Computer Systems

A Programmer's Perspective(3rd)

正式开始 CSAPP 了，有点儿小兴奋。据说这本书是国外许多著名大学的经典教材，深入讲解了计算机系统的底层原理，包括编码，汇编软件，指令，存储器层次结构，虚拟内存等。看完应该会很有收益，那事不宜迟，正式开始吧^<^。

参考机：Intel Core i7 Haswell

一、 计算机系统漫游

1.1 位+上下文

源文件 hello.c 本质上是由 0 和 1 组成的位（又叫做比特）序列，8 个位被组成一组，叫做字节。目前大多数计算机系统采用 ASCII 表示文本字符。
系统中的所有信息其实都是一串比特序列。相同的比特序列在不同的上下文信息中表示的意义不同，它可能是一个整数，浮点数，字符串或者是一个机器指令。

1.2 编译系统（Compilation System）

编译系统由预处理器，编译器，汇编器和链接器一起构成：
源程序（文本）hello.c
——>预处理器(cpp)——> 经过修改的源程序（文本）hello.i
——>编译器(ccl)——>汇编程序（文本）hello.s
——>汇编器(as)——>可重定位目标程序（relocatable object program 二进制）hello.o
——>链接器(ld)——>可执行目标程序（二进制）hello.out

选项 -E
gcc -E test.c -o test.i
将 test.c 预处理输出 test.i 文件。

选项 -S
gcc -S test.i
将预处理输出文件 test.i 汇编成 test.s 文件。

选项 -c
gcc -c test.s
将汇编输出文件 test.s 编译输出 test.o 文件。

无选项链接
gcc test.o -o test
将编译输出文件 test.o 链接成最终可执行文件 test。

选项 -Og
Gcc -Og test.c -o test
//告诉编译器使用符合原始 C 代码整体结构的机器代码的优化等级，使用较高的优化登记会使得代码严重变形，以至于产生的及其代码和原始代码之间的关系难以理解。实际上，-O1 或者-O2（指定）被认为是较好的优化级别。
了解编译系统后，你可以：

- 优化程序的性能；
- 理解链接时出现的错误；
- 避免安全漏洞；
  多年来，缓存区的溢出是造成大多数网络和 Internet 服务器安全漏洞的主要原因。

1.3 系统的硬件组成

1) 总线
   贯穿整个系统的电子管道，通常总线被设计成传送定长的字节块，叫做字（word）,通常每个系统中字包含的字节数是不同的，有的是 4 字节，即 32 位，有的是 8 字节，即 64 位。
2) I/O 设备
3) 主存
   即内存，是由一组动态随机存取存储器（DRAM）芯片组成。
4) 处理器
   从主存中读取指令，解释指令中的位，执行该指令指示的简单操作，然后更新 PC（Program Counter）使其指向下一个指令。两条指令在内存中的位置不必相邻。

两种描述观点：

- 指令集架构描述：每条机器代码指令的效果；
- 微体系结构描述：处理器实际上是如何运行的；

1.4 不同层次的储存设备

L0：处理器的寄存器
L1：L1 高速缓存（SRAM）

L2：L2 高速缓存（SRAM）
L3：L3 高速缓存（SRAM）
L4：L4 主存（DRAM）
L5：L5 本地二级存储（本地磁盘）
L6：L6 远程二级存储(分布式文件系统，Web 服务器)

层次越高，容量越小，造价越贵，存取速度越快；
储存器层次结构的主要思想是上一层次的存储器是下一层次的高级缓存。

1.5 操作系统管理硬件

可以把操作系统看作是应用程序和硬件之间插入的一层软件，所有的应用欲要对硬件进行操作必须经过操作系统。
操作系统有两个基本功能：

1) 防止硬件被失控的应用程序滥用；
2) 向应用程序提供简单一致的机制来控制复杂而又通常大不相同的低级硬件设备；

三个抽象概念：

1) 文件
   文件是对 I/O 设备的抽象表示。
2) 虚拟内存
   对主存和磁盘 I/O 设备的抽象表示。
3) 进程
   对处理器，主存和磁盘 I/O 设备的抽象表示。所谓进程，就是操作系统对一个正在运行的程序的一种抽象。

其他几个重要的概念：

1) 并发运行：一个进程的指令与另一个进程的指令是交错执行的。无论是单核还是多核系统，一个 CPU 看上去像是同时进行多个进程，但实际上是通过处理器在进程间切换来实现的，这种交错的机制叫做上下文切换。
2) 上下文：操作系统保持跟踪进程运行所需的所有状态信息；
3) 上下文切换：操作系统决定把控制权从当前进程转移到某个新的进程，即保存当前进程的上下文，恢复新进程的上下文，然后将控制权传递到新进程，新进程就会从他上次停止的地方开始。
4) 内核：从一个进程切换到另一个进程是由系统内核（kernel）管理的，内核是操作系统常驻内存的部分。注意，内核不是一个独立进程，相反，它是系统管理全部进程所用代码和数据结构的集合。
5) 线程：一个进程中可以由多个称为线程的执行单元组成，每个线程都运行在进程的上下文中，共享同样的代码和全局数据。
6) 虚拟内存：位于栈区的上方，用来储存操作系统的代码和数据的，不允许程序读写这一块的内存。

7) 文件：就是字节序列，仅此而已。
8) 并发（concurrency）和并行（parallelism）
   并发：指的是同时具有多个活动的系统；
   并行：使用并发使得一个系统运行得更快；
   - 线程级并发：从多核到多核多线程，比如四核八线程，采用的是超线程的技术，即在一个核中，程序计数器和寄存器等加倍，其他硬件不变。
   - 指令级并行，处理器同时执行多条指令的属性。如果一个处理器能够在一个周期内执行超过 1 条指令，则称为超标量（super-scalar）处理器。
   - 单指令和多数据并行：一条指令可以产生多个并行执行的操作，即 SIMD 并行性。

1.6 Amdahl 定律

Amdahl 定律，主要思想是当我们对系统的某个部分加速的时候，其对系统的整体性能提升取决于该部分的重要性和影响。假设占比 a，效率提升 k，则总时间提升：

$$\eta = \frac{T_{old}}{T_{new}} = \frac{T_{old}}{(1-a) \times T_{old} + a \times T_{old}/k} = \frac{1}{(1-a) + {}^{a}/_{k}}$$

性能提升最好的表示方法是$\frac{T_{old}}{T_{new}}$，如果性能有提升则大于 1，用“×”表示比例，写作“η ×”，都作“η倍”。

二、 程序结构和执行

2.1 字数据的大小

字，word，等于指针数据的标称大小（nominal size）,字的排列组合方式的总数等于虚拟内存的地址数，这里应该是一个组合方式制定一个虚拟内存的地址。对于一个字长为 w 位的机器，其虚拟内存地址为 0~2^w-1。如 32 为字长的系统就限制了虚拟内存大小为 4GB。
32 位的 long 类型和指针都占 4 个字节，64 位的 long 类型和指针类型都占 8 个字节

2.2 寻址和字节顺序

在几乎所有机器上，多字节对象都被存储为连续的字节序列，对象的地址为所使用字节中最小的地址。
排列一个对象的字节有两种方式，如储存 int 类型的 0x01234567，首地址为 0x100：
- 小端法（little endian）
  最低有效字节在最前面的方式

0x100	0x101	0x102	0x103
67	45	23	01

- 大端法（big endian）

  最高有效字节在最前面的方式

0x100	0x101	0x102	0x103
01	23	45	67

采用何种方式并没有孰优孰劣，但是目前用的最多的是小端法，两种方法都是用 16 进制输入和输出。小段法有一个好处就是发上截断的时候能保留低位，去掉高位，尽量保证数据的完整性。

加减法的优先级比位移运算符要高。
算术右移：用 0 填充；
逻辑右移：用 1 填充；
假设由 w 位组成的数据类型，如果移动的位数 k>=w 会出现什么后果呢？C 语言很小心地规避了这个问题，移位指令只会考虑位移量的低$\log_2 w$位，或者用 k mod w 的结果，又或者用 k%w 的余数表示。

2.3 补码

以前我们认识的补码计算是：如果转为二进制表示后最高位有 1，则取反码后加 1，转为十进制后添上符号则为该值。
但是新的计算方法是，最高位有 1 的话在二进制转十进制过程中，最高位填上符号，如：1011 则为

$$-1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = -5$$

$$\mathrm{B2}T_w(\vec{x}) = -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i\, 2^i$$

反汇编器是将二进制文件转换为 ASCII 码的程序，用的也是补码。

补码非对称性：

$$|T_{Min}| = |T_{Max}| + 1$$

也就是说，T_{Min}并没有与之对应的正数，因为 0 包含在非负数的范围中，这个小细节很容易会让人犯错。最大的无符号值正好比补码的最大值的两倍多 1。补码中所有的负数的位模式在无符号表示中都编程了正数。不过奇怪的是，-1 和 UMAX 有着相同的位表示——一个全 1 的串，其中-1 和 0 的表示要格外小心。
补码转换为无符号数：

$$\mathrm{T2}U_w(x) = x + 2^w(\mathrm{x} < 0)$$
$$\mathrm{T2}U_w(x) = x(\mathrm{x} \geq 0)$$

无符号数转换为补码：

$$\mathrm{U2}T_w(u) = u(\mathrm{x} \leq \mathrm{T}_{max})$$

$$T_w(u) = 2^w - \mathrm{u}(\mathrm{u} > \mathrm{T}_{max})$$

记忆：无符号数在相同位的情况下跟负数的补码相差 $2^{w-1} - (-2^{w-1}) = 2^w$

在 C 语言中，假如一个表达式中同时存在有符号和无符号两种格式，编译器则会隐式地将有符号的格式转换为无符号的格式，并假设都是非负的。这样子的话计算的时候没有多大的问题，但是在判断的时候很容易出错，比如在 32 位的系统中

$$-1 < 0\mathrm{U}$$

本来是真的，但是两变变成无符号之后，-1 就变成了无符号中的最大值，显然判断的结果为 false。

在有符号中我们习惯将最小值写成

$$T_{Min} = -T_{Max} - 1$$

的形式，就是为了凸显补码的不确定性。

2.4 拓展一个数字的位表示

将一个数转化为位数更多的数据类型的时候，只要简单的在前面添加零便可，这成为零拓展（zero extension）

有符号的补码前面补 1，或者使用公式

$$\mathrm{B2}T_w(\vec{x}) = -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i\, 2^i$$

只是位数 w 改为拓展后的位数。

如果是从位数多的数据类型转化为位数少的数据类型，则发生截断，截断前面的高位。

2.5 整数运算

1) 无符号加法

$$\mathrm{x} +{}_w^u y = x + y \ \ (x + y \le 2^w - 1)$$

$$\mathrm{x} +{}_w^u y = x + y - 2^w \ \ (x + y \ge 2^w)$$

我的直觉：减去2^w其实就是相当于把 w+1 位上的 1 去掉，这是应对会溢出的情况，当然了，如果其和不溢出的话就不需要去掉该位了。事实上确实是这样。

算术溢出：指完整的整数结果不能放到数据类型的字长限制中去。

溢出检测：设

$$\mathrm{s} = \mathrm{x} +{}_w^u y$$

当$\mathrm{s} < \mathrm{x}$的时候表示溢出。

推导：当发生溢出的时候，

$$s - x = y - 2^w < 0$$

即

$$s < x$$

这跟事实

$$s > x$$

不符，所以当$s < x$的时候表示发生溢出。

阿贝尔群（Abelian group）：模数加法形成的一种数学结构，他是可交换和可结合的，他有一个单位元 0，并且每个元素有一个加法逆元。百度百科上的定义是亦称交换群。一种重要的群类。对于群 G 中任意二元 a，b，一般地，ab≠ba.若群 G 的运算满足交换律，即对任意的 a，b∈G 都有 ab=ba，则称 G 为阿贝尔群。由于阿贝尔(Abel，N.H.)首先研究了交换群，所以通常称这类群为阿贝尔群。交换群的运算常用加法来表示，此时群的单位元用 0(零元)表示，a 的逆元记为-a(称为 a 的负元).用加法表示的交换群称为加法群或加群。

无符号数求反；

$$-_w^u x = x \ (x = 0)$$
$$-_w^u x = 2^w - x \ (x > 0)$$

2) 补码加法；

$$x +_w^t y = x + y + 2^w \ \ (x + y < -2^{w-1}, \text{负溢出，} Passive\ Overflow)$$

$$x +_w^t y = x + y \ \ (-2^{w-1} \le x + y < 2^{w-1})$$

$$x +_w^t y = x + y - 2^w \ \ (x + y \ge 2^{w-1}, \text{正溢出，} Positive\ Overflow)$$

溢出检测：当$x + y < -2^{w-1}$,而$x +_w^t y \ge 0$时，发生负溢出。

当$x + y > -2^{w-1}$,而$x +_w^t y < 0$时，发生正溢出。

实际上，补数加法也好判断的原理都是两个同号的数相加，得到的是一个不同于加数和被加数的符号，那肯定是发生了溢出，那到底是正溢出还是负溢出，还要看具体的情况，也就是加数和被加数的符号。

补码的非：

$$-_w^t x = TMin_w \ (x = TMin_w)$$
$$-_w^t x = -x \ (x > TMin_w)$$

为什么这样呢？因为补码是非对称的，$TMax_w$比$TMin_w$少 1，在正数或者负数下都可以找到自己的相反数，而$TMin_w$找不到正数的相反数，只能找自己然后通过溢出回归 0。

