。片上缓存、任何片外外部缓存（E-cache）和主存储器（RAM）的组合共同构成了*一个存储器层次结构*，每个连续的级别都比前一个级别更大但更慢。当然，内存层次结构的底部是虚拟内存（分页/交换），它通过将 RAM 页面移入和移出文件存储（再次变慢，很大幅度）来提供几乎无限数量的主内存的错觉。

> "缓存"这个词的发音就像"现金"……如"武器藏匿处"或"补给品藏匿处"。它的意思是藏匿或存放东西的地方。它***不***发音为"ca-shay"或"kay-sh"。

这有点像在图书馆的书桌前工作……你可能在桌子上打开了两三本书。访问它们很快（你可以看看），但你不能同时在桌子上放一对以上的书——即使你可以，访问摆在一张大桌子上的 100 本书也需要更长的时间，因为你必须在它们之间走动。相反，在书桌的角落里，你可能还有一堆十几本书。访问它们的速度较慢，因为您必须伸手过去，抓住一个并打开它。每次你打开一本新的，你还必须把桌子上已经有的一本书放回书堆里，腾出空间。最后，当你想要一本不在桌子上的书，也不在书堆里时，它访问起来非常慢，因为你必须站起来在图书馆里走来走去寻找它。然而，图书馆的规模意味着您可以访问数千本书，远远超过您的办公桌所能容纳的书籍。

典型的现代内存层次结构看起来像…

水平	*大小*	*延迟*	*物理位置*
一级缓存	32 KB	4 个周期	每个核心内部
二级缓存	256 KB	12 个周期	在每个核心旁边
三级缓存	6兆字节	~21 次循环	在所有内核之间共享
L4电子缓存	128兆字节	~58 次循环	独立的电子硬盘录像机芯片
公羊	4+ 国标	~117 次循环	主板上的 SDRAM 内存
交换	100+ GB	10，000+ 次循环	硬盘或固态硬盘

表 4 – 现代台式机/笔记本电脑的内存层次结构：Core i*4 Haswell。

甚至手机也有这样的内存层次结构……

水平	*大小*	*延迟*	*物理位置*
一级缓存	64 KB	4 个周期	每个核心内部
二级缓存	1 兆字节	~20 次循环	在核心旁边
三级缓存	4兆字节	~107 次循环	在内存控制器旁边
公羊	1 千兆字节	~261 次循环	独立的 SDRAM 芯片
交换	不适用	不适用	在 iOS 上不使用分页/交换

表 5 – 现代手机的内存层次结构：iPhone 8 中的 Apple A6。

缓存的惊人之处在于它们运行*得非常好*——它们有效地使内存系统看起来几乎与 L1 缓存一样快，但与主内存一样大。现代主 （L1） 缓存的延迟仅为 2 到 4 个处理器周期，比访问主内存快数十倍，而现代主缓存对于大多数软件的命中率约为 90%。所以90%的时间，访问内存只需要几个周期！

由于程序的工作方式，缓存可以实现这些看似惊人的命中率。大多数程序在时间和空间上都表现出局部性——当一个程序访问一段内存时，它很有可能需要在不久的将来重新访问同一块内存（时间局部性），而且它也很有可能在未来也需要访问附近的其他内存（空间*局部性*）。仅通过将最近访问的数据保留在缓存中来利用时间局部性。为了利用空间局部性，数据从主内存以一次几十字节的块传输到缓存中，称为*缓存行*。

从硬件的角度来看，缓存的工作方式类似于两列表 - 一列是内存地址，另一列是数据值块（请记住，每个缓存行都是整个数据块，而不仅仅是单个值）。当然，实际上缓存只需要存储地址的必要高端部分，因为查找通过使用地址的下半部分来索引缓存。当称为标签的较高部分与表中存储的*标签*匹配时，这是*命中*，可以将适当的数据发送到处理器内核.....

