第 1 章

CPU与编译器概论

1.1 高速路与人行道

近年来，台式机的 CPU 主频已达到 2~3GHz，就连 iPhone 等智能手机和便携式终端的 CPU 主频率亦可达到 0.5~1GHz。

这些 CPU，有如在高速路上奔驰的跑车一样。试想一下，如果高速路上遍布着红绿灯和人行道，那跑车的性能就不能完全发挥出来。

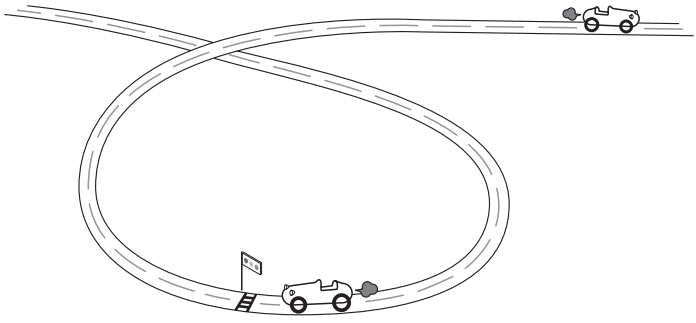


图 1–1　高速路与人行道

无论在高速路上跑得多快，一旦遇上红绿灯和人行道那该怎么办？想必就算是高性能跑车，在这个时候减速也会影响它的性能发挥。即使等到了绿灯，再次加速行驶，但遇到下一个红绿灯时也不得不再次减速。实际生活中的高速路上是没有人行道的，但计算机中的程序却是在“有红绿灯的高速路”上工作着。

读者在用 C 或 C++ 编写程序时，常常会在程序里设置很多“红绿灯和人行道”。虽然需要减速的原因有很多，但只要去掉其中几个主要的障碍，程序的运行速度就会提高几十倍。

那么，哪些程序在扮演着“高速路上的人行道”呢？我们该如何规避它？要想解答这些问题，我们就必须了解 CPU 的构造和工作原理，以及编译器运行的相关知识。

在这一章中，我们会在探讨性能优化的具体方法之前，先对 CPU 的构造与编译器的运行原理进行简单的讲解。

1.2 编译器是如何运作的

大多数程序员在日常编程中很少会直接用到 CPU 中的指令（即机器语言）。这主要是因为直接使用机器语言比较繁琐，所以我们选择人类更容易理解的语言来编程，然后再通过编译器将其翻译成机器语言。但是，编译器能否准确地将人类的逻辑思维转换为相应的机器语言呢？在这里，我们先来研究一下编译器到底是如何运作的。

比如，使用 GCC 按以下步骤将程序编译为目标代码（即汇编语言程序）。

1. 读取源程序并进行解析。将字符分离出来整理成较容易统计的形式，收集参数与函数名等标识符。
2. 对收集到的参数与标识符进行内存地址分配（即后文将提到的寄存器），将内存地址与参数或函数对应起来。
3. 根据逻辑程序生成汇编语言程序。

接下来，汇编编译器将已生成的汇编语言转换成机器语言的目标程序，链接器将目标程序和外部模块连接起来（图 1-2）。

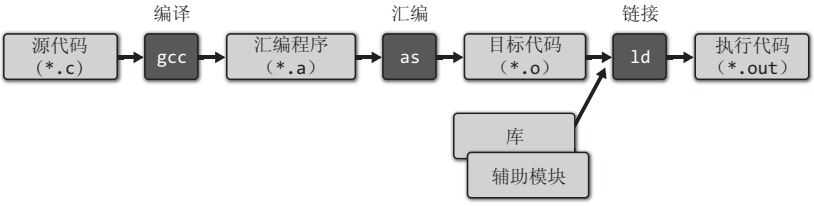


图 1–2　从源程序到执行代码的实现过程

近代的编译器实现了在编译过程中，让所生成的程序在更短的时间内得到相同的结果，达到更高的效率。其实现的方法多种多样，比如说，编译器扫描程序后，将多余的操作忽略，修改指令的运行顺序以使 CPU 处理得更快等。

