冯诺依曼架构与当代处理器

> 翻自：《Introduction to High Performance Scientific Computing》-Victor Eijkhout
>
> 李岳昆（realyurk@gmail.com）、蒋志政（gezelligheid@aliyun.com）译
>
> 知乎专栏：高性能计算翻译计划
>
> Github专栏：https://github.com/realYurkOfGitHub/translation-Introduction-to-HPC

了解计算机体系结构对于编写高效、科学的代码具有十分重要的作用。两段基于不同处理器架构编写的代码，其计算结果可能相同，但速度差异可能从几个百分点到几个数量级之间不等。显然，仅仅把算法放到计算机上是不够的，计算机架构也是至关重要的内容。

有些问题可以在单个中央处理单元（CPU）上解决，而有些问题则需要由多个处理器组成的并行计算机解决。我们将在下一章详细介绍并行计算机，但即便是使用并行计算机处理，也需要首先了解单个CPU的情况。

在该部分，我们将重点关注CPU及其内存系统内部发生的事情。首先讨论指令如何执行，研究处理器核心中的运算；最后，由于内存访问通常比处理器执行指令要慢得多，因此我们将重点关注内存、处理器以及处理器内部的数据移动情况；“flops（每秒浮点操作数）计数”作为预测代码性能的时代已经一去不复返了。这种差异实际上是一个不断增长的趋势，所以随着时间的推移，处理内存流量的问题变得越来越重要，而非逐渐销声匿迹。

这一章中，我们将对CPU设计是如何影响性能的，以及如何编写优化性能的代码等问题有一个清晰的认识。想学习更多细节，请参阅关于PC架构的在线书籍[114]，以及关于计算机架构的标准工作，Hennesey和Patterson[97]。

冯·诺依曼架构

虽然各类计算机在处理器细节上存在很多不同，但也有许多相似之处。总的看来，它们都采用了「**冯·诺伊曼架构**」（von Neumann architectures）。该架构主要包含：存储程序和数据的内存，以及一个在“获取、执行、存储周期”中对数据进行操作的指令处理单元。

注释 1： 具有指定指令序列的模型也称为「**控制流**」（control flow），与「**数据流**」（data flow）相对应。

由于指令和数据共同存储在一个处理器中，这使得冯·诺依曼架构区别于早期或一些其他特殊用途的硬接线当代处理器，能够允许修改或生成其他程序。这给我们提供了编辑器和编译器：计算机可以将程序视为数据进行处理。

注释 2： 存储程序的概念允许一个正在运行的程序修改其源代码。然而，人们很快就意识到这将导致代码变得难以维护，因此在实际中很少见到。

本书将不会讨论编译器将高级语言翻译成机器指令的过程，而是讨论如何编写高质量的程序以确保底层运行的效率。

在科学计算中，我们通常只关注数据在程序执行期间如何移动，而非程序代码具体如何。大多数应用中，程序与数据似乎是分开存储的。与高级语言不同，处理器执行的机器指令通常会指定操作的名称，操作数和结果的位置。这些位置不是表示为内存位置，而是表示为「**寄存器**」（registers）位置：即在CPU中被称作内存的一小部分。

注释 3： 我们很少分析到内存的架构，尽管它们已经存在。20世纪80年代的Cyber 205超级计算机可以同时有三个数据流，两个从内存到处理器，一个从处理器到内存。这样的架构只有在内存能够跟上处理器速度的情况下才可行，而现在已经不是这样了。

下面是一个简单的C语言例子：

```
void store(double *a, double *b, double *c) {
        *c = *a + *b;
}
```

及其X86汇编输出，由`gcc -O2 -S -o - store.c`得到：

```
          .text
          .p2align 4,,15
    .globl store
          .type   store, @function
     store:
           movsd   (%rdi), %xmm0  # Load *a to %xmm0
           addsd   (%rsi), %xmm0  # Load *b and to %xmm0
           movsd   %xmm0, (%rdx) # Store to *c
           ret
```

(程序演示的为64位系统输出；32位系统可以添加`-m64`指令输出)

这段程序的指令有：

- 从内存加载到寄存器；
- 执行加法操作；
- 将结果写回内存。

每条指令的处理如下：

- 指令获取：根据「**程序计数器**」（program counter）的指示将下一条指令装入进程。此处我们不考虑这是如何发生以及从哪里发生的问题。
- 指令解码：处理器检查指令以确定操作和操作数。
- 内存获取：必要时，数据将从内存取到寄存器中。
- 执行：执行操作，从寄存器读取数据并将数据写回寄存器。