求补码非另外一种方法是每一位取反然后再加 1，即$-x = \sim x + 1$。

求补码非另另外一种方法是寻找从右起第一个数字为 1 的位 k，然后从 k+1 到最后一位都取反。

3) 无符号乘法

$$x \times_w^u y = (x \times y) mod\ 2^w$$

将一个无符号截断为 w 位等价于计算该值的模2^w。

4) 补码乘法

其实原理跟无符号乘法类似，最后把高位加权一下就可以了

$$x \times {}_w^t y = U2T_w((x \times y) mod\ 2^w)$$

5) 乘以常数

由于整数乘法（大概需要 10 个或者更多的时间周期）比移位和加法（只需要 1 个时间周期）的代价大得多，所以在执行整数乘法的时候都会化为左移和加法的组合，比如变量 x×14，由于 14 可以写成

$$14 = 2^3 + 2^2 + 2^1$$

所以

$$x \times 14 = x(2^3 + 2^2 + 2^1) = x \ll 3 + x \ll 2 + x \ll 1$$

这对于无符号和补码都一样，因为这都是位级的运算，该溢出的溢出就是了，最后化为十进制的时候在跟据数据类型来转化就是了。

6) 除以常数

整数除法在比整数乘法的代价更大，大概需要 30 个时间周期，无符号和补码的运算分别采用逻辑右移和算术右移，如果结果是一个小数，对于无符号来说就是就是向下舍入，而补码来讲正数是向下舍入，负数时也是是向下舍入。不需要管什么则向上舍入，向零舍入或或者是向上舍入，一律得到的是不超过结果的最大整数。

政治正确的摄入为向零舍入，所谓向零舍入摄入就是正负都向着靠近零的方向摄入。这时候负数的除法需要添加偏执量，做法为

$$x/2^k = (x + 1 \ll k - 1)/2^k$$

这样加上$1 \ll k - 1$等同于加上 k 个 1 的串 111…111，一定可以使得第 k+1 位加 1，如果第 k+1 位加不了 1，说明原来的就满足要求，低位由于最后都要被舍去所以加多少都不要紧，这样做的结果是最终就是负数向上舍入

7) 浮点数的计算

三、 程序的机器级表示

当我们用高级语言编程的时候，比如 Java 和 C 语言，机器屏蔽了许多程序的细节，即机器级的实现。通常情况下，现代的编译器编译产生的代码至少跟一个熟练的汇编语言程员编写的代码一样有效。最大的优点是，用高级语言写的程序可以在很多不同的机器上跑，而汇编代码则是与特定的机器有关。

对于严谨的程序员来讲，能够阅读和理解汇编代码仍然是一项很重要的内容。

下面的学习基于 x86-64（Intel 的处理器系列俗称 x86），即 64 位处理器，而 32 位用的比较少，而且现在的 64 位普遍兼容 32 位，所以学 64 位的也很容易上手 32 位。

3.1 基础知识

1) 在汇编代码中，带有%的表示寄存器，程序计数器 PC 用%rip 表示，给出下一条指令在内存中的地址。
2) 汇编中指令末尾带有的 q，其实是大小指示符，没有什么实际的用途，可以舍去不影响实际运行。
3) 所有以“.”开头的行都是指导汇编器和链接器工作的伪指令，我们通常可以忽略这姓行。
4) 机器代码知识简单地把代码看成是一个很大的按字节寻址的数组。C 语言中的聚合数据类型，例如数组和结构，在机器代码中用一组连续的字节表示。操作系统负责管理虚拟内存空间，将虚拟地址翻译成实际处理器中的物理地址。
5) 对于一些应用程序，程序员必须要用汇编语言来实现访问机器低级特性，在 C 语言中插入汇编代码有两种方式
   - 用汇编代码编写整个函数，在链接阶段把他们和 C 函数组合起来。把写好的 c 程序放在一个独立的汇编代码文件中，让汇编器和链接器把它和用 C 语言书写的代码合并起来；
   - 利用 GCC 的支持，直接在 C 程序中嵌入汇编代码。使用 GCC 的内联汇编（inline assembly）特性，用 asm 伪指令可以在 C 程序中包含简短的汇编代码，这种方法的好处是减少了与及其相关的代码量。
6) 数据格式：字。一开始 Intel 用的是 16 位的处理器，所以当时就定义 1 字(word)等于 16 位，后来出现了 32 位处理器，就称 32 位为“双字”(double words)，64 位称为“四字”(quad words)，所以指令的后缀 b 表示传送字节，后缀 w 表示传送字，后缀 l 表示传送双字，后缀 q 表示传送四字，但是奇怪的是后缀 l 也能用来传送 8 字节的 double 浮点类型，这是因为浮点数使用的是一组完全不同的指令和寄存器。
7) 不允许从一个内存地址直接传送到另外一个内存地址，也不允许将立即数传送到内存。

3.2 汇编代码格式

目前汇编代码的格式主要有两种，分别是 ATT 和 Intel 格式
ATT 格式，来自于 AT&T，AT&T 是运营贝尔实验室多年的公司，该格式也是 GCC，OBJDUMP 和其他一些我们使用工具的默认格式。
Intel 格式，来自于 Intel 的文档，主要应用于巨硬的工具。

主要的跟 ATT 格式的区别是
1) Intel 格式省略了指示大小的后缀，
2) Intel 格式省略了寄存器前面的%符号，用的是“QWORD PTR[rbx]”代替(%rbx)
3) Intel 格式在列出多个操作数指令的情况下，列出的顺序跟 ATT 格式相反。

当然了，你可以通过一些命令把 ATT 格式转换为 Intel 的格式
linux> gcc -Og -S -masm=intel mstore.c

3.3 寻址方式

1) 通用目的寄存器
用来储存整数数据和指针。
- 最初的 8086 中有 8 个 16 位的寄存器，%ax~%bp：
- %ax，%bx，%cx，%dx，%si，%di，%bp
- 拓展到 IA32 的时候，标号从%eax~%ebp
- 拓展到 x86-64 的时候，标号从%rax~%rbp，另外还增加了 8 个新的寄存器，标号分别从%r8~%r15
- 对于生成少于 8 字节的指令，寄存器剩下的字节的处理情况有以下两种：如果指令为 1 字节或者 2 字节，则剩下的字节保持不变，生成 4 字节的指令会把高位的四字节置零。

%rax(%eax) 返回值/用于做累加
%rbx(%ebx) 被调用者保存/用于做内存查找的基础地址
%rcx(%ecx) 第四个参数/用于计数
%rdx(%edx) 第三个参数/用于保存数据
%rsi(%esi) 第二个参数/用于保存源索引值
%rdi(%edi) 第一个参数/用于保存目标索引值
%rbp(%ebp) 被调用者保存
%rsp(%esp) 栈指针

当然了，还有其他有特殊用途的寄存器，比如单个位的条件码(condition code)寄存器

2) 操作数
- 定义：指示出执行一个操作中要使用的源数据值，以及放置结果的目的位置。
- 立即数（immediate）用来表示常数，格式$Imm，其中 Imm 是操作数值。
- 寄存器，用其中的低位 1 字节，2 字节，3 字节或者 8 字节表示，用符号 r_a 表示任意寄存器，用引用 R[r_a]表示它的值，这是将将寄存器集合看成是一个数组 R，用寄存器标识符作为索引。
- 内存引用，跟据计算出来的有效地址访问某个内存位置。$M_b[Addr]$表示对存储在内存中从 Addr 开始的 b 个字节值的引用，为了简便起见，我们通常省去标 b。其实就相当于指针

3) 寻址方式：

类型	格式	操作数
立即数寻址	$Imm	Imm
寄存器寻址	r_a	R[r_a]

绝对寻址	Imm	$M_b[Imm]$
间接寻址	(r_a)	$M_b[\mathrm{R}[\mathrm{ra}]]$
（基址+偏移量）寻址	$Imm(r_a)$	$M_b[Imm+\mathrm{R}[r_a]]$
变址寻址	(r_a, r_b)	$M_b[\mathrm{R}[r_a]+\mathrm{R}[r_b]]$
变址寻址	$Imm(r_a, r_b)$	$M_b[Imm+\mathrm{R}[r_a]+\mathrm{R}[r_b]]$
比例变址寻址	$(, r_a, \mathrm{s})$	$M_b[\mathrm{R}[r_a]\cdot s]$
比例变址寻址	$Imm(, r_a, \mathrm{s})$	$M_b[Imm+\mathrm{R}[r_a]\cdot s]$
比例变址寻址	(r_a, r_b, s)	$M_b[\mathrm{R}[r_a]+\mathrm{R}[r_b]\cdot s]$
比例变址寻址	$Imm(r_a, r_b, \mathrm{s})$	$M_b[Imm+\mathrm{R}[r_a]+\mathrm{R}[r_b]\cdot s]$

4) 数据传送指令
5) 最频繁使用的指令是将数据从一个位置复制到另外一个位置
6) 一条指令是由
   指令名称+源操作数+目的操作数
   组成。源操作数指定的值是一个立即数，存储在寄存器或者内存中，目的操作数指定一个位置，要么是一个寄存器或者是一个内存地址。X86-64 加了一条限制：传送指令的两个操作数不能都指向内存位置，即不能直接把一个值从一个内存地址直接复制到另外一个内存地址，要分开两步：先将源值加载到寄存器中，再将该寄存器值写入目的位置。

7) 间接引用指针，其实就是取某指针的值*p，在汇编中指针相当于把指针放进一个寄存器中%p，间接引用指针*p 就相当于在内存引用中使用这个寄存器（%p）。
8) 变量通常保存在寄存器中%bianliang，指针也相当于变量，所以指针变量也是%p，在内存中引用这个指针变量的时候，就用（%p）。一般来讲，访问寄存器比访问内存要快得多。

9) ret 是返回 return 的意思
10) movsbq：在复制值的时候把多余的高位设置成 1；
11) movzbq：在复制值的时候把多余的高位设置成 0；

3.4 基本操作

1) movq 的操作分为 4 个不同的指令：irmovq, rrmovq, mrmovq 和 rmmovq，分别显式地指明源和目的的格式，i 表示 immediate 立即数，register 寄存器，memory 内存。
2) 压入栈数据
   栈在内存中是从上往下存放的，地址自上往下逐渐减少，而栈又是后进先出的类型，栈顶位于下方，栈底位于顶部。当要压入栈数据的时候，他首先需要把栈顶向下移动 n 个字节，n 个字节等于你要压入的数据的容量大小。移动栈顶这一操作具体表现为：R[%rsp] <- R[%rsp]-8；接着，才把数据放在栈

顶：M（R[%rsp]）<- S。
即：R[%rsp] <- R[%rsp]-8
M(R[%rsp])<- S
等价于：
sub $8 ,%rsp
movq S (%rsp)

3) 弹出栈数据
D <- M(R[%rsp])
R[%rsp] <- R[%rsp]+8
等价于：
movq (%rsp) , D
addq $8 , %rsp
另外：movq 8(%rsp) , %rdx //相当于把栈中的第二个元素给复制给寄存器%rdx.

4) Call

作者：朱元
链接：https://www.zhihu.com/question/59201886/answer/162932167
来源：知乎
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

0. 先扯句题外话，直觉告诉我，这段代码是 O2 编译的？这教科书不错啊
- 2 个 pushq 之前，堆栈指针%rsp 除以 16 会余 8。
- 2 个 pushq 之后，堆栈指针%rsp 除以 16 还是余 8，因为%rsp 减少了 16。
- 函数 Q 只需要一个 long 参数，所以不需要在堆栈中准备参数， X64 SYSTEM V ABI （linux 默认）要求第一个 integer 参数使用%rdi 传递。
- 这时候直接调用函数 Q 违背了另外一条 X64 SYSTEM V ABI 规则：函数调用时%rsp 要能整除 16（如果使用了矢量类型参数则可能要求整除 32 或者更多）。
- 所以编译器生成的汇编代码会先让%rsp 自减 8。此时%rsp 能整除 16 了， 然后再 call。call 的过程是先把下一条指令地址放%rsp 的地址里，然后%rsp 自减 8，再跳转到 Q。
- 注意此时代码执行序已经进入 Q，此时堆栈指针%rsp 除以 16 会余 8。
- 1 的结论来自 4.1：这是一个递归的保证过程!! 从 main 函数起每一个函数都依赖在函数入口时堆栈指针%rsp 除以 16 会余 8，来实现无需访问%rsp 具体的值的情况下(只知道%rsp mod 16 == 8)，保证在 call 其它函数时%rsp 能整除 16(只需要做编译期模运算，这是 1，2，3 的运算过程)。
- 4.2 我强调从 main 函数起没有别的意思，只是暗示在 i386 时代，对

齐的规则没有遵守的这么严格，导致有的版本 32 位ＧＣＣ编译器喜欢在 main 函数入口 mov %esp, %eax and $16, %esp push %eax（此时 %esp mod 16 == 12， 又可以开心的做模运算啦）来保证后续函数调用堆栈总是按 16 字节对齐的, 保证自己编译出来的结果在各种 GLIBC 和 shell 环境下能够正确执行。

5) 算术和逻辑操作

加载有效地址 leaq(load effective address)