优化与程序的调优有着密不可分的关联，在后面的章节中将会提到。

编译后的汇编语言程序

我们来看看由 GCC 生成的汇编语言程序。程序 1-1 是为检验而编写的小程序。

程序 1–1　10 次加 1 运算的程序

```
#include <stdio.h>
int a, b;
main()
{
    a = 0;
    do {
        b += a + 1;
        a++;
    } while (a < 11);
}
```

如果在编译此程序时加上 -S 选项，如“gcc -S test.c”，就会出现如程序 1-2 这样的汇编语言程序（该程序在 X86 系列 64 位环境下进行编译）。

程序 1–2　编译后的汇编语言程序（部分）

```
        .text
.globl main
        .type   main, @function
main:
.LFB2:
        pushq   %rbp
.LCFI0:
        movq    %rsp, %rbp
.LCFI1:
        movl    $0, a(%rip)      ……变量 a 赋值为 0
.L2:
        movl    a(%rip), %eax    ……变量 a 的值放到寄存器 %eax 中
        leal    1(%rax), %edx    ……（变量 a）+1 的值放入 %edx(%rdx 的低 32 位 ) 中
        movl    b(%rip), %eax    ……将变量 b 的值放入 %eax（%rax 的低 32 位）中
        leal    (%rdx,%rax), %eax   ……%rdx 和 %rax 的和放入 %eax 中
        movl    %eax, b(%rip)    ……%eax 的结果赋值给变量 b
```

```
        movl    a(%rip), %eax    ……变量 a 的值放入 %eax 中
        addl    $1, %eax         ……对 %eax 进行加 1 运算
        movl    %eax, a(%rip)    ……%eax 的结果赋值给变量 a
        movl    a(%rip), %eax    ……变量 a 的值放入 %eax 中
        cmpl    $10, %eax        ……将 %eax 的值与 10 进行比较
        jle     .L2              ……小于等于 10 的话跳到 L2
        leave                    ……释放栈中的变量
        ret                      ……程序跳出
.LFE2:
        .size   main, .-main
        .comm   a,4,4            ……分配给 a 以 4 为边界基准的 4 字节内存
        .comm   b,4,4            ……分配给 b 以 4 为边界基准的 4 字节内存
        …
```

大致上，左边是标签，中间是 CPU 指令，右边是操作目标。以“.”开始的字符串表示指定汇编程序集的函数名。以“:”结束的字符行是标识符的定义。函数名和标识符以外的部分是实际被执行的指令集，与 CPU 的机器语言相对应。以“%”开头的是寄存器名，以“$”开头的是常量。

“a(%rip)”表示外部变量 a，“b(%rip)”表示外部变量 b，外部变量引用以 %rip 标识的寄存器（程序计数器标识程序接下来该执行的指令）所指的地址中相对应的位置。

从标识符 .L2 到 jle 指令是程序的循环部分，相当于 C 语言源程序的“do~while(a<11)”。即使没有使用过汇编语言，一看到程序当中的备注也能大致理解其运行原理。

X86 系列 CPU 的寄存器

CPU 内部有若干个高速存储单元，也就是常说的寄存器，它可以用来储存数据，还可以作为指示其储存地址的指针来使用。在下面的表格中，我们列举了 X86 系列 CPU 中常用的寄存器。

一直以来，32 位的 CPU 中所使用的寄存器为 32 位寄存器，64 位 CPU 中则使用扩展到 64 位的寄存器，更甚者追加到寄存器 r8~r15。扩展后的 64 位寄存器可以在 64 位模式下使用。

在 64 位环境中，通过使用增加的寄存器，可完成程序中函数参数的传递。rax 被用于存放返回值，rdi、rsi、rdx、rcx、r8、r9 分别用于存放第 1~6 个整数型参数 ①。