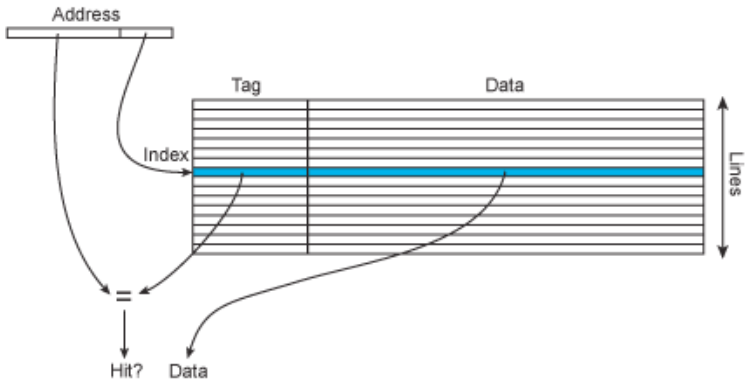


图 19 – 缓存查找。

可以使用物理地址或虚拟地址来执行缓存查找。每个都有优点和缺点（就像计算中的其他一切一样）。使用虚拟地址可能会导致问题，因为不同的程序使用相同的虚拟地址映射到不同的物理地址 - 可能需要在每个上下文切换时刷新缓存。另一方面，使用物理地址意味着虚拟到物理映射必须作为缓存查找的一部分执行，从而使每次查找都变慢。一个常见的技巧是使用虚拟地址进行缓存索引，而使用物理地址进行标记。然后，虚拟到物理映射（TLB 查找）可以与缓存索引并行执行，以便及时准备好进行标记比较。这种方案称为*虚拟索引物理标记*缓存。

现代处理器中各种缓存级别的大小和速度对性能绝对至关重要。到目前为止，最重要的是主一级数据缓存（D-cache）和L1指令缓存（I-cache）。一些处理器采用小型 L1 缓存（Pentium 4E Prescott、Scorpion 和 Krait 有 16k L1 缓存（针对 I 和D 缓存），早期的 Pentium 4s 和UltraSPARC T1/T2/T3甚至更小，仅为 8k），大多数处理器已将 32k 作为最佳点，少数在 64k 时更大（Athlon，Athlon 64/Phenom，UltraSPARC III/IV，苹果A7 / A8）或偶尔甚至128k（丹佛的I缓存，具有64kD缓存）。

对于现代 L1 数据缓存，加载延迟通常为 3 或 4 个周期，具体取决于处理器的一般时钟速度，但偶尔会更短（UltraSPARC III/IV上为 2 个周期，这要归功于无时钟“波”流水线，早期处理器需要 2 个周期，因为它们的时钟速度较慢，流水线较短，其中时钟周期更实时）。将负载延迟增加一个周期，例如从 3 增加到 4，或从 4 增加到 5，这似乎是一个很小的变化，但实际上是对性能的严重影响，最终用户很少注意到或理解这一点。对于正常的日常指针跟踪代码，处理器的负载延迟是影响实际性能的主要因素。

大多数现代处理器都有大型的第二级或第三级片上缓存，通常在所有内核之间共享。此缓存也非常重要，但其大小最佳位置在很大程度上取决于正在运行的应用程序类型和该应用程序的活动*工作集*的大小 - 对于某些应用程序，2MB 的 L3 缓存和8 MB之间的差异几乎无法测量，而对于其他应用程序来说，这将是巨大的。鉴于相对较小的 L1 缓存已经占据了许多现代处理器内核芯片面积的很大一部分，您可以想象大型 L2 或 L3 缓存将占用多少面积，但这对于对抗内存墙仍然绝对必要。通常，大型L2 / L3缓存占用了总芯片面积的*一半*，以至于在芯片照片中清晰可见，与内核和内存控制器的更“混乱”的逻辑晶体管相比，它是一个相对干净，重复的结构。