它的第一个操作数看上去是引用一个内存地址，但实际上并没有引用该地址，但该指令并不是从指定位置读入数据，而是将有效地址写入到目的操作数，目的操作数必须是一个寄存器。

leaq S, D //实际上是 D <- &S

leaq (%rdi, %rsi,4) , %rax //x+4y 储存在寄存器%rax 中。

leaq 不改变任何的内容，只传递地址，或进行地址的计算。

6) 一元操作

只有一个操作数，既是源又是目的，这个操作数既可以是一个寄存器，又可以是一个内存位置。

如 incq (%rsp) //自加 1

decq (%rsp) //自减 1

negq (%rsp) //取负

notq (%rsp) //取补

7) 二元操作

第二个操作符既是源又是目的

如 addq %rcx, (%rax) //(%rax)加上%rcx 再赋值给(%rax)

subq %rcx, (%rax) //(%rax)减去%rcx 再赋值给(%rax)

imulq $16, (%rax, %rdx, 8) //乘法

好奇怪啊，为什么没有除法，小白表示很慌

8) 移位操作

先给出移位量，再给出要移位的数。

SAL 左移

SHL 左移，跟上面的命令一样

SAR 算术右移

SHR 逻辑右移

9) 特殊的算术操作

有符号全乘法 imulq

他跟之前的双操作数有符号全乘法 imulq 不同的是,之前的双操作数有符号全乘法 imulq 不能保存溢出的位，但是单操作数下的有符号全乘法 imulq 可以保存溢出的位数，具体的原理是：首先这两个指令都要求一个参数必须在%rax 中，而另一个参数作为源操作数给出，然后乘积的高 64 位，即高 8 字节放在%rdx 中，低 64 位，即低 8 字节放在%rax 中。那么编译器是怎么分辨这两种乘法的呢？其实编译器是根据操作数的个数来区别这两种不同的乘法的。具体的做法如下：

//void store_uprod(uint128_t *dest, uint64_t x, uint64_t y)

//dest in %rdi, x in %rsi y in %rdx
// dest 是一个指针变量，所有的变量都存放在寄存器中，要引该变量的内存地址，需要在汇编中假如括号。
Movq %rsi %rax
Mulq %rdx
Movq %rax, (%rdi) //储存低 8 字节
Movq %rdx, 8(%rdi) //采用小端法储存高 8 字节
无符号全乘法 mulq $16, %rdx //同理

双操作数的除法是整除，无法储存余数，而但操作数的除法可以把余数也保留下来。原理是首先准备两个寄存器，%rax 和%rdx，%rax 负责储存被除数，%rdx 负责储存被除数%rax 的符号位，如果是无符号除法，则%rdx 被设置为全 0，如果是有符号除法，则储存%rax 的符号位。而设置%rdx 符号位的操作是通过无操作数指令 cqto 实现的。接着使用单操作数的 idivq 储存除数。结果的商 quotient 储存在寄存器%rax 中，而余数 remainder 储存在%rdx 中。具体的例子如下：

```
void remdiv(long x, long y, long *qp, long *rp){
    long q=x/y;
    long r=x%y;
    *qp=q;
    *rp=r;
}
//x in %rdi, y in %rsi, qp in %rdx, rp in %rcx
    movq %rdi, %rax
    movq %rdx, %r8
    cqto
    idivq %rsi
    movq %rax (%rdx)
    movq %rdx (%rcx)
```

有符号除法 idivq %rdx %rax
无符号除法 divq %rdx %rax

10) 为什么经常要在 ret 前面加上 rep;
原因在 http://bbs.csdn.net/topics/380086102 这里可以找到
摘自 http://board.flatassembler.net/topic.php?t=6264
其中提到 AMD64 的优化指南中提到，这么做是为了优化，在两种条件下无法发挥 cpu 的分支预测功能：
1> 一个分支，里面带有 near-return（opcode C3h），例如
label：
    ret
2> 一个判断条件紧跟着就是 near-return，例如
jle label2

ret

为了使得代码还是能够用得上 cpu 的分支预测（branch prediction），最简单的解决办法就是在 ret 前面插入 rep，这称之为 two-byte ret，使得性能可以提高。

我上面的问题应该是第二种情况，所以要用 rep ret。

3.5 条件码

前面所学的东西都是直线代码，所谓直线代码就是指令一条一条的顺序执行。

1) 条件码

除了整数寄存器，CPU 还维护着一组单个位的条件码(condition code)寄存器。

- CF：进位标志，最近的操作使得最高位产生了进位，可用来检查无符号操作的溢出；这里可以回想一下无符号加法溢出的判断，当$s = x + {}^{u}_{w}y$，而 s<x 的时候表示溢出。
- ZF：零标志，最近的操作得出的结果为 0；
- SF：符号标志。最近的操作得到的结果为负数，其实就是运算的结果为负数的时候；
- OF：溢出标志，最近的操作导致了一个补码的溢出——正溢出或者负溢出，这个可以联想补码加法的溢出条件，两个同号的补码相加，得到异号的结果，这肯定是溢出了。

几个规则：

leaq 不改变任何条件码，因为他是用来进行地址计算的；

对于逻辑操作，如 XOR，进位标志和溢出标志都会设置成 0；

对于移位操作，进位标志将设置为最后一个被移除的位，溢出标志设置成 0；

INC 和 DEC 指令会设置溢出标志和零标志，但不会改变进位标志；

所有的计算，逻辑和比较指令都会更新逻辑码；

其实可以分为两类指令，一类是既更新目的寄存器，又更新条件码；另一类只更新条件码，不更新目的寄存器。

一类指令：既更新目的寄存器，又更新条件码	二类指令：只更新条件码，不更新目的寄存器
ADD	
SUB	CMP S1,S2 //比较，基于 S2-S1，比较的时候是倒过来的，千万记得^< ^
MUL	
DIV	
XOR	
OR	

AND	TEST S1,S2 //测试，基于 S1&S2
INC	Test %rax, %rax 可以测试%rax 是否为 0，假如为 0，则 ZF=1，如果是非零，则 ZF=0，对于非零的情况，如果 SF=1，说明为负数，如果 SF=0，说明为正数。综上所述，这条语句可以检测%rax 的正负和零的情况。另外，CF 和 OF 会被设置成 0。
DEC	
NEG	
NOT	
SHL	
SHR	
SAL	
SAR	

我知道为什么乘法和除法有区分有符号和无符号，而加减法不区分了。因为汇编有设置单位的条件码，假如加减法溢出，无论是有符号溢出还是无符号溢出，最多只有一位溢出，这时用单位的条件码来储存溢出的位即可，而乘除法移除的时候可能就是 64 位的溢出，用单位的条件码根本储存不过来，所以乘除法需要另外设置无符号和补码的指令。

访问条件码

- 可以根据条件码的某种组合，将一个字节设置为 0，1 或者 2；
  SET 指令，跟据条件码的某种组合将一个字节设置成 0 或者 1。一条 SET 指令的目的操作数是低位单字节寄存器或者是一个字节的内存地址。
- 可以条件跳转到程序的某个其他的部分；
- 可以有条件的传送数据；

setl %al　　//小于<的条件，%al=SF^OF，这里是什么意思呢？负条件符异或溢出符，这里两种情况，假如没有发生溢出，就普通的比较的话，SF=1，OF=0；假如发生溢出了，符合小于的条件的只有负溢出，其中 S2 为负数，S1 为正数，S2<S1，S2-（S1）>0，即 OF=1，SF=0。
setg %al　　//大于>的条件，%al= ~(SF^OF)
seta %al　　//大于>的条件，%al= ~CF //无符号的只用判断是否进位即可
setb %al　　//小于<的条件，%al= CF
sete %al　　//等于=的条件，%al= ZF
sete %al　　//等于=的条件，%al= ~ZF

很多情况下，机器代码对于无符号和补码的使用同样的操作，这是因为他们有相同的位级行为，而有些情况却需要区别开来，比如右移，除法和乘法指令，以及不同的条件码组合；

3.6 跳转

1) 无条件跳转 jmp
   在产生目标代码文件的时候，汇编器会确定所有带标号指令的地址，并将跳

转目标(目的指令的地址)编码为跳转指令的一部分。

- 直接跳转，直接在 jmp 后面给出一个标号

```
jmp .L1
```

- 间接跳转，在 jmp 后面给出一个星号*，然后在从星号后面的寄存器或内存中读取跳转的地址，如

```
jmp *%rax
```

或者

```
jmp *(%rax)        //以%rax 为读地址，从内存中读取跳转目标
```

2) 有条件跳转指令

je，jne，js，js，jl，jle，jge，jg，jbe，jb，jae，ja

或者使用他们的同义名，在 j 后加 n，表示否定

jz，jnz，　　，jnge，jng，jnl，jnle，jna，jnae，jnb，jnbe

3) 编码例子：

```
    movq $0,%rax
    jmp .L1
.L1:
    popq %rdx
```

4) 跳转目标的编码方式：

汇编器以及后来的链接器，会产生跳转目标的适当编码

- PC 相对寻址(PC-relative)，这个最常用将目标地址与紧跟着跳转指令后面的那条指令的地址之间做差作为编码，这些地址偏移量可以编码为 1，2 或 4 字节；
- “绝对”地址寻址，用 4 字节直接制定目标；

汇编器和链接器会选择适当的跳转目的编码。

5) 非常奇怪的是，在 C 语言编写过程中，使用 goto 指令是普遍嗤之以鼻的做法，我么一般会用 if-else 等语句替代，但是当把这种无 goto 指令的带条件的 C 语言代码翻译成汇编语言的时候，确使用了 goto 语句，通用模板如下：

```
if(test-expr){
    then-statement
}else{
    else-statement
}
```

翻译成汇编语言后，使用有条件跳转和无条件跳转分支，汇编器为 then-statement 和 else-statement 各自生成代码。

```
    t= test-expr
    je (~t)
        goto false
    then-statement
    goto done;
false:
    else-statement
done:
```

6) 跳转的方式
   - 用条件控制来实现条件分支，也即好像我们上面写的例子一样，虽然这种机制比较简单通用，但是在现代的处理器上，他可能会比较低效
   - 用条件传送来实现条件分支，现把两种情况的结果都执行，最后跟据比较的结果再决定返回哪一个结果。详细讲就是当传送条件满足的时候，指令把源值 S 复制到目的 R，类似于三目运算符 A？S：R

   那问题的关键是为什么采用数据的条件跳转的执行效率会比按控制的条件跳转高呢？

   这里面涉及到现代处理器使用流水线（pipelining）获得高性能的原理。每一条指令的执行其实都分成好几个不同的过程——取指令，分析指令，执行指令，具体可以看一下《计算机组成原理》这本中文的教材， 而在 CSAPP 这本书中，分为以下过程：从内存中取指令，确定指令类型，从内存读数据，执行算术运算，向内存写数据，以及更新程序计数器。流水线在执行每一条指令的时候并不是严格让前一条指令完全执行完毕后再执行下一条指令，而是稍微会有些重叠，比如上一条指令的算术运算过程会跟下一条指令的取指过程重合。这种重叠连续指令的方法能获得高性能。一般来讲，当机器遇到条件指令的时候，当分支条件求值完成后，才能决定要走哪一条方向。而现代的处理器很“聪明”，他会预测分支的结果，然后执行接下来的操作，如果预测成功的话，就可以进行重叠处理。如果预测失败的话，就需要推倒重来，这样子会花费更多的时间，招致很严重的“惩罚”，浪费 15~30 个时间周期。现代处理器有精密的分支预测逻辑试图达到 90%以上的准确率，让流水线上充满着指令。

   那如何确定分支预测错误的惩罚呢？我们运用一般的概率论知识便可求解期望值。

$$T_{avg}(p) = (1-p)T_{success} + p\left(T_{fail} + T_{success}\right) = T_{success} + pT_{fail}$$

$$T_{fail} = \frac{T_{avg}(p) - T_{success}}{p}$$

7) 条件传送指令，类似于三目运算符 A？B：C，当传送条件满足的时候，指令把源值 S 复制到目的 R；
   - comve S,R
   - comvne S,R

   - comvs S,R
   - comvns S,R

   - comvg S,R
   - comvge S,R
   - comvl S,R
   - comvle S,R

- comva S,R
- comvae S,R
- comvb S,R
- comvbe S,R

当然了，使用条件传送实现条件跳转也并别总是提高代码的效率，因为条件传送总会浪费一半的时间，这需要权衡浪费掉的时间与由于分支预测错误造成的惩罚，哪个的代价比较大。很遗憾的是，编译器还没有聪明到可以做出一个可靠的选择。

CSAPP 对 GCC 进行过实验，只有当表达式比较简单容易计算的时候，比如说都只有一条加法指令的时候，才会使用条件传送。其他的即使分支预测造成的惩罚的开销特别大，编译器还是会选择使用条件控制转移。

3.7 循环

非常遗憾的是，汇编中并没有提供相应的循环指令，只能使用条件测试和跳转组合实现循环的功能。

1) do-while 循环

```
do
    body-statement;
    while (test-expr);
```

等效的汇编代码如下：

```
loop:
    body-statement
    t= test-expr;
    if(t)
        goto loop;
```

2) while 循环
3) for 循环

3.8 switch 多重分支

可以根据一个整数索引进行多重分支(multiway branching)，而且通过跳转表(jump table)这种数据结构使得实现更加高效。

在 C 语言中，我们知道&表示指向数据的指针，GCC 的作者们创建了一个新的运算符&&，表示指向代码位置的指针。

```
index=100;
static void *jt[7]={&&loc_A, &&loc_def, &&loc_B, &&loc_C, &&loc_D,
&&loc_def, &&loc_D}
if(index>6){
  goto loc_def;
}else{
```

```
  goto *jt[index];
 }

 loc_A:
  //
 loc_B:
  //
 loc_C:
  //
 loc_D:
  //
 loc_def:
  //
 done:
```

对应的汇编代码如下：

```
    .section .rodata
    .align 8                //Align address to multiple of 8
.L4:
    .quad .L3               //Case 100：loc_A
    .quad .L8               //Case 101：loc_def
    .quad .L5               //Case 102：loc_B
    .quad .L6               //Case 103：loc_C
    .quad .L7               //Case 104：loc_D
    .quad .L8               //Case 105：loc_def
    .quad .L7               //Case 106：loc_D
```