在 32 位环境中，将函数写入栈中，这样可以缩短执行时间。

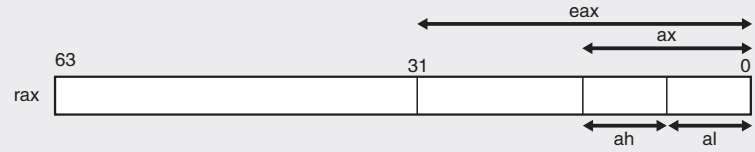
另外，对部分寄存器来说，寄存器的一部分也可进行数据存储和计算操作。比如，寄存器 rax 的低 32 位是 eax，eax 的低 16 位是 ax；ax 的高 8 位是 ah，ax 的低 8 位是 al 等。

x86/x64 系列 CPU 的主要寄存器

寄存器名（64 位环境）	用　途	寄存器名（32 位环境）	用　途
rax	第一函数值，操作用	eax	函数值，操作用
rbx	寄存器变量	ebx	寄存器变量
rcx	操作用，第四函数变量	ecx	操作用
rdx	操作用，第三函数变量，第二函数值	edx	操作用
rsi	操作用，第二函数变量	esi	寄存器变量
rdi	操作用，第一函数变量	edi	寄存器变量
r8	操作用，第五函数变量	—	
r9	操作用，第六函数变量	—	
r10~r11	操作用	—	
r12~r15	寄存器变量	—	
rbp	基指针，寄存器变量	ebp	基指针，寄存器变量
rip	程序计数器（提示下一条指令）	eip	程序计数器（提示下一条指令）

*1 函数变量仅在 64 位模式下使用。

*2 寄存器变量在某些时候被分配为操作用寄存器。



部分寄存器

① 参考：System V Application Binary Interface（http://www.x86-64.org/documentation/abi.pdf）

添加优化选项后的结果

想必即使是不熟悉汇编语言的读者也能看出，前面所生成的代码太过冗余。虽然在计算时将内存上变量 a 和 b 的内容复制到了寄存器 %eax 上，但是如果能将变量 a 和 b 分别放在不同的寄存器上，然后仅对寄存器进行操作，这样岂不是更合理？

接下来，我们将添加优化选项（-O），执行“gcc -S -O3 example.c”指令后再来编译看看。其结果如程序 1-3 所示，可以看得出所生成的代码更为简练。

程序 1–3　优化选项的效果

```
        .text
        .p2align 4,,15
.globl main
        .type   main, @function
main:
.LFB13:
        movl    b(%rip), %edx       ……变量 b 的值放到寄存器 %edx 中
        movl    $0, a(%rip)         ……变量 a 赋值为 0
        xorl    %eax, %eax          ……将寄存器 %eax 清零
        .p2align 4,,7
.L2:
        addl    $1, %eax            ……对寄存器 %edx 做加 1 运算
        addl    %eax, %edx          ……寄存器 %edx 与 %eax 相加
        cmpl    $10, %eax           ……将寄存器 %eax 的值与 10 相比
        jle     .L2                 ……小于等于 10 则跳到 L2
        movl    %edx, b(%rip)       ……将加法运算的结果赋值给变量 b
        movl    $11, a(%rip)        ……变量 a 赋值为 11
        ret
.LFE13:
        .size   main, .-main
        .comm   a,4,4
        .comm   b,4,4
        …
```

前面冗余的代码在循环内对外部变量逐一计算并赋值，但是在优化

后的代码中，我们用寄存器代替了外部变量，在最后灵活地将变量 a 赋值为 11，并且在循环代码前面使用了能缩短执行时间的逻辑操作指令 xorl（XOR），这也是为了提高效率。