数组的情况稍微复杂一些：加载（或存储）的元素被确定为数组的基址后会加上一个偏移量。

在某种程度上，现代CPU在程序员看来就像冯·诺伊曼机器，但也有各种例外。首先，虽然内存看起来为随机寻址，但在实际中存在着「**局部性**」（locality）的概念：一旦一个数据项被加载，相邻的项将更有效地加载，而重新加载初始项也会更快。

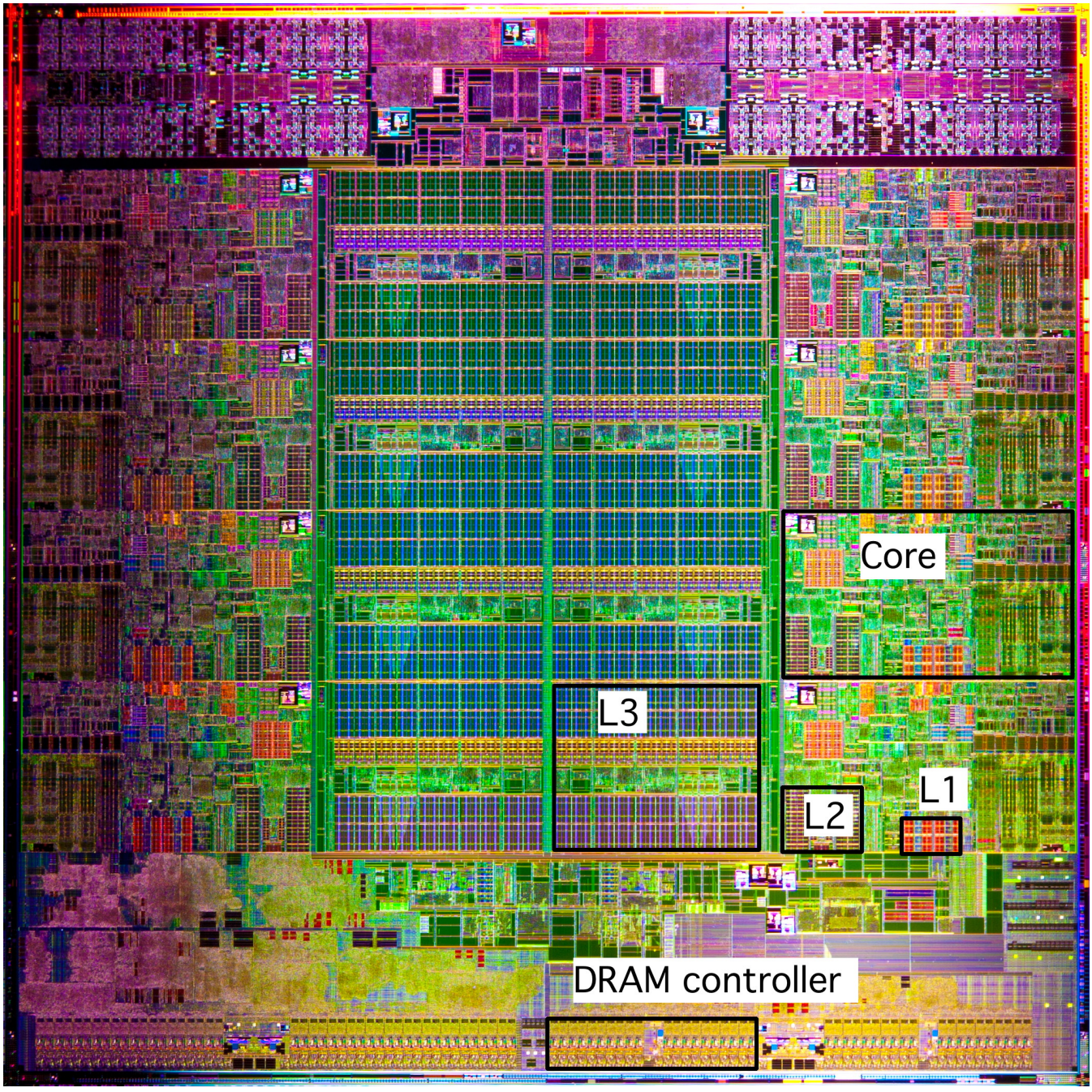
简单数据加载的另一个复杂之处是，当前的CPU同时操作多条指令，这些指令被称为“「**正在执行**」（in flight）”，这意味着它们处于不同的完成阶段。当然，与这些同步指令一起，它们的输入和输出也以重叠的方式在内存和处理器之间移动。这是超标量CPU体系结构的基本思想，也被称为「**指令级并行**」（Instruction Level Parallelism，ILP）。因此，虽然每个指令可能需要几个时钟周期才能完成，但处理器可以在合适的情况下每个周期完成一条指令；在某些情况下，每个周期可以完成多条指令。