缓存冲突和关联性

理想情况下，缓存应保留将来最有可能需要的数据。由于缓存不是通灵的，因此一个很好的近似方法是保留最近使用的数据。

遗憾的是，*准确*保留最近使用的数据意味着来自任何内存位置的数据都可以放入*任何*缓存行中。因此，缓存将恰好包含最近使用的*n*KB数据，这对于利用局部性非常有用，但不幸的是不适合允许快速访问 - 访问缓存需要检查*每个*缓存行是否可能匹配，这对于具有数百行的现代缓存来说非常慢。

相反，缓存通常只允许来自内存中任何特定地址的数据占用缓存中的一个或最多几个位置。因此，在访问过程中只需要进行一次或几次检查，因此可以保持快速访问（这是首先拥有缓存的全部意义）。但

是，这种方法确实有一个缺点 - 这意味着缓存不会存储最近访问的绝对最佳数据集，因为内存中的几个不同位置都将映射到缓存中的*同一位置*。当需要同时两个这样的内存位置时，这种情况称为*缓存冲突*。

缓存冲突可能会导致"病态"的最坏情况性能问题，因为当程序重复访问碰巧映射到同一缓存行的两个内存位置时，缓存必须继续从主内存存储和加载，从而在每次访问时遭受长时间的主内存延迟（100 个周期或更多，记住！这种情况称为*抖动*，因为缓存没有实现任何目标，只是妨碍了 - 尽管有明显的时间位置和数据重用，但由于内存位置和缓存行之间的简单映射的限制，缓存无法利用此特定访问模式提供的位置。

为了解决这个问题，更复杂的缓存能够将数据放置在缓存中的少量不同位置，而不仅仅是单个位置。一段数据可以存储在缓存中的位置数称为其*关联性*。"关联性"一词来自这样一个事实，即缓存查找通过关联工作 - 也就是说，内存中的特定地址与缓存中的特定位置（或集合关联缓存的一组位置）相关联。

如上所述，最简单和最快的缓存只允许内存中的每个地址在缓存中有一个位置 - 每条数据都简单地映射到缓存中的地址*百分比大小*，只需查看地址的较低位（如上图所示）。这称为*直接映射*缓存。内存中任何两个位置，其地址对于较低地址位相同，都将映射到直接映射缓存中的同一缓存行，从而导致缓存冲突。

允许数据根据其地址占据 2 个位置之一的缓存称为2 路*集合关联*。同样，4向集关联缓存允许任何给定数据片段有 4 个可能的位置，8 向缓存允许 8个可能的位置。集合关联缓存的工作方式与直接映射缓存非常相似，除了有几个表，所有表都并行索引，并且比较每个表中的标记以查看其中任何一个是否匹配.....

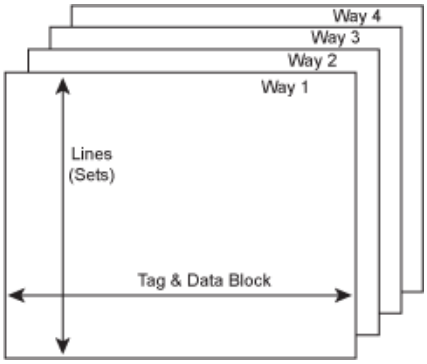


图 20 –4 路集关联缓存。

每个表或方式也可能有标记位，以便在引入新行时仅逐出最近最少使用*方式*的行，或者可能是该理想的一些更快的近似值。

通常，集关联缓存能够避免由于不幸的缓存冲突而导致直接映射缓存偶尔出现的问题。添加更多方法可以避免更多冲突。遗憾的是，缓存的关联性越高，访问速度就越慢，因为在每次访问期间要执行的操作就越多。即使比较本身是并行执行的，也需要额外的逻辑来选择适当的命中（如果有），并且缓存还需要在每种方式中适当地更新标记位。还需要更大的芯片面积，因为相对更多的缓存数据被标签信息而不是数据块消耗，并且需要额外的数据路径来并行访问缓存的每个单独方式。任何和所有这些因素都可能对访问时间产生负面影响。因此，2 路集关联缓存比直接映射缓存更慢但更智能，4 路和8 路更*慢但更智能*。