这只是编译器进行优化的一个小小例子，但我们可以从中看出，直接生成的代码和优化后的代码之间存在哪些区别。

1.3 CPU 是如何运作的

上一节中我们介绍了程序经过编译器和汇编程序的转换后，最终得到机器语言的过程，也探讨了由 C 语言程序生成汇编语言的过程。下面我们再进一步探究 CPU 是如何执行机器语言程序的。

指令集架构与微架构

CPU 能够执行什么样的指令，或者说 CPU 所具备的指令集，称为 CPU 的指令集架构。

指令集架构是规定程序设计如何使用指令的规范，它包括寻址模式和寄存器构成、中断、异常处理等。所以，只要指令集架构是相同的，即使使用不同品牌的 CPU，所得出的结果也是一样的。比如，在上一节所展示的汇编语言程序中，只要是 x86_64 架构的 CPU，不管其厂家是 AMD 还是英特尔，结果都是一样的。

相反，如果指令集架构不同，那么 CPU 的指令也会相应发生变化。但是，不管使用的方法是否一样，几乎所有的 CPU 中都具备了以下基本指令。

- 算术运算
- 逻辑运算（AND，OR，XOR）
- 移位指令
- 条件比较指令
- 寄存器与内存之间的传送
- 跳转指令

另一方面，CPU 执行指令集架构的方法叫做微架构。各品牌为了使 CPU 更高效化，在设计上花费了不少功夫，所以即使是拥有同一指令集架构的 CPU，其性能和负荷方面的特点也是有所不同的。

高效编程的一个重要手段就是掌握 CPU 高效执行指令的方法。下面我们了解一下 CPU 的基本构造，以及它是如何执行指令的。

如何执行指令

所谓计算机，就是指按顺序执行分配到内存上的程序（指令的排列），并通过这一操作完成数据处理的机器。CPU 大致按以下顺序执行指令（图 1-3）。

1. 从内存读取指令（读取）
2. 解析所读取的指令（解码）
3. 提取操作目标的数据
4. 执行计算和条件对比等指令
5. 输出加工后的数据

图 1–3　指令的执行

第三步是从寄存器或者内存中读取出第四步执行指令时所需要的数据。

CPU 中有分管上述各功能的管理单元，解码负责支配执行指令时各单元的运行。第四步中的指令执行装置包含执行加减运算的算术逻辑单元（ALU）、逻辑运算器、乘法器、除法器、装载 / 记忆装置等单元，与所执行的指令结合使用。如下例，图 1-4 是在执行寄存器间的计算指令时各单元的运作情况，图 1-5 是在执行将内存数据装载到寄存器时各单元的运作情况。