CPU的处理速度处在千兆赫级（G），意味着处理器的速度是决定计算机性能的主要因素。速度虽然与性能紧密联系，但实际情况却更为复杂：一些算法被CPU所限制，此时进程的速度是最重要的制约；另外一些算法受到内存的限制，总线速度和缓存大小等方面是影响该问题的关键。

在科学计算中，第二种情况相当显著，因此在本章中，我们将大量关注将数据从内存转移到处理器的过程，而对实际处理器的关注相对较少。

当代处理器

当代处理器极为复杂，在这一节中，我们将简短地介绍一下其组成部分。下图是Intel Sandy Bridge处理器的芯片图。这种芯片大约一英寸大小，包含近十亿个晶体管。



处理核心

冯·诺依曼模型中只有一个执行指令的实体。自21世纪初以来，这种情况并没有显著的增长。上图所示的Sandy Bridge有8个核，每个核都是执行指令流的独立单元。在本章中，我们将主要讨论单个核心的各个方面；第1.4节将讨论多核的集成方面。

指令处理

冯·诺伊曼模型也是不现实的，因为它假设所有的指令都严格按照顺序执行。在过去的二十年中，处理器越来越多地使用了无序指令处理，即指令可以按照不同于用户程序指定的顺序进行处理。当然，处理器只有在不影响执行结果的情况下才允许对指令重新排序！

在`图1.2`中，你可以看到与指令处理有关的各种单元：这种聪明的做法实际上要花费相当多的能源及大量的晶体管。正因如此，包括第一代英特尔Xeon Phi协处理器，Knights Corner在内的处理器都采取了使用顺序指令处理的策略。然而，在下一代Knights Landing中，这种做法却由于性能不佳而被淘汰。

浮点单元

在科学计算中，我们最感兴趣的是处理器如何处理浮点数据。而并非整数或布尔值的运算。因此，核心在处理数值数据方面具有相当的复杂性。

例如，过去的处理器只有一个「**浮点单元**」（FPU），而现在的它们有多个且能够同时执行的浮点单元。

例如，加法和乘法通常是分开的；如果编译器可以找到独立的加法和乘法操作，就可以同时调度它们从而使处理器的性能翻倍。在某些情况下，一个处理器会有多个加法或乘法单元。

另一种提高性能的方法是使用「**乘加混合运算**」（Fused Multiply-Add，FMA）单元，它可以在与单独的加法或乘法相同的时间内执行指令$x \leftarrow ax + b$。配合流水线操作，这意味着处理器在每个时钟周期内有几个浮点运算的渐近速度。