其中 rodata 表示 read-only data 只读数据，.section .rodata 这一行表示声明一个跳转表，.L4 表示代码入口，.quad 表示四字八字节。

一个完整的汇编代码：

```
switch：
    comp $7, %rdi
    ja .L2
    jmp *.L4(,%rdi,8)    //如果是 jmp *.L4(,%rdi,8) 那么就先找到 .L4 然后往后找 8 个字节（或 8 的倍数），因为每一个分支位置都是 quad 八字节。
    .section .rodata
.L7
    //
.L3
    //
……
```

3.9 过程

过程是软件中一种很重要的抽象，他提供了一种封装代码的方式，用一组指定的参数和一个可选的返回值实现某种功能，不同的编程语言中，他都有不同的形式：

- 函数(Function)
- 方法(Method)
- 子例程(Subroutine)
- 处理函数(Handler)

1) 机制
   - 传递控制
     进入过程 Q 的时候，程序计数器必须设置为 Q 的起始地址，假设过程 P 调用过程 Q，Q 执行后返回 P，那么还要把程序计数器设置为 P 中调用 Q 后面的那条指令的地址 A。调用 Q 的时候用 call Q 指令把地址压入栈中，并将 PC 设置为 Q 的起始地址；,,,最后利用 ret 指令从栈中弹出地址 A，并把 PC 设置成 A。
   - 传递数据
     值传递
   - 分配和释放内存
     局部变量的创建和释放
2) 运行时栈
   关键特性：栈数据结构提供的后进先出的内存管理原则。
   当 P 调用 Q 的时候，控制和数据信息添加到栈尾
   向低地址方向增长，栈指针%rsp 指向栈顶元素
   以 pushq 和 popq 指令将数据存入栈中或是从栈中取出，将栈指针减少一个适当的量可以为没有初始值的数据在栈中分配空间，也可以通过增加栈指针来释放空间。
   栈帧(stack fram)：当需要的储存空间超过寄存器能够存放的大小的时候，就会在栈中分配空间，这部分称为过程中栈帧(stack fram)，超过 6 个以外的参数的传递称为栈传递，他们位于“参数构造区”，而且通过栈传递的数据大小都是向 8 的倍数对齐；
   许多过程有 6 个或者更少的参数，那么所有的参数都可以通过寄存器传递。实际上，很多的函数甚至根本不需要栈帧，CSAPP 这本书自称“到目前为止我们仔细审视过的所有函数都不需要栈帧”7。
   寄存器作为储存参数的顺序：%di，%si，%dx，%cx，%r8，%r9；
3) 栈上的局部储存
   局部数据必须放在内存中的常见情况如下：
   - 寄存器不足够存放所有的本地数据；
   - 对一个局部变量使用地址运算符&，因此必须为他产生一个地址；
   - 某些局部变量是数组或者是结构，因此必须能够通过数组或者结构

引用被访问到；

4) 寄存器中的局部存储空间

寄存器组是唯一被所有过程共享的资源。

被调用者保存寄存器

按照惯例，寄存器%rbx，%rbp 和%r12~%15 被划分为“被调用者保存寄存器”，所谓被调用者保存，就是被调用者有责任去保证这些寄存器的值不被改变。

被调用者 Q 该如何保存这些值呢？有两种办法：

- 根本不去改变他们；
- 把原始值压入栈中，改变寄存器的值，然后在返回的时候从栈中弹出旧值；

其实为什么要使用被调用者保存寄存器呢？或者这么问：在什么场景下需要用到被调用者保存寄存器？我想了很久，也看了几遍 CSAPP 相关的内容，得出以下结论：

- 被调用函数父子之间的冲突：

  在被调用的函数 Q 中存在另外的调用函数 R，当 R 中的第一个参数不是 Q 中的第一个参数，而是另外的参数的时候，我们知道被调用的函数的参数顺序是对应相应的寄存器的，比如第一个参数约定使用%rdi，第二个约定使用%rsi。那么假如 R 的参数是 Q 中的第二个参数时，那么 R 需要用到%rdi 来储存 Q 的%rsi，但是这时%rdi 已经储存了 Q 的第一个参数了，所以为了保证不改变 Q 的第一个参数，需要使用另外的寄存器来储存 Q 的第一个参数，即在%rdi 中的值，好让寄存器%rdi 能被 R 使用。
- 被调用函数兄弟之间的冲突：

  被调用函数内有多个带返回值的子被调用函数，此时需要用另外的寄存器保存前面的子被调用函数的返回值%rax。因为后面的子被调用函数返回值也需要用到%rax 寄存器。

调用者保存寄存器

除了被调用者保存寄存器和%rsp，剩下的都被归类为调用者保存寄存器，意味着所有函数都可以改变他们的值，所以在调用前的调用者有责任去保存这些寄存器的值。

5) 递归的过程

栈规则提供一种机制，每次函数都有自己私有的状态信息（保存的返回位置和被调用者保护寄存器的值）存储空间，如果需要，他还可以提供局部变量额存储。

3.10数组的分配和访问

1) 原则：

T A[N]

这样的声明有两个效果：首先在内存中分配一个 L • N 个字节的连续空间区域，其中 L 是 T 类型的大小，单位是字节；其次，标识符 A 指向数组开头的首地址。

2) 访问

movl (%rdx,%rcx,4), %eax

表示读 A[4]的值。

假设 A 的类型是 int，整数索引 i 放在%rcx 中

A	movq %rdx, %rax
A[0]	movl (%rdx), %rax
A[i]	movl (%rdx,%rcx,4), %eax
& A[2]	leaq 8%rdx, %rax

3) 高维数组

行优先索引策略

int A[5][3]

等价于下面的声明

typedef int row3_t[3];

row3_t A[5]

A in %rdi, i in %rsi, and j in %rdx，在 A[M][N]中寻找 A[i][j]，利用吕氏公式

$$\%\text{rax} = X_A + (\text{M} \times \text{i} + \text{j}) \times \text{sizeof(A)}$$

很多问题都会引刃而解。

```
leaq (%rsi, %rsi, 2), %rax      //前面所有行的元素个数总和
leaq (%rsi, %rax, 4), %rax      //为什么要乘以 4，因为数据类型是 int，4 字节
movq (%rax, %rdx, 4), %eax
```

4) 定长和变长数组

5) 异质的数据结构：

结构 struct

联合 union

6) 对齐原则

定义：任何 K 字节的基本对象的地址必须是 K 的倍数（K=2,4 或者 8）

这种对齐限制简化了形成处理器和内存系统之间的硬件接口。

- 如果空间有限，则优先保证第一个元素满足对齐原则。
- 结构体中的对齐原则可能会使元素间产生空隙（多余的空字节）
- 对于结构体的数组，每一个最后需要空出多个字节，因为需要让后一个数组元素的第一个元素满足对齐原则。

虽然没有对齐原则并不会影响程序的行为，但是某些型号的 Intel 和 AMD 处理器对于有些实现多媒体操作的 SSE 指令，就无法正确执行，因为这些指令是对 16 字节的数据块机型操作的，其要求内存地址必须是 16 的倍数，任何试图以不满足对其要求的地址来访问内存都会导致异常，默认的行为时程序终止。为了让所有有可能被 SSE 指令访问到的内存访问时没有问题，需要：

- 任何内存分配函数生成的块的起始地址都必须是 16 的倍数；
- 大多数函数的栈帧的边界都必须是 16 字节的倍数。

7) 缓冲区溢出（buffer overflow）

跟据 Spafford 的说法

- 蠕虫（worm）：可以自己运行，并且能够将自己的等效副本传播到其他机器

- 病毒（virus）：能将自己添加到包括操作系统在内的其他程序中，但他自己没办法独立运行。
- 攻击代码：如果要想在内存插入攻击代码，则还需要插入该代码的地址，因为攻击代码的地址也是攻击代码的一部分。过去的内存分配方式是比较固定的所以攻击代码的地址比较容易预测，所以许多系统都容易受到同一种病毒的攻击，这种现象称为“安全单一化”(security monoculture)
- 栈随机化：为了避免“安全单一化”，最新的 GCC 版本建立了一种新的机制——栈随机化，它的思想是让栈的位置在每次运行的时候都不一样。实现的方式是在每次运行的时候，程序开头插入一段 0~n 字节之间的随机大小的空间。

8) 栈保护的三种主要机制
   - 地址空间布局随机化（Address-Space Layout Randomization）
     在 linux 系统中，栈随机化已经变成一种标准的做法，同时栈随机化也是地址空间布局随机化技术（ASLR）中的一种。采用 ASLR，会使得每次运行程序的不同部分，包括程序代码，库代码，栈，全局变量和堆数据，都会被加载到内存中的不同位置。
   - 栈破坏检测
     GCC 最近引入了一种新的机制——栈保护者(stack protector)，用来检测缓冲区越界，其思想是在栈帧中任何局部缓冲区和栈状态间插入一个特殊的金丝雀值(Canary，因为历史上金丝雀用来检测煤矿中的有毒气体)，在程序每次运行的时候随机产生。在恢复寄存器或者函数返回前，程序都会检测金丝雀值是否被改变，如果被改变则会使程序异常终止。
   - 限制可执行代码区域
     只有编译器产生的代码才能可执行，其他的区域只能读和写。以前的编译器把读和可执行访问放在同一个 1 位标志位。也就是说，如果设置为可读，那同时也是可执行的。以前很多机制都是基于这种特性建立的。如今把可读和可执行分开，那些机制通常会带来严重的性能损失。

四、 处理器体系结构

现代微处理器算是人类创造出的最复杂的系统之一，本章将要介绍处理器硬件的设计。与一个时刻只执行一个指令相比，通过同时处理多条指令的不同部分，处理器可以获得更高的性能。后面还会介绍一种硬件系统控制部分的简单语言，HCL(Hardware Control language).我么将模仿 x86-64 涉及我们自己的 Y86-64

4.1 Y86-64 指令集体体系结构

这个体系结构包括：定义各种状态单元，指令集和他们的编码，一组编程规范和异常事件处理。

1) 程序员可见的状态
   - 首先定义什么是程序员可见的状态：每条指令都会读取或者修改处理器

状态的某些部分。这里的程序员既可以是用汇编编写代码写程序的人，也可以是产生机器级代码的编译器；

- 设计有 15 个寄存器：
  %rdi, %rsi, %rdx, %rcx, %rbx, %rax, %rsp, %rbp, %r8~%r14，%15 由于比较复杂所以省略了对他的设计。每个寄存器都可以储存 64 位，%rsp 被入栈，出栈，调用和返回指令作为栈指针；
  这 15 个寄存器代表每个都有一个对应的范围在 0~0xE 之间的寄存器标识符（register ID）程序寄存器存在 CPU 一个寄存器文件中，当不应该访问寄存器的时候 ID 值用 0xF 表示。

表 4-1 Y86-64 寄存器标识符

数字	寄存器名字	数字	寄存器名字
0	%rax	8	%r8
1	%rcx	9	%r9
2	%rdx	10	%r10
3	%rbx	11	%r11
4	%rsp	12	%r12
5	%rbp	13	%r13
6	%rsi	14	%r14
7	%rdi	15	无寄存器

- 有 3 个一位的条件码：SF，ZF，OF，保存着最近的算术和逻辑指令造成影响的有关信息；
- 程序计数器 PC 存放当前正在执行指令的地址；

2) Y86-64 指令
   - Y86-64 指令基本上是 x86-64 指令的一个子集，但是内容会少很多，只包括 8 字节的整数操作，寻址方式比较少，由于只有 8 字节的操作，所以称之为“字（word）”不会有歧义。汇编代码的风格接近于 ATT 格式。
   - 不允许从一个内存地址直接传送到另外一个内存地址，也不允许将立即数传送到内存。
   - 每条指令还有他自身的字节级编码，每条指令需要 1~10 个字节不等，每条指令的第一个字节表明其指令的类型，高四位代表代码(code)部分，低四位代表功能(function)功能部分
   - 比如指令 rmmovq %rsp 0x123456789abcd(%rdx)，rmmovq 编码为 40，%rsp 寄存器标识符为 4，%rdx 标识符为 2，那么确定前面 16 位为 4042，接着就是 8 字节常数 0x123456789abcd，前面补 0 得：00 01 23 45 67 89 ab cd，由于习惯使用小端法编码，所以常数写成 cd ab 89 67 45 23 01 00，这样合起来就是 4042cdab896745230100。
   - 其实我不是很明白为什么有些指令需要加上 8 字节或者 4 字节的常数。在 4.3 节中终于搞清楚那八字节的常数到底有什么用了，原来用来跟里面的值相加得到内存的地址。
   - 指令集的一个重要性之就是字节编码必须有唯一的解释，这样的性质可以保证处理器可以无二义性地执行目标代码程序。即使将代码嵌入在程

序的其他字节当中，只要从序列中的第一个字节开始处理，我们仍然可以很容易地确定指令序列。反过来说，假如你不知道代码的起始位置，你就很难确定如何把代码划分为单独的指令。

表 4-2 Y86-64 指令集

字节		0	1	2	3	4	5	6	7	8	9
halt		0 0									
nop		1 0									
rrmovq rA, rB		2 0	rA rB								
irmovq V, rB		3 0	F rB	V							
rmmovq rA, D(rB)		4 0	rA rB	D							
mrmovq D(rB), rA		5 0	rA rB	D							
OPq rA, rB		6 fn	rA rB								
jXX Dest		7 fn	Dest								
cmovXX rA, rB		2 fn	rA rB								
call Dest		8 0	Dest								
ret		9 0									
pushq rA		A 0	rA F								
popq rA, rB		B 0	rA F								