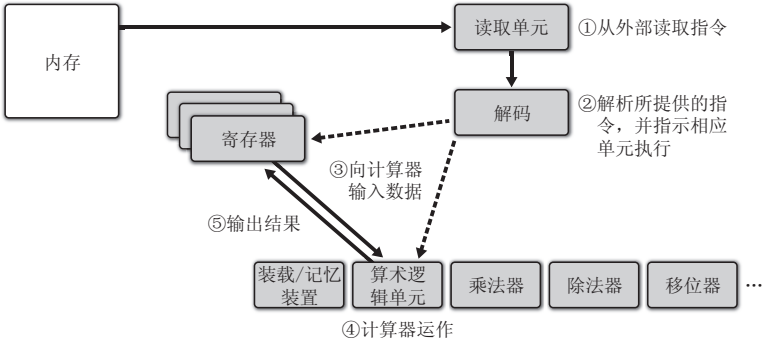


图 1–4　指令的执行（寄存器间的运算）

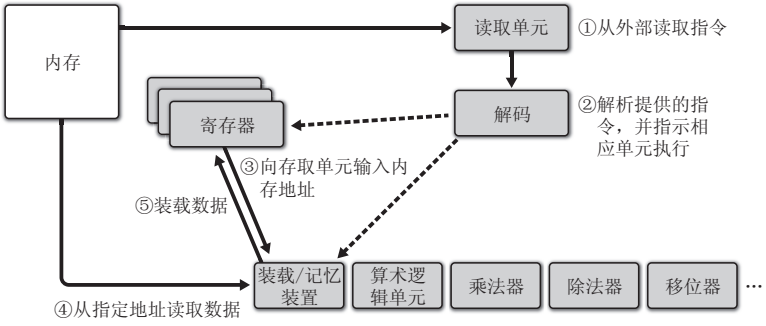


图 1–5　指令的执行（内存中数据装载到寄存器上）

想必读者在购买计算机的时候一定会参照 CPU 的主频来确定其性能。例如，在本书实验中所使用的英特尔 Xeon W5580，其主频为 3.2GHz，1 秒钟内能经历 32 亿个时钟周期。CPU 内的各单元可以通过获得时钟信号来进行同期运作，上述各单元中的读取、解码以及寄存器中的装载计算器等，会在 1 个时钟周期内完成一次动作（除法运算和内存之间的操作会花费更多的时间，在这里只是简单地举例）。

所以，将两个寄存器的值相加，假定写入寄存器的指令适用于图 1-4，则完成①～⑤这一连串操作需要花费 5 个时钟周期。那么执行两个指令岂不是要花 10 个时钟周期？不，如果让各个单元都高效地运作起来，其性能

是可以得到提升的。

指令流水线

假定现在，我们需要执行以下指令操作。

第一，将寄存器 a 的值复制到寄存器 c

第二，将寄存器 b 的值与寄存器 c 相加

第三，将寄存器 c 的值向左移动 2 位（值将变成原来的 4 倍）

第四，对寄存器 c 的值做加 1 运算

图 1-6 为各单元按顺序执行这一指令操作的过程。使用同样的 CPU 单元执行 4 条指令仅需要 8 个时钟周期，由此可见其执行性能得到很大的提升。

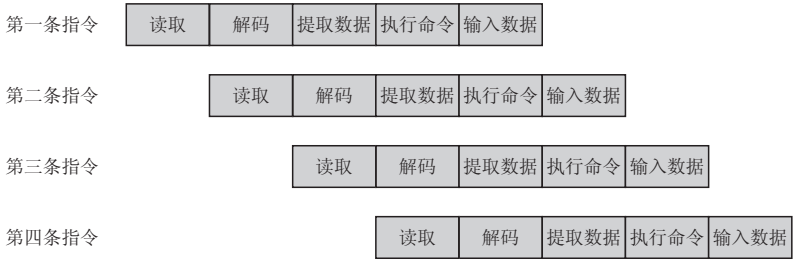


图 1–6　指令流水线

像这样利用 CPU 中各单元按流水作业的方式来执行指令，我们称之为流水线。如若众多指令能通过流水线形式来执行的话，CPU 的实际执行速度就能达到近乎 1 个时钟周期完成 1 条指令。

但实际上情况会复杂得多，比如执行乘除运算指令时需要花费数个时钟周期，而内存的读写则需要数十个乃至更长的时钟周期来完成。特别是内存读写比较频繁的时候，CPU 的性能会受到显著影响，所以我们需要引入缓存这一架构。

高速缓存

主存储器的存取速度相当缓慢，比 CPU 要慢上数十倍甚至数百倍。在与 CPU 所处的同一芯片内设计有高速小容量内存，它们的工作就是频繁复制使用中的主存储器，代替主存储器工作，我们将其称为高速缓存，由此可以缓和 CPU 与主存储器之间速度不匹配的矛盾。

表 1-1 为本书实验所使用设备的高速缓存型号及存取速度的相关内容。

表 1–1　实验设备的高速缓存型号及存取速度

内存的种类	等待时间（存取所需时间）
一级缓存（32KB、8 路）	4 个时钟周期
二级缓存（256KB、8 路）	未满 12 个时钟周期
三级缓存（8MB、16 路）	30~40 个时钟周期
主存储器	180~200 个时钟周期