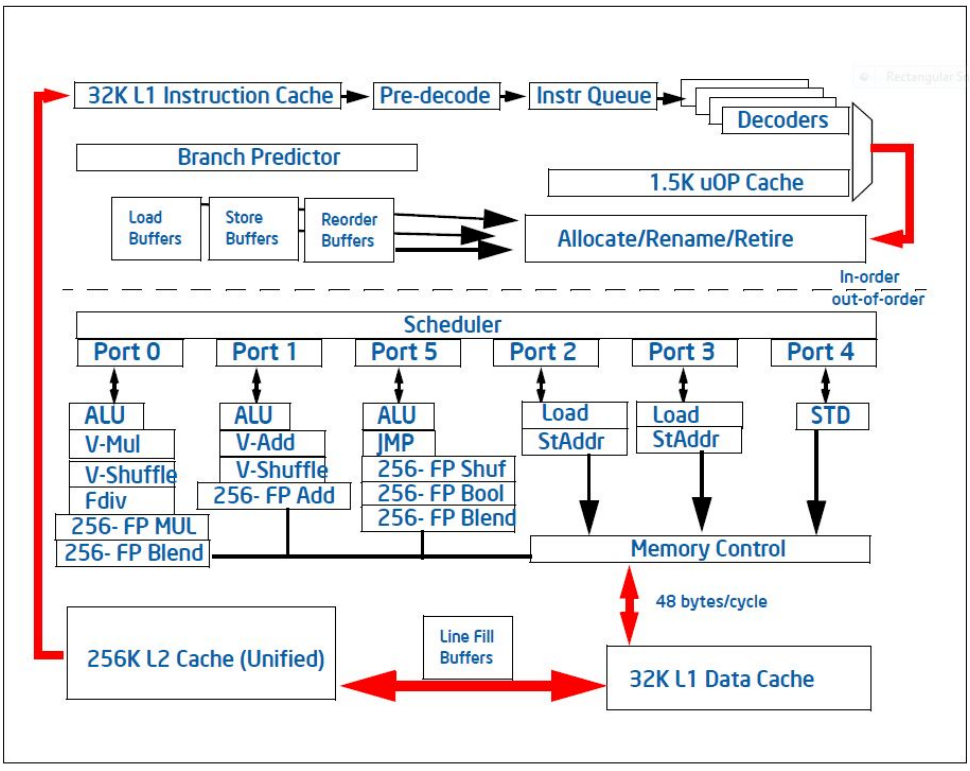


Figure 2-1. Intel microarchitecture code name Sandy Bridge Pipeline Functionality

顺便说一句，很少有用除法当制约因素的算法。在现代CPU中，除法操作的优化程度远不及加法和乘法。除法操作可能需要10或20个时钟周期，而CPU可以有多个加法和/或乘法单元（渐进地）使之每个周期都可以产生一个结果。下表为多个处理器架构的浮点能力(每个核心)，以及8个操作数的DAXPY周期数

处理器	年份	加/乘/乘加混合 单元(个数×宽度)	daxpy cycles(arith vs load/store)
MIPS R10000	1996	$1 \times 1 + 1 \times 1 + 0$	8/24
Alpha EV5	1996	$1 \times 1 + 1 \times 1 + 0$	8/12
IBM Power5	2004	$0 + 0 + 2 \times 1$	4/12
AMD Bulldozer	2011	$2 \times 2 + 2 \times 2 + 0$	2/4
Intel Sandy Bridge	2012	$1 \times 4 + 1 \times 4 + 0$	2/4
Intel Haswell	2014	$0 + 0 + 2 \times 4$	1/2

流水线

处理器的浮点加乘单元是流水线式的，其效果是独立的操作流可以以每个时钟周期一个结果的渐近速度执行。流水线背后的思想如下：假设一个操作由若干个简单操作组成，并且每个子操作在处理器中都有独立的硬件。如果我们现在有多个操作要执行，我们可以通过让所有的子操作同步执行来获得加速：每个操作将其结果交给下一个操作，并接受前一个操作的输入。

注释4 这与排队救火的过程非常类似，是一种收缩算法。

例如，加法指令可以包含以下组件：

- 指令解码，包括查找操作数的位置。
- 将操作数复制到寄存器中（数据获取）。
- 调整指数；加法$.35 \times 10^{-1} + .6 \times 10^{-2}$变成$.35 \times 10^{-1} + .06 \times 10^{-1}$。
- 执行尾数的加法，在这个例子中是$.41$。
- 将结果归一化，在本例中为$.41 \times 10^{-1}$。(本例中归一化并未执行任何操作，$.3 \times 100 + .8 \times 100$和$.35 \times 10^{-3} + (-.34) \times 10^{-3}$做了很大的调整)