表 4-3 OPq 整数操作指令集

字节		0	1	2	3	4	5	6	7	8	9
addq rA, rB		6 0	rA rB								
subq rA, rB		6 1	rA rB								
andq rA, rB		6 2	rA rB								
xorq rA, rB		6 3	rA rB								

表 4-4 jXX 指令集

字节		0	1	2	3	4	5	6	7	8	9
jmp Dest		7 0	Dest								
jle Dest		7 1	Dest								
jl Dest		7 2	Dest								
je Dest		7 3	Dest								
jne Dest		7 4	Dest								
jge Dest		7 5	Dest								
jg Dest		7 6	Dest								

表 4-5 cmocXX 传送指令集

字节		0	1	2	3	4	5	6	7	8	9
rrcmovq rA, rB		2 0	rA rB								
cmovle rA, rB		2 1	rA rB								
cmovl rA, rB		2 2	rA rB								

cmove rA, rB		2 3	rA rB								
cmovne rA, rB		2 4	rA rB								
cmovge rA, rB		2 5	rA rB								
cmovg rA, rB		2 6	rA rB								

- Y86-64 指令集的运算操作中无法直接使用立即数，只能先通过 irmovq V, rB 指令把立即数放到寄存器中，然后再在运算操作中使用该寄存器；
- Y86-64 指令集的运算操作中无法直接使用内存数据和寄存器数据相加减，只能通过 mrmovq D(rB), rA 操作把内存中的数据暂存到寄存器中，然后再在运算操作中使用该寄存器；
- RISC 与 CISC 之争，一开始是 CISC（读作：sisk）复杂指令集计算机，后来是 RISC（读作：risk）精简指令集计算机，再后来的指令集是取以上两者之精华。

3) Y86-64 异常
   - 对 Y86-64 来讲，程序员可见的状态包括状态码 Stat，它描述的是程序执行的总体状态，任何除开 AOK 以外的代码都会使处理器停止。

表 4-6 Y86-64 状态码

值	名字	含义
1	AOK	正常操作
2	HLT	遇到执行 halt 指令
3	ADR	遇到非法地址
4	INS	遇到非法指令

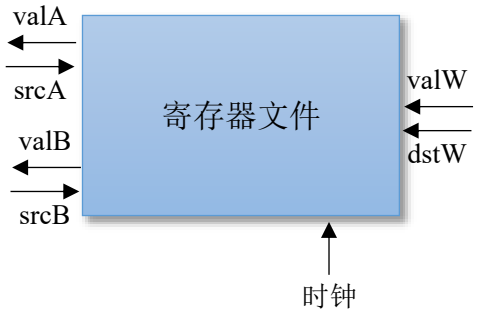
   通常来讲，在更加完整的设计中，处理器会调用一个异常处理程序(exception handler)
   - 以“.”开头的词是汇编器伪指令(assembler directives)，它们告诉编译器调整地址，以便在那里产生代码或者插入一些数据。比如伪指令“.pos 0”，告诉编译器从地址为 0 的地方开始产生代码。“.align 0”,告诉编译器起始地址及在 8 字节边界处对齐。

4) YAS 专门针对 Y86-64 的汇编器，作用是把汇编代码翻译成机器代码
   汇编文件中有代码和数据的行上，目标代码包含有一个地址，后面是 1~10 个字节的指令编码。
5) YIS 指令集模拟器，目的是模拟 Y86-64 机器代码程序的执行，而不用试图去模拟具体的处理器的实现的行为。模拟输出的第一行总结了执行以及 PC 和程序状态的结果值，模拟器只打印在模拟器过程中被改变的寄存器和内存中的字，左边是原始值（这里都是 0），右边是最终的值。
6) 一些细节问题：对于 pushq 指令，他会先把栈指针寄存器的值减去 8，然后再把 pushq 后面寄存器的值写入内存当中。因此，当执行 pushq %rsp 的时候，处理器的行为是不确定的，因为要入栈的寄存器会被同一条指令修改，通常有两种约定：1）是压入%rsp 的原始值；2）压入减去 8 的%rsp 的值。具体是哪一种方式，取决于不同的处理器型号，其实我也不知道。

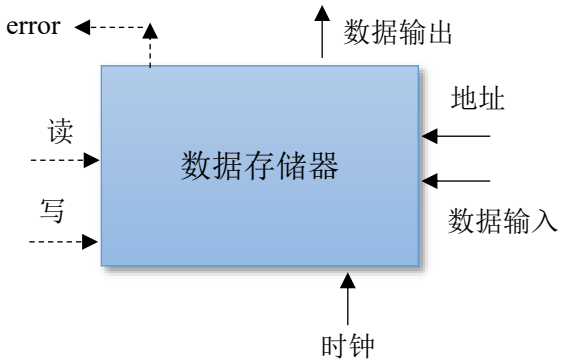
4.2 逻辑设计和硬件控制语言 HCL

1) 大多数现代电路都是用信号线上的高电压和低电压来表示不同的位值，逻辑 1 用 1.0V 左右的高电压表示，0 用 0.0V 左右的低电压表示。
2) 一个数字系统需要三个主要的组成部分：
   - 计算对位进行操作的函数的组合逻辑
   - 存储位的存储器单元
   - 控制存储器单元更新的时钟信号
3) HDL 其实是一种文本表示，与编程语言类似，但是它描述的是硬件的结构，而不是程序行为，最常用的编程语言是 Verlilog，类似于 C 语言，还有一种是 VHDL，类似于 Ada。
4) 逻辑门
   - 逻辑门是数字电路的基本计算单元，它用&&，||和！分别代表 C 语言中的位运算符&，|和~。
   - 逻辑门总是活动的(active)，一旦一哥们发生了变化，在很短的时间内，输出就会发生变化。
5) 组合电路
   将多个逻辑门组合在一起，就能构建计算块(computational block)，
6) 多路复用器(multiplexor，通常称为”MUX”)
   跟据输入控制信号的值，从一组不同的数据信号中选出一个
   (S&&a) || (!S&&b) 最后输出 a 还是输出 b 全靠 S 的值决定
7) 字级的组合电路和 HCL 整数表达式
   word Out4=[
   ```
   !s1 && !s0  :   A;  #   00
   !s1         :   B;  #   01
   !s0         :   C;  #   10
   1           :   D;  #   11
   ```
   ]
8) 算术/逻辑单元(ALU)
   这是一种很重要的组合电路
9) 储存器和时钟
   从本质上讲，组合电路是不储存任何信息，相反，他只是简单地相应输入信号。所以，为了产生时序电路(sequential circuit)，必须要有储存信息的容器和时钟信号。时钟信号用来控制容器何时加载信息。而储存的容器我们称之为寄存器设备。
   寄存器是设备一般有以下两种：
   时钟寄存器（简称寄存器），存储单个位或者字，用时钟信号控制寄存器输入那个值
   随机访问寄存器（简称内存），存储多个字，用地址来选择该读或者改写哪个字。
   硬件的寄存器和机器级编程的寄存器的含义不同，前者是将它的输入和输出线连到其他电路，而后者指的是在 IA32 或者 x86-64CPU 中为数不多可寻址

的字，这里的地址指的是寄存器标识符 ID，这些字通常存储在寄存器文件中。为了避免歧义，我们习惯把前者称之为“硬件寄存器”后者称之为“程序寄存器”



src 表示寄存器中目标程序寄存器的标识符 ID，dstW 表示寄存器中目的程序寄存器的标识符 ID，



4.3 Y86-64 的顺序实现

SEQ 处理器，即顺序 sequence 实现处理器

1) 过程
   - 取指(fetch)：
     程序计数器 PC 里面包含这在内存中指令的地址，通过 PC 抽取出指令指示符字节的两个四位部分，分别称为 icode 指令代码和 ifun 指令功能。当然了，取指不仅会读两个四位，还会读后面的的该指令的所有字节。而程序计数器 PC 下一条指令的地址 valP 等于 PC 的值加上已取出指令的长度。比如你读取该指令的全部 10 字节，那么 valP=valP+10；
   - 译码(decode)
     从寄存器文件中读取最多两个操作数，得到值 valA 和/或 valB。说白了就是读取寄存器里面的值。通常它会读指明的寄存器，但是有时候也会读%rsp 栈指针寄存器。
   - 执行(execute)
     算术/逻辑单元 ALU，就是执行指令的操作，然后把结果保存为 valE。并且 Set CC，即更新条件码。
   - 访存 memory

将数据写入内存，或者从内存中读入数据。读出的值称为 valM。这个主要是用到指针访问内存的时候用到

- 写回 write back
  最多可以写两个结果到寄存器文件，其实就是把 valE 更新到目的寄存器中。
- 更新 PC(PC update)
  将 PC 设置成下一条指令的地址。即 PC=valP

表 4-7 逻辑操作和赋值操作指令过程示例

阶段	OPq rA, rB	rmmovq rA, D(rB)	mrmovq D(rB), rA
取指	icode, ifun ← M_1[PC] rA, rB ← M_2[PC] valP←PC+2	icode, ifun ← M_1[PC] rA, rB ← M_2[PC] valC ← M_8[PC] valP←PC+10	icode, ifun ← M_1[PC] rA, rB ← M_2[PC] valC ← M_8[PC] valP←PC+10
译码	valA←R[rA] valB←R[rB]	valA←R[rA] valB←R[rB]	valB←R[rB]
执行	vaE ← valA OPq valB Set CC	valE ← R[rB]+ valC	valE ← R[rB]+ valC
访存		M_8[valE] ← valA	valM← M_8[valE]
写回	R[rB] ← vaE		R[rA]← valM
更新 PC	PC ← valP	PC ← valP	PC ← valP

注意：符号 M_n[x]表示访问地址 x 处的 n 个字节。终于搞清楚那八字节的常数到底有什么用了，原来用来跟里面的值相加得到内存的地址。

表 4-7 压入和弹出栈的指令过程示例

阶段	pushq rA	popq rA
取指	icode, ifun ← M_1[PC] rA, rB ← M_2[PC] valP←PC+2	icode, ifun ← M_1[PC] rA, rB ← M_2[PC] valP←PC+2
译码	valA←R[rA] valB←R[%rsp]	valA←R[%rsp] valB←R[%rsp]
执行	valE ← valB +(- 8)	valE ← valB + 8
访存	M_8[valE] ← valA	valM ← M_8[valA]
写回	R[%rsp] ← valE	R[rA] ← valM R[%rsp] ← valE
更新 PC	PC ← valP	PC ← valP

注意：这里 pushq 和 popq 类似，但 popq 需要读取两次栈指针，好像显得有些多余，但是 CSAPP 中给出的解释是：“会使后面的流程跟其他的指令（笔者：这里指的是 pushq 指令）更相似，增强设计的整体一体性”

表 4-8 跳转操作指令过程示例

阶段	jXX Dest	call Dest	ret
取指	icode, ifun ← M_1[PC] valC ← M_8[PC+1] valP←PC+9	icode, ifun ← M_1[PC] valC ← M_8[PC+1] valP←PC+9	icode, ifun ← M_1[PC] valP←PC+1
译码		valB←R[%rsp]	valA←R[%rsp] valB←R[%rsp]
执行	Cnd ← Cond(CC, ifun)	valE ← valB+(-8)	valE ← valB+8
访存		M_8[valE] ← valP	valM ← M_8[valA]
写回	R[rB] ← vaE	R[%rsp] ← valE	R[%rsp] ← valE
更新 PC	PC ← Cnd? valC:valP	PC ← valC	PC ← valM

2) SEQ 唯一的问题，就是太慢了，时钟必须非常慢，以使信号能在一个周期内传播所有的阶段。这种实现方式不能充分利用硬件单元。因为每个单元只在整个式中周期的一部分时间内才被使用，

4.4 流水线的通用原理

流水线化的重要特性是提高了系统的吞吐量 thoughput，不过他会稍微增加延迟 latency。

1) 计算流水线

在现代逻辑设计中，电路延迟以微微秒或者皮秒 picosecond——10^{-12} 计算，在未实现流水线化的时候，在开始下一条指令之前必须完成前一个。指令与指令的执行之间没有重叠。

$$\text{吞吐量} = \frac{1\ 条指令}{(20+300)ps} \times \frac{1000ps}{1ns} = \frac{1000}{(20+300)} \times \frac{1000 \times 10^6 条指令}{s}$$

$$= 3.12 \times 1000M条指令/s \approx 3.12GIPS$$

$$1ns = 10^{-9}s$$

$GIPS$——每秒千兆条指令，也即每秒十亿条指令

$$\text{延迟} = \frac{1}{\text{吞吐量}}$$

当我们使用流水线处理的时候，每条指令的完成的时间可能会有所增加，假设增加到 360ps,把每一条指令的分成三个过程，各占 120ps，然后三条指令分别延迟上一条指令 120ps 执行，总时间 600ps 内完成 3 条指令。

$$\text{吞吐量} = \frac{3\ 条指令}{(360)ps} \times \frac{1000ps}{1ns} = 8.33GIPS$$

这样吞吐量比原来增加了$\frac{8.33}{3.12} = 2.67$ *倍*，代价是增加了一些硬件，以及延迟的少量增加$\frac{360}{320} = 1.12$ *倍*，延迟增大是由于增加的流水线寄存器的时间开销。