CPU 在执行指令时需要从内存上获取数据，此时 CPU 首先会在缓存中查找相应的数据，在缓存内没有的情况下才会去读取主存储器上的数据（如图 1-7）。读取的数据将被调入高速缓存中，以便下次使用。所以，从第二次存取开始就能大大缩短时间。

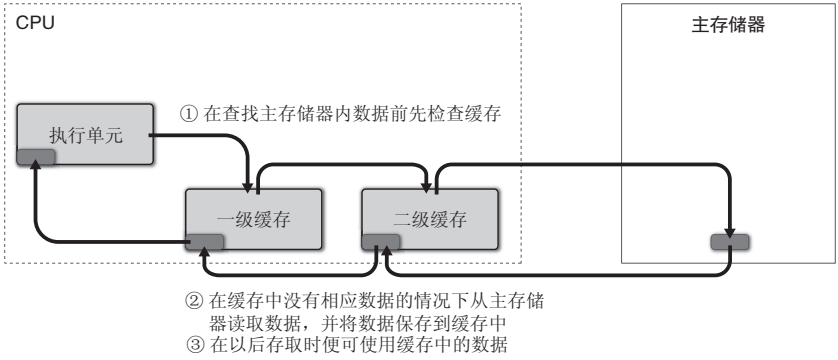


图 1–7　高速缓存

存储器容量越大，存取所需的时间就越多。在现代计算机中采用了高

速且小容量的一级缓存、比一级缓存容量稍大但存取速度稍慢的二级缓存，还有根据 CPU 而定的容量更大的三级缓存。

深入探讨高速缓存

关于如何快速从缓存中读取所需数据，我们接下来做进一步的探讨。

高速缓存与缓存块（也可称为缓存条或者缓存行）被分区管理。块的型号因 CPU 的构造有所不同，Xeon W5580 的一级缓存的型号为 64 字节。主存储器与缓存，或者是上层缓存与下层缓存之间的数据传送，均由块单位来执行。也就是说，当 CPU 想读取出 4 字节的数据时，在缓存里没有该数据的情况下，将从外部内存复制已包含相应地址数据的块大小单位。

例如，假设我们有一个由 1024 个 64 字节的块构成的 64KB 的缓存。CPU 在访问地址 32008 数据的时候，首先确认是否有以 32000 地址开始的 64 字节的缓存数据，如若没有，则需从外部内存中获取。

如果能将内存中访问地址的一部分作为索引来使用的话，那么获取缓存内块地址就相对简单。与缓存相比，主存储器的容量较大，所以假设缓存容量为 64KB，那么主存储器则分为 64KB 大小的段，使其分别对应于缓存中的块（如图 1-8、图 1-9）。