这些部分通常被称为流水线的“**「深度」**（stages）”或“**「阶段」**（segments）”。

如果按照顺序执行，上文中每个组件设计为1个时钟周期，则整个过程需要6个时钟周期。然而，如果每个操作都有自己对应的硬件，我们就可以在少于12个周期内执行两个操作：

- 第一个操作执行解码；
- 第一个操作获取数据，同时第二个操作执行解码；
- 同时执行第一操作的第三阶段和第二操作的第二阶段；
- 以此类推。

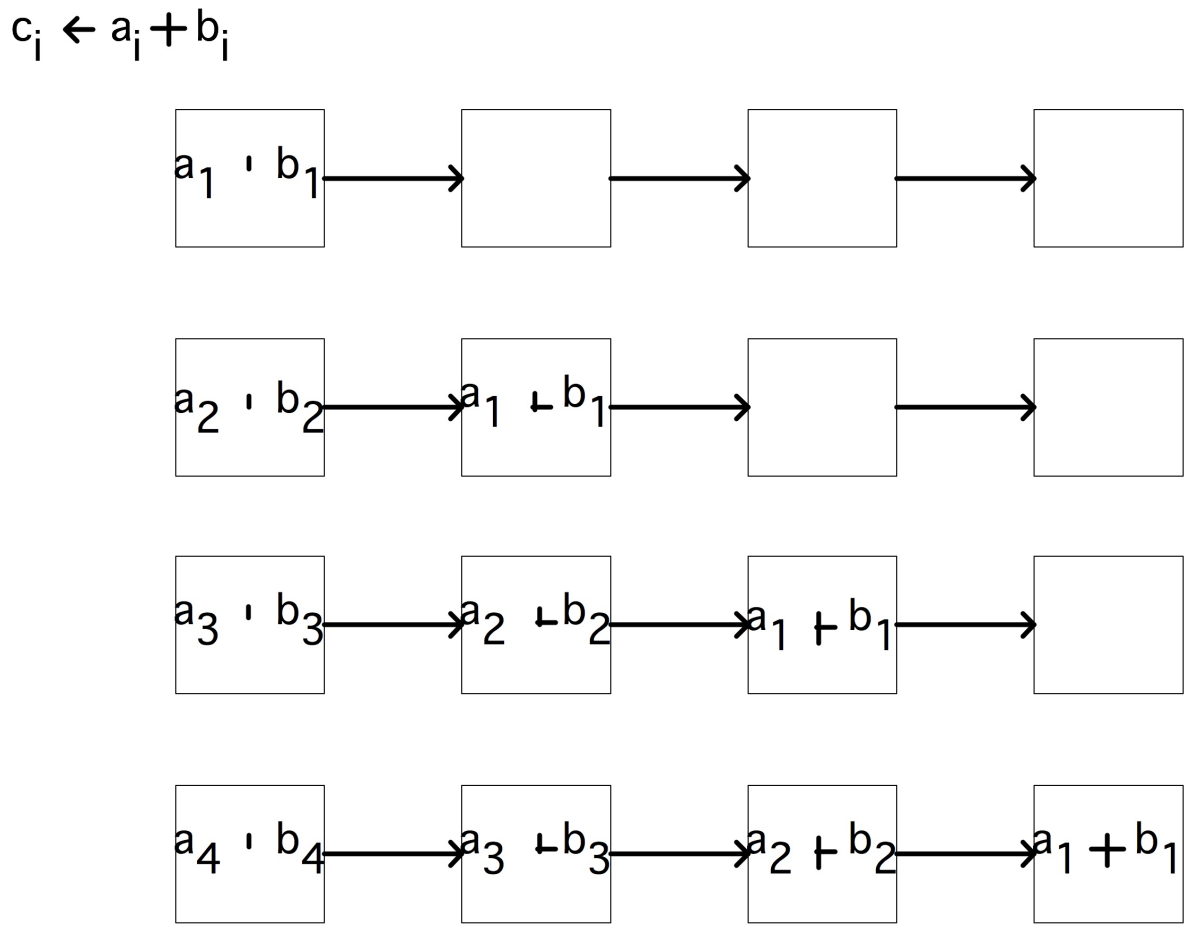
可以看到，第一个操作仍然需要6个时钟周期，但第二个操作只需再延后一个周期就能同时完成。

对流水线获得的加速做一个正式的分析：在传统的浮点单元上，产生n个结果需要花费时间为 $t(n) = n\ell\tau$，其中ℓ是状态个数，而τ是时钟周期。结果产生的速率是$t(n)/n$的倒数：$r_{serial} \equiv (\ell\tau)^{-1}$

另一方面，对于流水线的浮点单元，时间是$t(n) = [s + l + n - 1]t$其中s是设置成本；第一次执行时必须要经历一个完整的串行阶段，但在此后，处理器将在每个周期获得更多的收益。可以记作公式

$$t(n) = [n + n_{1/2}]\tau$$

表示线性时间，加上偏移量。流水线示意图描述如下：



练习1.1 请对比传统FPU和流水线FPU的速度差异。证明结果速率依赖于n：给出$r(n)$和$r_\infty = \lim_{n\to\infty} r(n)$的公式。在非流水线情况下$r$的加速极限是什么样？它需要多长时间才能接近极限情况？注意到$n = n_{1/2}$，可以得到$r(n) = r_\infty/2$，这通常被用做$n_{1/2}$的定义。