另外，时钟过慢不会影响流水线的行为，但是当时钟过快的时候，就会有灾难性的后果，值可能来不及通过组合逻辑，寄存器得到的还不是合法的值。

2) 不一致的划分
当指令的阶段划分不一致的时候，运行时钟的速率受最慢阶段的延迟限制。对于硬件设计师来讲，将系统设计为一组具有相同延迟的阶段是一个非常严峻的挑战。通常对于处理器中的某些硬件单元，比如 ALU 和内存，它们是不能划分成多个延迟较小的单元的，这使得创建这样一组平衡的阶段非常的困难。
3) 流水线过深
当然了我们可以把一条指令划分成很多个阶段，原本是可以很大的提升性能的，但是，流水线过深会使得延迟变大，所以这是一个具有提升上限的过程。为了提高时钟频率，现代处理器采用了很深的流水线，15 或者更多的阶段。
4) 带反馈的流水线系统
有时候一条指令的输入是上一条指令的输出，这样设计流水线系统就会非常的麻烦

4.5 SQE+重新安排计算阶段

调整指令执行的步骤，把更新 PC 放到 6 个步骤的开始，称此时的 SQE 为 SQE+
后面将流水线冒险和惩罚的内容实在太麻烦了，以后很少能用到，先战术性放弃，以后有时间再看。

五、 优化程序性能

编写高效的程序要做到一下几点：一是必须选择一组适当的算法和数据结构；二是编写编译器能够有效优化以转换成高效可执行代码的源代码；
但是程序员必须在实现和维护程序的简单性与它的运行速度之间做出权衡。

5.1 优化编译器的能力和局限性

1) 一般来讲，-Og 是最低的优化级别
linux> gcc -Og mstore.c
当然了，还有-O1，-O2，-O3 的级别，其中大多数使用 GCC 编译器来讲，-O2 已经称为被接受的标准，但是有时候使用-O1 的版本会比最高优化的版本

性能更好。

2) 妨碍编译器优化编译代码的因素

- 内存别名使用(memory aliasing)
  两个不同名的指针有可能指向内存同一块位置，这会妨碍编译器优化编译代码的策略。
- 函数调用的副作用
  有些函数调用会调用一些全局变量，那些全局变量的值受该函数调用次数的影响，由此产生函数调用的副作用。当然了，目前有一种被称为“内联函数替换 inline substitution”的过程对函数调用进行优化，但是缺点是任何尝试对这个调试进行追踪或者设置断点的尝试都会失败。

5.2 表示程序性能

1) 评价标准
   每元素的周期数 Cycle Per Element，CPE，表示程序的性能并指导我们改进代码的方法。说白了，就是每增加一个元素，需要增加多少个周期数

$$\mathrm{CPE} = \frac{dT}{dn}$$

当标明一个处理器是“4GHz”的时候，表明该处理器的运行时钟频率有 4GHz，周期为频率的倒数 0.25ns 或者 250ps

5.3 优化性能

1) 消除循环的低效率
   代码移动 code motion
   识别代码中要执行多次（例如在循环里）但是计算结果不会改变的计算，因而可以把计算移到代码前面不会被多次求值的部分。如：
   把循环中的判断条件尽量简单化，特别是当判断条件需要调用函数计算时，由于每次循环调用都会计算改判断的函数，会造成性能损失，所以有机会一定要把判断条件简单化。
   优化编译器的时候也会想着帮程序员进行尽可能的代码移动，但是正如咱们上面所讨论的，函数调用有可能会产生副作用，对于这个问题编译器必须非常小心。最好还是程序员显式地帮编译器完整代码移动的操作。
   对于一个长度为 n 的字符串，strlen 所用的时间与 n 成正比，因为对于 lower1 的 n 次迭代的每一次都会调用 strlen，所以 lower1 的运行时间正比于 n^2，CSAPP 说“这个函数对各种长度的字符串的实际测量值证实了上述分析”
   这个例子也说明了在编程过程中经常会存在“渐进低效率 asymptotic inefficiencyS”
2) 消除不必要的内存引用
   在汇编程序中，两个指针内容没办法直接进行逻辑或者运算操作，因为指令执行的过程是先进行执行阶段在进行访存的操作，所以。其中一个指针一般

先从内存中取出，然后放在某个寄存器中，然后利用寄存器进行相应的逻辑或者运算操作，最后再把寄存器中的值 mov 给该指针内存。这样存取内存的过程在循环中实在是没有必要的。可以先定义一个局部变量进行运算，循环结束后再 mov 给该指针变量的引用。

然而有人又提出来编译器应该帮我们程序员进行这样的优化，事实上由于内存别名使用的问题，编译器无法保证能进行安全的优化。

5.4 理解现代处理器

“现代处理器了不起的功绩之一是：它们使用复杂而又奇异的微处理器结构，其中，多条指令可以并行地执行，同时又呈现出一种简单的顺序执行指令的表象。

1) 两种下界描述处理器的最大性能
   - 延迟下界 latency bound
     因为在下一条指令开始之前，这一条指令必须完成；
   - 吞吐量下界 throughput bound
     刻画了处理器功能单元的原始计算能力，这个界限是程序性能的终极限制。
2) 乱序处理器
   据说性能比按 SQE+的要高，但是有点复杂。
   指令执行的顺序不一定跟它们在机器级程序中的顺序一致，整个涉及包括两个主要部分：
   - 指令控制单元 ICU，introduction control unit
     从内存中读出指令，并根据这些指令序列生成一组针对程序数据的基本操作；
   - 执行单元 CU，control unit
     执行这些操作；
3) 功能单元的性能
   跟据参考机和引用 Intel 的文献提供的
   延迟 latency：表示完成运算所需要的总时间；
   发射时间 issue time：表示两个连续同类型的运算之间需要的最小时钟周期数；
   容量 capacity：表示能够执行该运算的功能单元的数量；

表 5-1 参考机运算性能表

运算	整数			浮点数		
	延迟	发射	容量	延迟	发射	容量
加法	1	1	4	3	1	1
乘法	3	1	1	5	1	2
除法	3~30	3~30	1	3~15	3~15	1

发射时间为 1 的功能单元又称为***完全流水线化*** fully pipelined：每个时钟周期都可以开始一个新的运算。

除法器的发射等于延迟的时间，说明每条除法运算必须等上一条完成后才能进行下一条。

5.5 处理器操作的抽象模型

1) 对于行成循环的代码片段，将访问到的寄存器分为四类：
   - 只读：在循环过程中寄存器中的值不会被修改；
   - 只写：这些寄存器作为数据传送操作的目的；
   - 局部：在循环内部被修改和使用，但是每次循环之间互不相关，如条件码寄存器，在循环判断的时候使用；
   - 循环：在循环内部被修改和使用，而且这一次的修改会作为下一次循环的源值；
2) 关键路径
   这个比较难，最好还是看看书 P362

5.6 循环展开

1) 每次迭代结算两个前置和，从而使得迭代计算的次数减半，从而提升性能。
2) 代码示例：

```
//a 是一个已经初始化的数组，
int OP(int a[]){
    int length=strlen(a);
    int limit= length-(k-1);
    int i, result;
    for(i=0; i< limit; i+=k){
        result= (result OP a[i]) OP a[i+1] …… OP a[i+k-1];
    }
    for(; i< length; i++){
        result=result OP a[i];
    }
    return result;
}
```

这里有个需要注意的地方，就是为什么 limit 的值等于 length 减去(k-1)，而不是直接减去 k 呢？那是因为在第一个循环当中，每 k 个一组，从 i 开始，直到 i+k-1 就是 k 个了，如果一直到 i+k 的话，那就是 k+1 个了。所以对于正常情况下 i 对应的是 length，那么 i+k-1 对应的应是 length-(k-1)，也即 limit 值。

3) k × 1 展开

2）中的代码叫k × 1 展开，编译器很容易就进行循环展开，只要优化的级别

足够高，许多编译器用优化等级 3 或者更高的优化等级调用 GCC，他就会执行循环展开。

理论上来讲，当 k 不断增大的时候，程序的 CPE 会逐渐逼近延迟界限，但是无法突破延迟界限，因为在循环的关键路径上，每一个累计器的值都是下一个累计器的初始值，正是这个原因使得循环无法突破延迟界限。

5.7 提高并行性

1) 由于普通k × 1 展开无法打破延迟界限，要想进一步缩短程序的 CPE，唯有另辟蹊径。
2) 代码示例：

```
//a 是一个已经初始化的数组，
int OP(int a[]){
    int length=strlen(a);
    int limit= length-(k-1);
    int i, result1, result2, result3, …… resultk;
    for(i=0; i< limit; i+=k){
        result1= result1 OP a[i]) ;
        result2= result2 OP a[i+1]) ;
        result3= result3 OP a[i+2]) ;
        ……
        resultk= resultk OP a[i+k-1]) ;

    }
    for(; i< length; i++){
        result1=result1 OP a[i];
    }
    return (result1+ result2+ result3+ …… +resultk);
}
```

3) k × k 展开

k × k 展开最大的特点是在第一个循环中把原来只用一个累计器计算一组内的所有操作，改为用 k 个累计器分别计算一组内的操作，使得第一个循环在每次迭代过程中都可以充分利用处理器的流水性能，从而突破延迟界限。

4) 通常，只要保持能够执行该操作的所有功能单元的流水线都是满的，程序才能达到该操作的吞吐量极限。对于延迟为 L，容量为 C 的操作而言，要求循环因子k > L × C。
5) 不过有时候需要注意，数据可能会出现周期性的严重不连续，或者周期性的忽大忽小，而且大的很大，小的很小，这种情况很容易造成某些累计器出现上溢或者下溢的情况。当然了，这种情况很少出现，对大多数应用程序来讲，

使性能翻倍要比冒对奇怪的数据模式产生不同的结果的风险更重要。但是程序开发人员需要跟潜在的客户进行协商，看看是否有特殊的条件，可能会导致修改后的算法不能接受，大多数编译器并不会尝试对浮点数代码进行这种变换，因为它们没有办法判读引入这种会改变程序行为的转换所带来的风险，不论这种改变是多么小。

5.8 重新结合变换

1) 目前除了k × k 展开外，还有另外一种方法可以打破延迟极限。

2) 代码示例：

```
//a 是一个已经初始化的数组，
int OP(int a[]){
    int length=strlen(a);
    int limit= length-(k-1);
    int i, result;
    for(i=0; i< limit; i+=k){
        result= result OP (a[i]) OP a[i+1] OP …… OP a[i+k-1]);
    }
    for(; i< length; i++){
        result=result OP a[i];
    }
    return result;
```

这乍一看，不就跟k × 1 展开差不多吗？只是在第一个循环中，括号的地方改变了，这样需要进行操作的数组先进行结合，然后再跟累计器进行结合。通过结合顺序的改变，使第一个循环的每一次迭代累计器只使用一次。这样看来的实质跟k × k 展开类似。

3) k × 1a 展开

他还有一个地方跟k × k 展开类似，就是对于浮点数的操作。CSAPP 上的观点是“对于整数加法和乘法，这些运算是可结合的，这表示这种重新改变结合顺序对结果没有影响，对于浮点数运算的情况，必须再次评估这种重新结合是否有可能严重影响结果。我们会说，对于大多数应用来说，这种差别不重要。”

4) 大多数编译器不会尝试对浮点数运算进行重新结合，因为这些运算不保证是可结合的，当前的 GCC 版本会对整数的加法和乘法执行重新结合，但不是总会有好的效果。CSAPP 上的观点是“通常，我们发现循环展开和并行地累积在多个值中，是提高程序性能的更可靠的方法”

5.9 向量指令

1999 年，Intel 引入了 SSE 指令，全称”Streaming SIMD Extensons”流 SIMD 扩展，而 SIMD（读作”sim-dee”）”Single-Introduction, Mutiple-Data”(单指令多数据)。SSE 经历过多代，最新的版本为高级向量拓展(advanced vector extension)或者叫 AVX。目前 AVX 向量寄存器有 32 字节，可容纳 8 个 32 位数或者 4 个 64 位数，这些数据既可以是整数也可以是浮点数。他可以并行执行 8 组数值或者 4 组数值的加法或者乘法。
可以看到，向量代码几乎可以比原来的标量代码获得 8 倍性能的提升，

5.10一些限制的因素

吞吐量的限制实际上是发射时间的限制

1) 寄存器溢出
   在现在处理器中，寄存器的数量有限——16 个，累计器的数量超过寄存器的数量的时候，多余的累计器需要保存在内存的栈帧中。在循环的每次迭代中都需要从内存中取存需要更多的开销，造成的后果是当$k \times k$ 展开的 k 过大的时候，开销会更多，需要花费的时间成本不降反升。
2) 分支预测和预测错误的处罚
   说人话就是减少使用 if 条件的条件控制转移，而多使用三木运算符 A cmp B?A:B 的条件传送转移，这样可以减少预测错误的处罚，提高效率。
3) 内存的性能
   - 加载
   - 储存

5.11程序剖析

1) 其实这是一个程序，把他的代码插入到你的程序里面，可以确定程序的各个部分需要多少时间。剖析的一个有力之处是可以在现实的基准数据（benchmark data）上运行实际程序的同时，进行剖析。
2) 剖析程序必须进行编译和链接，使用 GCC 或者其他 C 语言编译器时，添加参数”-pg”，确保编译器不通过内联替换来尝试执行任何的优化，否则无法正确刻画函数调用，所以还要添加参数”-Og”：
   linux> gcc -Og -pg prog.c -o prog.out
   linux> ./ prog.out file.txt
   //其中 file.txt 时剖析程序运行时的参数。
   此时运行的时间比平时要慢许多，而且会产生一个新的文件 gmon.out。
   调用 GPROF 来分析 gmon.out 中的数据
   linux>gprof prog.out