在大多数现代处理器中，指令缓存可以承受高度的设置关联，因为它的延迟在某种程度上被处理器管道早期阶段的获取和缓冲所隐藏。另一方面，数据缓存通常在某种程度上是设置关联的，但通常不会过度关联，以最大程度地减少所有重要的加载延迟。大多数处理器已经确定4路集合关联作为最佳点，但少数不太关联（Athlon，Athlon 64 / Phenom，PowerPC G5和Cortex-A15 / A57中的2路），少数更具关联性（PowerPC G4e，Pentium M及其核心后代中的8路）。作为前往遥远的主存储器之前的最后手段，大型L2 / L3缓存（有时称为LLC的"最后一级缓存"）通常也是高度关联的，可能多达12或16路，尽管外部E缓存有时是直接映射的，以实现大小和实现的灵活性。

缓存的概念也延伸到软件系统中。例如，操作系统使用主内存来缓存文件系统的内容以加快文件 I/O，Web 浏览器缓存最近查看的页面、图像和 JavaScript 文件，以防您再次访问这些站点，Web 缓存服务器（也称为代理缓存）将远程 Web 服务器的内容缓存在更本地的服务器上（您的 ISP 几乎肯定会使用

一个）。关于主内存和虚拟内存（分页/交换），它可以被认为是一个智能的、完全关联的缓存，就像最初提到的理想缓存（上面）一样。毕竟，虚拟内存是由操作系统内核的（希望）智能软件管理的。

内存带宽与延迟

由于内存是按块传输的，并且由于缓存未命中是一种紧急的"显示阻止"类型的事件，可能会使处理器停止其轨道（或至少严重阻碍其进度），因此从内存传输这些块的速度至关重要。内存系统的传输速率称为其*带宽*。但这与延迟有何不同*?*

一个很好的类比是高速公路...假设您想从 100 英里外开车前往城市。通过将车道数量增加一倍，每小时可以行驶的汽车总数（带宽）翻倍，但您自己的旅行时间（延迟）不会减少。如果你只想增加每秒的汽车数，那么增加更多的车道（更宽的公交车）就是答案，但如果你想减少特定汽车从A到B的时间，那么你需要做其他事情——通常要么提高速度限制（公交车和RAM速度），要么减少距离， 或者也许建立一个区域购物中心，这样人们就不需要经常去城市（缓存）。

对于内存系统，延迟和带宽之间通常存在微妙的权衡。低延迟设计将更适合指针跟踪代码，例如编译器和数据库系统，而面向带宽的系统对于具有简单线性访问模式的程序（例如图像处理和科学代码）具有优势。当然，*增加带宽相当容易* - 只需添加更多内存组并使总线更宽，即可轻松将带宽增加一倍或四倍。事实上，许多高端系统这样做是为了提高性能，但它也有缺点。特别是，更宽的总线意味着更昂贵的主板，对将RAM添加到系统的方式的限制（成对或四人一组安装）以及更高的最低RAM配置。

不幸的是，*延迟*比带宽更难改善——俗话说：*"你不能贿赂上帝"*。即便如此，在过去几年中，*有效*内存延迟也有一些很好的改进，主要是以同步时钟DRAM（SDRAM）的形式，它使用与内存总线相同的时钟。SDRAM的主要优点是它允许*对内存系统进行流水线处理*，因为SDRAM芯片操作的内部时序方面和交错结构暴露在系统中，因此可以利用。这减少了有效延迟，因为它允许在当前内存访问完成之前启动新的内存访问，从而消除了在较旧的异步 DRAM 系统中发现的少量等待时间，这些系统必须等待当前访问完成才能开始下一次访问（平均而言，异步内存系统必须等待从先前访问传输半个缓存行才能启动新请求， 这通常是几个公共汽车周期，我们知道这些有多慢!