由于向量处理器同时处理多个指令，因此这些指令必须是独立且相互之间没有依赖关系的。$\forall_i : a_i \leftarrow b_i + c_i$有独立的加法运算；$\forall_i : a_{i+1} \leftarrow a_i b_i + c_i$将一次迭代$(a_i)$的结果输入到下一次迭代的输入$(a_{i+1} = \dots)$，所以这些操作并非独立。

与传统的CPU相比，流水线处理器可以将操作速度提高4、5甚至6倍。在上世纪80年代，当第一台向量计算机成功上市时，这样的加速效率十分常见。现在，CPU可以有20个阶段的流水线，是否意味着它们的速度非常快？这个问题有点复杂。芯片设计者不断提高主频，流水线部分不再能够在一个周期内完成他们的工作，所以他们进一步分裂。有时甚至有一些时间片段什么也没有发生：但这段时间是又是必须的，以确保数据可以及时传输到芯片的不同部分。

人们能从流水线CPU得到的改进是有限的，为了追求更高的性能，计算机科学家们尝试了几种不同的流水线设计。例如，Cyber 205有单独的加法和乘法流水线，可以将一个流水线输入另一个流水线，而无需先将数据返回内存。像$\forall_i : a_i \leftarrow b_i + c \cdot d_i$这样的操作被称为“**「链接三元组」**（linked triads）”（因为到内存的路径数量，一个输入操作数必须是标量）。

练习1.2 分析链接三元组的加速和$n_{1/2}$。

另一种提高性能的方法是使用多个相同的流水线。NEC SX系列完善了这种设计。例如，有4条流水线时，$\forall_i : a_i \leftarrow b_i + c_i$操作将对模块4进行拆分，以便第一条流水线对索引$i = 4 \cdot j$操作，第二条流水线对索引$i = 4 \cdot j + 1$操作，以此类推。

练习1.3 分析具有多个并行操作流水线处理器的速度提升情况和$n_{1/2}$。也就是说，假设有p个执行相同指令的独立流水线，每条流水线都可以处理的操作数流。

（你可能想知道我们为什么在这里提到一些相当老的计算机：真正的流水线超级计算机已经不存在了。在美国，Cray X1是该系列的最后一款，而在日本，只有NEC还在生产。然而，现在CPU的功能单元是流水线的，所以这个概念仍然很重要。）

练习1.4 如下操作

```
for (i) {
  x[i+1] = a[i]*x[i] + b[i];
}
```

不能由流水线处理，因为在操作的一次迭代的输入和前一次迭代的输出之间存在依赖关系。但是，我们可以将循环转换为数学上等价的循环，并且可能更有效地计算。导出一个表达式，该表达式从$x[i]$中计算$x[i+2]$而不涉及$x[i+1]$。这就是所谓的「**递归加倍**」（recursive doubling）。假设有足够的临时存储空间。参考如下：

- 做初步计算；
- 计算$x[i], x[i+2], x[i+4], \ldots$，并从这些中
- 计算缺失项$x[i+1], x[i+3], \ldots$

通过给出$T_0(n)$和$T_s(n)$的计算公式，分析了该格式的有效性。你能想到为什么初步计算在某些情况下可能不那么重要吗?

收缩计算

上面描述的流水线操作是「**收缩算法**」（systolic algorithm）的一种情况。在20世纪80年代和90年代，有研究使用流水线算法并构建特殊硬件——「**脉动阵列**」（systolic arrays）来实现它们[125]。这也与「**现场可编程门阵列**」(Field-Programmable Gate Arrays，FPGA)的计算连接，其中脉动阵列是由软件定义的。

峰值性能

现代CPU由于流水线的存在，时钟速度和峰值性能之间存在着较为简单的关系。由于每个FPU可以在一个周期内产生一个结果，所以峰值性能是时钟速度乘以独立FPU的数量。浮点运算性能的衡量标准是“**「每秒浮点运算」**（floating point operations per second）”，缩写为flops。考虑到现在计算机的速度，你会经常听到浮点运算被表示为“gigaflops”：10^9次浮点运算的倍数。