六、 存储器层次结构

实际上，存储器系统 memory system 是一个具有不同容量，成本和访问时间的存储器设备的层次结构。计算机技术的成功很大程度上源自于存储技术的巨大进步。

6.1 存储技术

1) 随机访问存储器 Random-access Memory
   静态的 SRAM
   更快，更贵，作高速缓存的存储器，既可以在 CPU 芯片上，也可以在片下，但是一个桌面系统的 SRAM 不会超过几兆字节。
   动态的 DRAM
   作为主存和图形系统的帧缓存区，DRAM 却有几百或几千兆字节。
2) 静态的 SRAM
   双稳态存储器 bistable，只有两个稳定的状态，其他都是非稳态，非稳态会迅速转移到两个稳态之一的状态。原则上，只要供电，它会无限期保持它的值，即使遇到干扰，当干扰消除的时候，电路就会恢复到稳定值。
3) 动态的 DRAM
   DRAM 将每个位储存为对一个电容充电，但是电量很少，每个单元由一个电容和访问晶体管组成。DRAM 跟 SRAM 不同，DRAM 对干扰非常敏感，而且 DRAM 一旦受到干扰就无法恢复。很多原因会导致电容漏电，比如光照等，会使 DRAM 单元在 10~100 毫秒时间失去电荷。幸运的是，电路的时间计量单位是纳秒级的
4) 对比：

表 6-1 SRAM 和 DRAM 对比

	每位晶体管数	相对访问时间	持续的？	敏感的？	相对花费	应用
SRAM	6	1*	是	否	1000*	高速缓存存储区
DRAM	1	10*	否	是	1*	主存，帧缓冲区

5) 传统的 DRAM
   传统的 DRAM 单元（位）由 d 个超单元 supercell 组成，每个超单元包含有 w 位，所以一个 d*w 的 DRAM 一个储存了 dw 位的信息。
   每个 DRAM 芯片被连接到某个内存控制器 memory controller 的电路。这个电路可以一次传送 w 位或者一次传出 w 位。访问 DRAM 芯片的时候，内存控制器首先把行地址 i(Row Access Strobe，行访问选通脉冲)发送，接着把列地址 j(Column Access Strobe，列访问选通脉冲)发送。
   电路设计者把 DRAM 设计成二维阵列而不是线性组合，原因是想减少针脚的数量，但是缺点是必须分两步发送地址，增加了访问的时间。
6) SRAM 和 DRAM 都是易失性存储器(volatile)，断电后即丢失他们的信息。而非易失性存储器，即使关电后，仍然保持这他们的信息

7) PROM(Programmable ROM，可编程 ROM)
只能被编程一次，PROM 的每个存储器内有一种熔丝(fuse)，只能用高电流熔断一次。
8) EPROM(Erasable Programmable ROM，EPROM)可擦写可编程
它里面有一个透明的石英窗口，允许光到达储存单元，当紫外线光照进窗口的时候，EPROM 单元就会被置 0。
9) 闪存 flash memory
一类非易失性存储器，基于 EEPROM。EEPROM 与 EPROM 不同的地方在于，前者是电子可擦除的 PROM。

6.2 访问主存

1) 数据流通过称为总线 bus 的共享电子电路在处理器和 DRAM 主存之间来来回回。这些步骤称为总线事务 bus transaction。
   - 读事务 read transaction
     从主存传送数据到 CPU
   - 写事务 write transaction
     从 CPU 传送数据到主存
2) 总线
总线是一组并行的导线，能携带地址，数据和控制信号

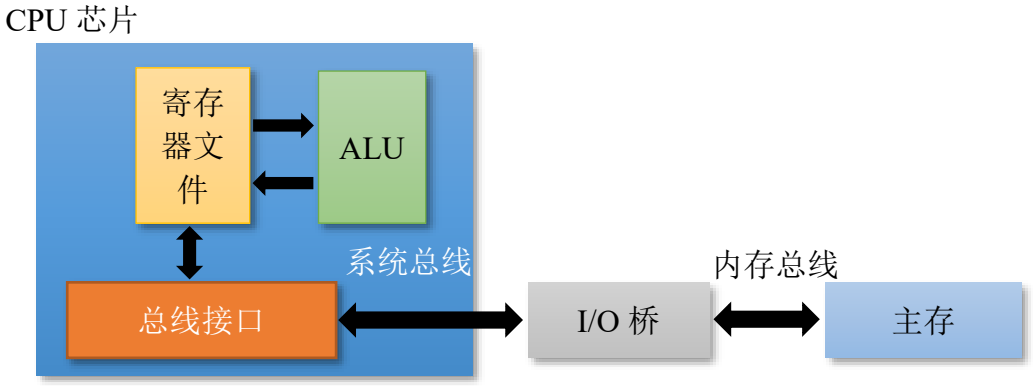


图 6-1 连接 CPU 和主存的总线结构示例

3) I/O 桥接器将系统总线的电子信号翻译成内存总线的电子信号。
4) 一个加载操作的过程：
movq A, %rax
   - CPU 把地址 A 放到系统总线上，I/O 桥接器将信号传递到内存总线；
   - 主存感受到内存总线上的地址信号，从内存总线上读地址，从 DRAM 中取出数字，并将数据写到内存总线，I/O 桥将内存总线上的信号翻译成系统总线信号，然后沿着系统总线传递；
   - CPU 感受到系统总线上的数据，从总线上读数据，并将数据复制到寄存器%rax；

movq %rax, A
   - CPU 把地址 A 放到系统总线上，I/O 桥接器将信号传递到内存总线，内

存从内存总线上读取地址，并等待数据到达；

- CPU 将%rax 中的数据字复制到系统总线；
- 主存从内存总线读出数据字，并将这些位存储到 DRAM；

6.3 磁盘存储

1) 磁盘容量

$$磁盘容量 = \frac{字节数}{扇区} \times \frac{平均扇区数}{磁道} \times \frac{磁道数}{表面} \times \frac{表面数}{盘片} \times \frac{盘片数}{磁盘}$$

2) 扇区 sector
每个扇区包含相同的数据位，通常 512 字节，扇区之间由一些间隙 gap 分开，这些间隙不存在数据位，间隙存储用来标识扇区的格式化数据位。扇区的数目是由最靠内的磁道能记录的扇区数决定。为了保持每个磁道有固定的扇区数，越往外的磁道扇区隔得越开。
3) 盘片 platter
表面 surface
磁道 track
磁盘 disk
柱面 cylinder
4) 对于与 DRAM 和 SRAM 容量相关的计量单位，通常 K=2^{10}，M=2^{20}，G=2^{30}，T=2^{40}；对于像磁盘和网络这样的 I/O 设备容量相关的计量单位，通常以 K=10^3，M=10^6，G=10^9，T=10^{12}。速率和吞吐量也是使用这些前缀。
5) 磁盘以扇区大小的块来读写数据。对扇区的访问时间 access time 主要由三部分组成：寻道时间 seek time，旋转时间 rotational latency 和传送时间 transfer time。
   - 寻道时间 seek time
     取决于传动臂在盘面上的移动的速度，$T_{avg\ seek}$ 通常为 3~9ms
   - 旋转时间 rotational latency
     经过寻道时间找到期望的磁道后，便在期望磁道上寻找期望扇区。其速度取决于旋转时间 rotational latency。$T_{avg\ rotation}=0.5* T_{max\ rotation}$
   - 传送时间 transfer time
     读或者写的时间。
6)

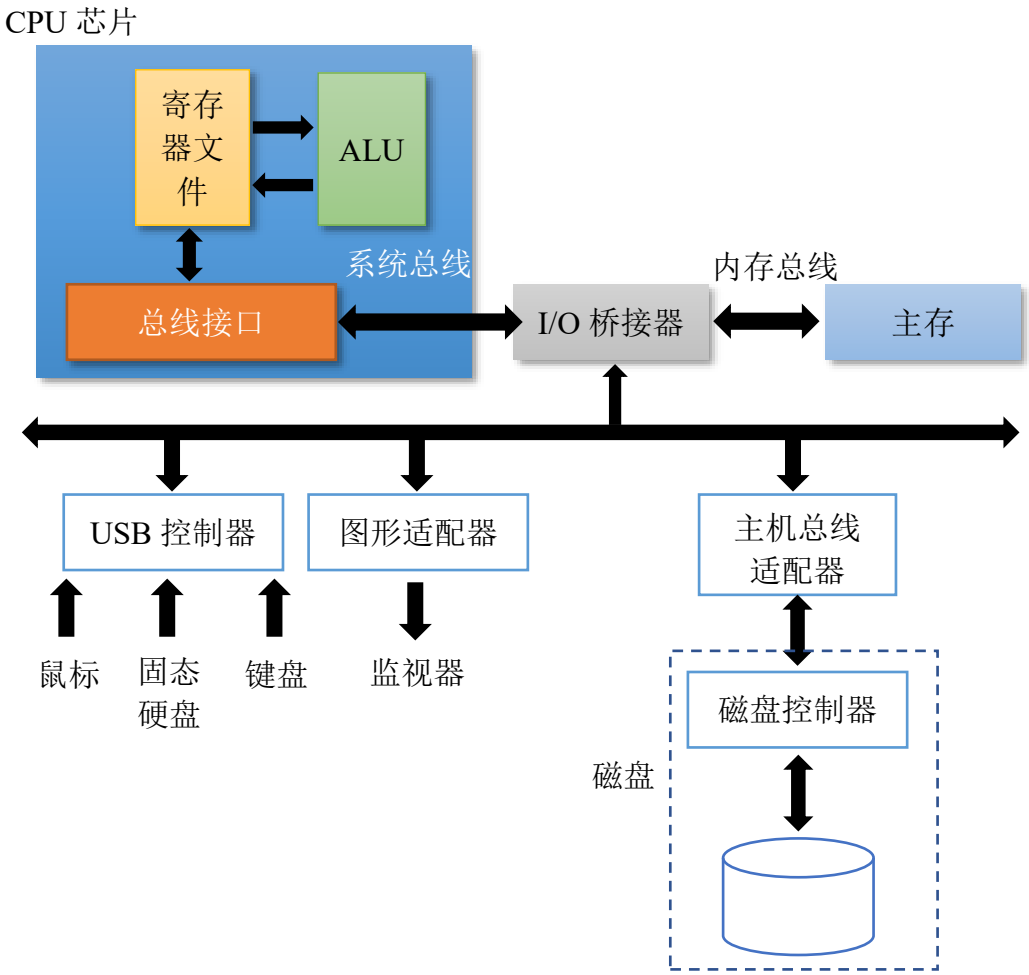


图 6-2 总线结构示例，它连接 CPU，主存和 I/O 设备

6.4 局部性 locality

1) 倾向于引用其他最近引用过的数据项的数据项，或者最近引用过的数据项本身。这种倾向性称为局部性原理 principle of locality，这是一个持久的概念。
   - 时间局部性 temporal locality
     具有良好时间局部性的程序中，被引用过一次的内存位置很可能在不远的将来再被多次引用。
   - 空间局部性 spatial locality
     具有良好空间局部性的程序中，被引用过一次的内存位置附近的内存位置很可能在不远的将来被多次引用。
2) 其实整个计算机系统从硬件，到操作系统，再到应用程序，各个层次都应用了局部性的原理：
   硬件：高速缓存保存最近被引用的指令和数据项；
   操作系统：使用主存作为虚拟地址空间的最近被使用块的高速缓存；
   应用程序：例如 web 浏览器将最近被引用的文档放在本地磁盘上。

3) 对程序数据引用大的局部性
在一个循环程序里面，每次迭代都会用到累计器 s，那么它具有良好的时间局部性；而对于循环里面每一个数组元素 a[i]来讲，它具有良好的空间局部性。从这个角度上去讲，可以断定该函数具有良好的局部性。
对于二维或者更高维的数组而言，如果对他们的引用是遵循行优先原则，则会有很好的空间局部性，否则会得到很差的空间局部性。
4) 取指令的局部性
对于循环体里面的指令而言，由于指令是连续执行的，所以具有很好的空间局部性，同时每次迭代都会用到这些指令，因而也具有良好的时间局部性。

6.5 缓存

1) 储存器层次结构的中心思想：对于每个 k，位于 k 层的储存器作为位于 k+1 层储存器的缓存，最小的缓存，CPU 寄存器组。一般地，第 k+1 层存储器被划分成连续的数据对象租块 block，每一个块都有对应的位移的地址或者名字。而第 k 层也拥有类似的块，每个块的大小或者容量都跟第 k+1 层相同，只是块的数量比第 k+1 层少。在任何时刻，第 k 层的缓存包含第 k+1 层块的一个子集的副本。数据总是以块大小为传送单位 transfer unit 在第 k 层和第 k+1 层之间来回复制。
2) 缓存命中 cache hit
当程序需要第 k+1 层的某个数据对象 d 时，它首先在当前储存在第 k 层的一个块中查找 d，若 d 刚好还存在第 k 层，则称缓存命中 cache hit
3) 缓存不命中 cache miss
若第 k 层中没有我们所需要的数据对象 d，我们就说缓存不命中 cache miss，此时需要从第 k+1 层选取包含数据对象 d 的那个块，来覆盖第 k 层的一个块，覆盖的过程称为替换 replacing 或者驱逐 evicting 这个块。被驱逐的那个块被称为牺牲块 victim block。决定要驱逐哪个块由缓存策略 replacement police 来控制。
缓存不命中的种类
强制性不命中 compulsory miss 或者冷不命中 cold miss
空的缓存称为冷缓存 cold cache，只要发生了不命中，就执行某个放置策略 placement policy，确定它从第 k+1 层中取出的块放在哪里。最灵活的策略就是允许来自第 k+1 层的任何块放在第 k 层的任何块中。对于存储器层次结构的高层缓存（靠近 CPU），他们是用硬件来实现这种任意策略的。不过这种任意策略的随机定位代价很高。所以硬件缓存一般会执行严格的放置策略。比如使用余数储存策略，第 k+1 层标号 mod 第 k 层的块总数。
冲突不命中 conflict miss
比如查找块 8，然后块 0，接着又查找块 8，然后块 0，如此循环抖动 thrash，抖动这个词用得非常好，非常形象生动^< ^。抖动描述的是一种情况：即高速缓存反复地加载和驱逐相同的高速缓存块的组。如:

```
for(i=0; i<n; i++){
    sum+=x[i]*y[i];
```