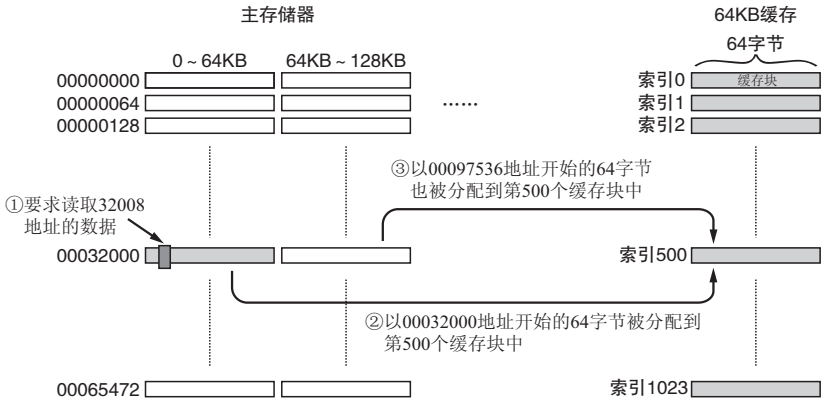


图 1–8　直接映射方式的缓存分配

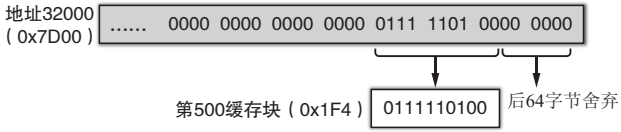
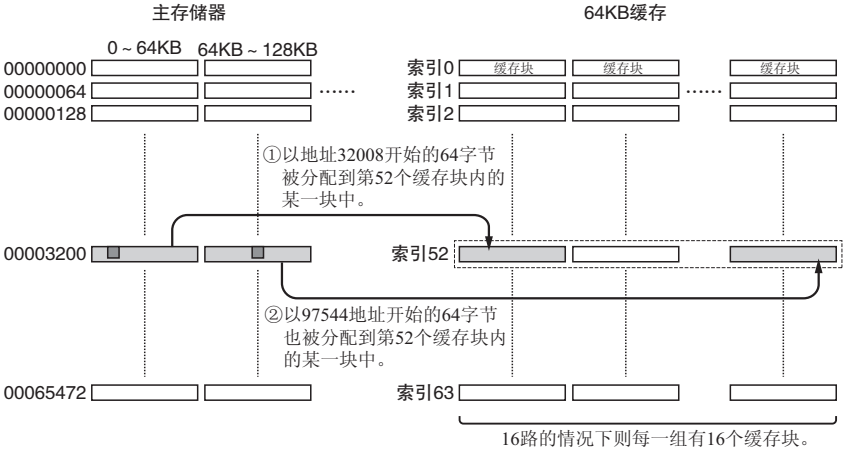


图 1–9　将内存访问地址的一部分作为索引使用

但是，这种方法会将像 32000、97536、163072 等间隔为 64KB（65536 字节）的数据放在同一缓存块中。因此在有以上数据内存存取倾向的程序中，缓存块的替换会变得频繁，导致性能降低。但是如果我们准备若干个缓存块组，从检索序列号相同的缓存块中寻找空白块并进行分配，就能缓解性能降低的问题。这就是组相联映射方式（Set Associative），常被应用到数据缓存当中（图 1-10）。



在相同容量的缓存中采取组相联映射方式的话，列方向的缓存块组数就会减少。那么16路组相联的方式，缓存块的序列数会是直接映射方式的1/16。

图 1–10　组相联映射方式的缓存分配

缓存中每一组的缓存块数目（链数、路数）需要结合缓存的命中率来设计，在 x86 系列的 CPU 中一级缓存需要 2~8 个缓存块，二级缓存则需要

4~8 个，三级缓存需要 4~16 个。

缓存块的替换算法

缓存中的数据随着程序的执行而不断更新。理想的情况是将经常访问的数据放在缓存中，但要想预测哪些数据常会被使用是比较困难的。

当缓存中没有空白块可分配时，就有必要置换出缓存中的某一块数据。我们依据各种置换策略决定置换哪一块数据，常用的置换策略如下。

LRU（Least Recently Used）：最近最少使用的数据将被置换出缓存。这是基于“之前未被使用则以后也不会被使用”的考虑。因为需要记录缓存块的使用顺序，所以路数的增加就会导致缓存设计变得复杂。

PLRU（Pseudo-LRU）：一段时间内未使用的缓存块作为替换候补，在需要替换时将某一候补替换出去。我们对某一时间段内使用过的缓存块进行标记，在需要置换的时候将未标记的某一缓存块清理出去，通过这一简单的方法即可实现 PLRU 替换操作。