除了有效延迟的减少之外，带宽也大幅增加，因为在SDRAM内存系统中，多个内存请求在任何时候都可能未完成，所有这些都以高效，完全流水线的方式处理。内存系统的流水线对内存带宽有巨大的影响 - SDRAM内存系统通常提供相同时代异步内存系统的两倍或三倍的持续内存带宽，即使SDRAM系统的延迟仅略低，并且相同的底层存储单元技术正在使用（并且仍然如此）。

内存技术的进一步改进，以及更高级别的缓存，是否能够继续阻止内存墙，同时扩展到越来越多的处理器内核所需的更高带宽？或者我们很快就会不断受到内存（带宽和延迟）的瓶颈，处理器微架构和内核数量都没有太大区别，内存系统才是最重要的？这将是有趣的，虽然预测未来绝非易事，但有充分的理由保持乐观.....

确认

本文的整体风格，特别是关于处理器"指令流"和微体系结构图的风格，源自Norman Jouppi和David Wall在1989年发表的一篇著名的ASPLOS论文，Shlomo Weiss和James Smith的POWER&PowerPC一书，以及两本非常著名的Hennessy/Patterson教科书《计算机体系结构》： 定量方法和计算机组织与设计。

当然，同样的材料还有很多其他的介绍，自然它们都有些相似，但是上述四个非常好（在我看来）。要了解有关这些主题的更多信息，这些书籍是一个很好的起点。

更多信息?

如果您想了解有关最新处理器设计细节的更多详细信息，以及比原始技术手册更有见地的内容，这里有几篇好文章.....

- 英特尔 Skylake 移动式和台式机发布，带有架构分析 – 当前的英特尔 x86 处理器设计，酷睿 i*6"Skylake"，这是 Haswell 的增量步骤。
- 英特尔的Haswell CPU微架构 - 以前的Intel x86处理器设计，Core i*4"Haswell"，主要基于之前的"Sandy Bridge"设计。

- 英特尔的Sandy Bridge微架构 – 最近最重要的Intel x86处理器设计，Core i*2"Sandy Bridge"，融合了奔腾Pro和Pentium 4的设计风格。
- AMD 的推土机微架构 – AMD 基于 Bulldozer 的处理器设计中使用的新颖资源共享方法，模糊了 SMT 和多核之间的界限。
- 英特尔下一代微架构揭幕 – 英特尔从奔腾 Pro/II/III/M中复兴了古老的 P6 内核，以生产酷睿微架构。
- 奔腾 4 和 PowerPC G4e（以及第二部分） – 比较了两个非常流行和成功的处理器的不同设计，尽管有些恶意。
- 进入K7（和第二部分） -AMD速龙，唯一真正挑战英特尔在x86处理器领域的统治地位的竞争对手。
- AMD 皓龙微处理器（视频） –1 小时的演示，涵盖皓龙/速龙 64 处理器和 AMD 对 x86 架构的64 位扩展。
- Niagara II： The Hydra Returns——Sun创新的UltraSPARC TNiagara处理器，针对第二代进行了修订，并将线程级并行性发挥到了极致。
- Crusoe Explored– Transmeta Crusoe 处理器及其基于软件的 x86 兼容性方法。

这里有一些与任何特定处理器无关的文章，但仍然非常有趣.....

- 设计 Alpha 微处理器 – 深入了解在制造新处理器的项目的各个阶段真正发生的事情。
- CPU 架构师需要考虑的事情（视频） – 由Bob Colwell 提供的有趣的 80 分钟演讲，他是奔腾 Pro/II/III 的主要架构师之一。

如果您想跟上微处理器领域的最新消息.....

- Ars Technica
- 阿南德科技
- 微处理器报告
- 真实世界技术

这应该让你很忙！