8位，16位，32位，64位

处理器的特征通常是可以处理多大的数据块。这可以联系到

- 处理器和内存之间路径的宽度：一个64位的浮点数是否可以在一个周期内加载，还是分块到达处理器。
- 内存的寻址方式：如果地址被限制为16位，只有64,000字节可以被识别。早期的PC有一个复杂的方案，用段来解决这个限制：用段号和段内的偏移量来指定一个地址。
- 单个寄存器中的数值位数，特别是用于操作数据地址的整数寄存器的大小；参见前一点。(浮点寄存器通常更大，例如在x86体系结构中是80位。)这也对应于处理器可以同时操作的数据块的大小。
- 浮点数的大小：如果CPU的算术单元被设计成有效地乘8字节数（“双精度”；见3.2.2节），那么一半大小的数

字（“单精度”）有时可以以更高的效率处理，而对于更大的数字（“四倍精度”），则需要一些复杂的方案。例如，一个四精度的数字可以由两个双精度的数字来模拟，指数之间有一个固定的差异。

这些测量值不一定相同。例如，原来的奔腾处理器有64位数据总线，但有一个32位处理器。另一方面，摩托罗拉68000处理器（最初的苹果Macintosh）有一个32位CPU，但16位的数据总线。

第一个英特尔微处理器4004是一个4位处理器，它可以处理4位的数据块。如今，64位处理器正在成为标准。

缓存（Caches）：芯片上的内存

计算机内存的大部分是在与处理器分离的芯片中。然而，通常有少量的片上内存（通常是几兆字节），这些被称为「**高速缓存**」（cache）。后面我们将会详细解释。

图形、控制器、专用硬件

“消费型”和“服务器型”处理器之间的一个区别是，消费型芯片在处理器芯片上花了相当大的空间用于图形处理。手机和平板电脑的处理器甚至可以有专门的安全电路或mp3播放电路。处理器的其他部分专门用于与内存或I/O子系统通信。我们将不在本书中讨论这些方面。

超标量处理和指令级并行性

在冯·诺伊曼模型中，处理器通过控制流进行操作：指令之间线性地或通过分支相互跟踪，而不考虑它们涉及哪些数据。随着处理器变得越来越强大，一次可以执行多条指令，就有必要切换到数据流模型。这种超标量处理器分析多个指令以找到数据相关性，并行执行彼此不依赖的指令。

这个概念也被称为「**指令级并行**」(Instruction Level Parallelism，ILP)，它被各种机制推动：

- 多发射（multiple-issue）：独立指令可同时启动；
- 流水线（pipelining）：上文提到，算术单元可以在不同的完成阶段处理多个操作；
- 分支预测和推测执行（branch prediction and speculative execution）：编译器可以“预测”条件指令的值是否为真，然后相应地执行这些指令；
- 无序执行（out-of-order execution）：如果指令之间不相互依赖，并且执行效率更高，则指令可以重新排列；
- 预取（prefetching）：数据可以在实际遇到任何需要它的指令之前被推测地请求(这将在后面进一步讨论)。

在上面我们看到了浮点操作上下文中的流水线操作。事实上，不仅是浮点运算，当代处理器的整个CPU都是流水线的，任何类型的指令都将尽快被放入指令流水线中。注意，这个流水线不再局限于相同的指令：现在，流水线的概念被概括为同时 "在执行 "的任何指令流。

随着主频的增加，处理器流水线的长度也在增加，以使分段在更短的时间内可被执行。可以看到，更长的流水线有着更大的$n_{1/2}$，因此需要更多的独立指令来使流水线以充分的效率运行。当达到指令级并行性的极限时，使流水线变长（或者称为“更深”)将不再有好处。因此，芯片设计者们通常转向多核架构以更高效地利用芯片上的晶体管。

这些较长的流水线的第二个问题是：如果代码到达一个分支点（一个条件或循环中的测试），就不清楚要执行的下一条指令是什么。在该点上，流水线就会会停止。例如，CPU总是假设测试结果是正确的，因此采取了「**分支预测执行**」（speculative execution）。如果代码随后接受了另一个分支（这称为分支错误预测），则必须刷新流水线并重新启动。执行流中产生的延迟称为「**分支惩罚**」（branch penalty）。