FIFO（First In First Out）：置换最先放入缓存的缓存块。

RANDOM：随机置换。

超标量指令执行

如若进一步探究 CPU 的高效化会发现，增加解码器与计算器等执行装置的数量，将若干指令并行执行，这也是一个可行之法。若干解码器安排各种 CPU 单元的运作并促进程序的执行，通过这一操作可以实现多条指令并行执行。这个方法虽然在控制上比较复杂，但如果能操作好，则可实现 1 个时钟周期内完成多条指令。灵活使用此类架构，首先执行非依赖指令，就能达到不按顺序执行也能得到相同结果的目的。

以上，我们走马观花地探讨了 CPU 如何执行指令，以及采用怎样的方法来实现高效的操作。当然还有更为复杂的要素和手法将在之后的章节中进行说明。如果能充分理解这一章的内容，接下来的内容就会比较容易理解了。

SRAM 与 DRAM

高速缓存的存取速度较之主存储器要快得多，这是因为它们的内存元件有所不同。高速缓存中所使用的 SRAM（Static Random Access Memory）是由晶体管（6~8 个晶体管构成 1 位）构成的逻辑电路组成的。逻辑电路的速度的确很快，但因为晶体管的使用数量较多，从而导致芯片上的容量相应减少。

另一方面，主存储器常用的 DRAM（Dynamic Random Access Memory）则由电容器（容纳电荷的器件）构成，通过储存在电容器中的电荷多寡来表现其状态。根据电容器的充放电来进行读取时速度较慢，特别是在放电时会破坏储存状态，所以需要不断刷新已读出的数据（即重新充电）。但是，如果将电容器垂直（朝芯片较厚的方向）设计的话，每 1 位相当于 1 个晶体管大小，因此相同芯片面积上容量较大。

从内存读取数据，CPU 需要向内存发送数据地址和数据读取要求，然后内存收到指令后输出数据。

存储容量较大的 DRAM，因为被用来表示地址的位数增加，所以需要通过高位低位两条指令来向内存读取数据，从而导致访问速度迟缓。但是对内存里连续存储的数据操作，高位地址只需传送一次，然后只传送低位地址即可读取数据。这样一来传送速度就提高了。缓存与主存储器之间的数据传送是通过 32~64 字节的缓存块来执行的，因此传送速度的快慢至关重要。

第 1 章是不是偏离了主题


本书一开始就探讨 CPU 内部的各种细节，是否偏离了主题呢?


表面看起来是这样，但通过第 1 章能让读者了解到计算机内部程序是如何运作的，这叫“磨刀不误砍柴工”。


的确是这样，这些内容如果在之后的章节中出现会使读者感到迷茫，无法理解。不过，一定要避免纸上谈兵噢，就比如英语口语培训班只是教给学生发音和一些成语，学生并不能学到实用的知识。


你这是亲身经历过还是……


那倒不是，只是担心所学到的并不能在实际生活中得到应用。


那打个比方，如果 F1 赛车手不了解赛车的话，那他估计也没有出赛的资格吧。


哦，是啊。就比如说，倘若我的车只是购物的代步工具，那我不需要了解太多关于车的机械知识，但如果职业赛车手想要把赛车的性能发挥到极致的话，不了解机械知识是办不到的。


在本书之后的章节中，我们会通过实验来了解当变量增加时所需计算器的数量、缓存具有怎样的特性，其特性会对性能的发挥产生怎样的影响等内容。在实验中得出的数据是微不足道的，因为随着编程环境的变化，数据也会产生变化。


嗯……那您的意思是其中另有深意?


倒也不是，只是希望大家在本书的学习中了解到如何去实现所写的代码，怎样的操作会导致编程成本高。

也就是说“灵感是源于体验”，对吧?

就像编译器的后台处理过程对程序员是不可见的，但也可以通过其他手段来了解。在本书中我想向大家呈现这样的思维方式。

不了解 CPU 的内部构造和编译器的运作方式，就如同职业赛车手在比赛时报不上自己的名字一样。作为职业赛车手，精通机械知识是理所当然的。本书的第 1 章所揭示的就是这个道理。

是这样的。