}

容量不命中 capacity miss

比如一个数组的数目超过了缓存的大小

4) 储存器层次结构：每一层都需要缓存管理。

6.6 高速缓存存储器

L1 高速缓存：大约需要 4 个时钟周期；

L2 高速缓存：大约需要 10 个时钟周期；

L3 高速缓存：大约需要 50 个时钟周期；

1) 一般而言，高速缓存的结构可以使用元组(S, E, B, m)来描述，容量 C=S*E*B 指的是所有块的大小之和，标记位和有效位不在内。$S=2^s$ 个高速缓存组 cache set，每个组包含 E 个高速缓存行，每个行由一个 $B=2^b$ 个字节的数据 block 组成，一个有效位 valid bit 指明这个行是否包含有意义的信息，还有每行 t=m-(b+s)个标记位（tag bit）指明这个组中的哪一行包含这个字，如果 t 是 n 位的话，那么第 k 层有 S*E 行 ，而第 k+1 行有 2^n*S*E 行，s 位表示组索引，指明这个字必须储存在哪个组中，b 位块偏移，指明在 B 个字节的数据块中字偏移。

2) 直接映射高速缓存，每一组只有一行

抽取被请求的字 w 的过程分为三个步：

- 组选择

  从 w 的地址中间抽取 s 个组索引位，这些位被解释为一个对应于一个组号的无符号整数，或者说是一个一维数组，于是来到对应的组。

- 行匹配

  由于每组只有一个行，然后检查 w 地址的标记位跟高速缓存行的标记位是否匹配，如果匹配则说明 w 的数据在高速缓存中存在，缓存命中，否则缓存不命中。

- 字抽取

  跟据偏移量抽取即可；

如果映射不命中，则进行行替换。替换标记位和组索引。

每一组是只有一行是造成冲突不命中的主要原因。

3) 组级联高速缓存

4) 全相联高速缓存

5) 高速缓存参数的性能指标

不命中率

命中率

命中时间

不命中处罚

6) 较高的级联度（即 E 值比较大）的优点是能降低由于冲突不命中出现的抖动的可能性

7) 编写高速缓存友好的代码

- 对局部变量的反复引用是好的；

- 步长为 1 的引用模式是好的；

8) 存储器山

一个程序从储存器中读数据的速率称为读吞吐量 read thoughput，或者叫读带宽 read bandwidth

9) 对于多个循环嵌套，最好保证每一层循环内部的步长都为 1，这样的性能最高。

10) 以下是一些利用局部性的示例，假设高速缓存块是四个字，每个字 4 个字节，初始缓存为空。如：

```
for(i=0; i<N; i++){
    for(j=0; j<N; j++){
        S+=a[i][j];
    }
}
```

	j=0	j=1	j=2	j=3	j=4	j=5	j=6	j=7
i=0	1[m]	2[h]	3[h]	4[h]	5[m]	6[h]	7[h]	8[h]
i=1	9[m]	10[h]	11[h]	12[h]	13[m]	14[h]	15[h]	16[h]
i=2	17[m]	18[h]	19[h]	20[h]	21[m]	22[h]	23[h]	24[h]
i=3	25[m]	26[h]	27[h]	28[h]	29[m]	30[h]	31[h]	32[h]

不命中率为 0.25

七、 链接

链接 linking 是将各种代码和数据片段收集合成为一个单一文件的过程，这个文件可被加载（复制）到内存执行。在现代系统中，链接是由链接器 linker 的程序自动执行的。

可被执行的时机包括：

- 编译时 compile time
  被翻译成机器代码时
- 加载时 load time
  被程序加载器加载到内存并执行时
- 运行时 run time
  由应用程序来执行

7.1 学习链接的作用

1) 理解链接器将帮助你构造大型程序
2) 理解链接器将帮助你避免一些危险的编程错误
3) 理解链接将帮助你理解语言的作用域规则是如何实现的
   如：全局变量和局部变量，static 变量等
4) 理解链接将帮助你理解其他重要的系统概念
   如：加载和运行程序，虚拟内存，分页和内存映射

5) 理解链接将使你能够利用共享库

7.2 编译器驱动程序

大多数编译系统提供编译器驱动程序 compiler driver，也就是我们平常用的一步到位的 gcc 命令。它代表用户在需要时调用语言预处理器，编译器，汇编器和链接器。

- 预处理器 cpp
  驱动程序首先运行 C 预处理器 cpp，它将 C 的源程序 main.c 翻译成一个 ASCII 码的中间文件 main.i。
  cpp [other arguments] main.c main.i
- 编译器 ccl
  它将 main.i 翻译成汇编文件 main.s
  ccl main.i -Og [other arguments] -o main.s
- 汇编器 as
  它将 main.s 翻译成可重定位目标文件(relocatable object file) main.o
  as main.s -Og [other arguments] -o main.o
- 链接器 ld
  它将 main.o 翻译成可执行目标文件(executable object file) main.out
  ld main.o sum.o -o main.out [other arguments]
- 运行可执行文件
  linux> ./ main.out
  shell 调用操作系统中一个叫作加载器 loader 的函数，它将可执行文件 main.out 的代码和数据复制到内存中，然后控制转移到这个程序的开头。

7.3 静态链接和目标文件

1) 静态链接
   以一组可重定位目标文件和命令行参数作为输入，生成一个完全链接的可以加载运行的可执行目标文件作为输出。
   - 符号解析 symbol resolution
     目标文件中的每个符号，其实都对应着一个函数，一个全局变量或者一个静态变量。符号解析的目的是将每一个符号引用正好和一个符号定义关联起来；
   - 重定义 relocation
     编译器和汇编器生成从地址 0 开始的代码和数据节，链接器通过把每个符号定义与一个内存位置关联起来。
2) 目标文件
   - 可重定位目标文件
   - 可执行目标文件
   - 共享目标文件

一种特殊的可重定位目标文件，可以加载或者运行时被动态地加载进内存并链接。

3) 从技术上讲，一个目标文件 object file 就是一个以文件形式储存储存在磁盘的目标模块 object module。所谓目标模块，就是一个字节序列。

4) 各个系统的目标文件的格式都不相同
   - Windows 系统
     可移植执行执行 Portable Executable, PE
   - Mac OS-X
     Mach-O 格式
   - 现代 x86-64 的 Linux 和 unix 系统
     可执行可链接格式 Executable and Linkable Format, ELF

5)

表 7-1 典型的 ELF 可重定位目标文件

节名称	功能
ELF header	以 16 字节开始，描述了生成该文件的系统的字的大小和字节顺序。剩下的部分用来帮助链接器语法分析和解释目标文件的信息。
.text	已编译程序的机器代码
.rodata	只读数据，如 printf 函数，开关函数的跳转表
.data	已初始化的全局和静态 C 变量，局部变量的运行时被保存在栈中，不在.data 中，也不在.bss 中
.bss	未初始化的全局和静态 C 变量，以及初始化为 0 的全局和静态 C 变量
.symtab	一个符号表，包括定义和引用的函数和全局变量的信息。和编译器中的符号表不同，.symtab 不包括局部变量的条目
.rel.text	一个.text 节中位置列表。当链接器需要把目标文件和外部文件组合时，需要改变这些位置，比如引用外部函数或者全局变量。
.rel.data	被引用或者定义的所有全局变量和外部函数的重定位信息。一般来讲，任何已初始化的全局变量，如果它的初始值是一个全局变量的地址或者外部定义函数的地址。都需要被修改。
.debug	一个调试表，其条目包括程序中定义的局部变量和类型定义，程序中定义和引用的全局变量，以及原始 C 源文件，但是只有使用-g 来调用编译器驱动程序的时候才有。
.line	原始 C 源程序中的行号和.text 中机器代码指令之间的映射，只有使用-g 来调用编译器驱动程序的时候才有。
.strtab	一个字符串表，其内容包括.symtab 和 debug，以及节头部中的节名字。实质是以 null 结尾的字符串序列。
节头部表	描述不同节的大小和位置

.bss，起始于 IBM 704 汇编语言，大约在 1957 年中“块储存开始 Block Storage Start”指令的首字母缩写，现代可以看作是 Better Save Space 的缩写。

7.4 符号和符号表

1) 每个可重定位的目标模块都有一个符号表，它包含 m 定义和引用的符号和信息。
2) 链接器的上下文中，有三种不同的符号
   - 由模块 m 定义并能被其他模块引用的全局符号
   - 由其他模块定义并能被模块 m 引用的全局符号
   - 只被模块 m 定义和引用的局部符号
3) .symtab 中的符号表不包括本地非静态程序变量的所有符号，因为这些局部符号在程序运行的时候都归栈管理，链接器对这些东西不感兴趣。
4) 假设在同一模块中的两个函数各自定义了一个静态局部变量 x，在这种情况下，编译器向汇编器输出两个不同名字的局部链接器符号，比如，他可以用 x.1 表示函数 f 中的定义，用 x.2 表示函数 g 的定义。
5) 在 C 中，源文件扮演着模块的角色，任何带有 static 属性的声明的全局变量或者函数都是模块私有的，不能被其他模块访问，尽可能用 static 属性保护你的变量是很好的编程习惯。

7.5 符号解析

1) 当编译器遇到一个不在当前模块定义的符号大的时候，会假设该符号已经在其他模块中定义，生成一个链接器符号表条目，并交给链接器处理，如果链接器在它输入的任何模块中都找不到该符号，就会输出条错误信息并终止。
2) 当遇到相同名字的全局变量的时候，链接器必须要么标志一个错误，要么以某种方式选出一个定义并抛弃其他的定义。幸运的是，在 C++和 Java 中允许使用重载的方法。
3) 对于可重定位目标文件的符号表中，函数和已初始化的全局变量为强符号，未初始化的全局为弱符号
   链接器对于重定义的强弱符号的规则如下
   - 不允许有多个重名的强符号；
   - 如果有一个强符号和多个弱符号，则选择强符号；
   - 如果有多个若符号重名，则任意选择一个；

   对于第三条规则，编译器不会报错，但是会警告。如果我们需要把这类警告转化为错误的话，可以使用命令 GCC -fno -common。或者使用-Werror 选项，把所有的警告都报为错误。

7.6 与静态库链接

1) 链接提供一种机制：把相关的模块打包成一个单独的文件
2) 链接的时候需要复制静态库中的被应用程序引用的目标模块，减少了可执行文件在磁盘和内存中的大小。

3) 在 Linux 中，静态库是以一种存档 archive 的特殊文件格式存放在磁盘中，存档文件是一组连接起来的可重定位目标文件的集合，后缀名为.a
4) 在链接的过程中，编译器从左到右扫描目标模块和存档文件，链接器维护一个可重定位的目标文件的集合 E，一个未解析的符号集合 U，以及一个在前面输入文件中已经定义的符号集合 D，初始时，集合 E，U 和 D 都为空。扫描过程中不断有 U 转化为 D，当完成扫描后，U 必须时非空的，否则会报错。
5) 一般把库放在它们命令行的结尾
在命令行中，如果定义一个符号的库出现在引用这个符号的目标文件之前，那么引用就不能被解析，链接会失败。

7.7 动态链接共享库

1) 正如 window 系统中的 dll 文件，它是一个目标模块，在运行和加载时，可以加载到任意的内存地址，并和一个在内存中的程序链接起来，这个过程称为动态链接 dynamic linking

7.8 库打桩机制

1) 你可以截获共享库函数的调用，取而代之执行自己的代码
2) 这里有点难，忽略了

八、 异常控制流

8.1 库打桩机制

九、 虚拟内存

9.1 定义和功能

1) 虚拟内存 VM 是硬件异常，硬件地址翻译，主存，磁盘文件和内核软件完美交互。为每个进程提供了一个大的，一致的和私有的地址空间。但储存在磁盘中 N 个连续地址。
2) 功能：
   - 将主存看成是一个存储在磁盘上的地址空间的高速缓存，主存只保留活动区域，并根据需要在磁盘和主存之间来回传递数据；
   - 为每个进程提供了一致的地址空间；
   - 保护了每个进程的地址空间不被其他进程破坏。
3) 主存的组织形式

计算机系统的主存被组织成一个由 M 个字节大小的单元组成的数组，每字节都有一个唯一的物理地址 physical address, PA，第一个地址为 0，第二个为 1，如此类推。

4) 早期的 PC 使用物理寻址，然而现代处理器使用的是一种称为虚拟寻址 virtual addressing。使用虚拟寻址时，CPU 先发出一个虚拟地址 VA，这个虚拟地址被传到内存之前，先转换成适当的物理地址。这个翻译的任务称为地址翻译 address translation。CPU 芯片上叫做内存管理单元 Memory Manage Unit, MMU 的硬件，负责利用放在主存上的查询表来动态地翻译虚拟地址，该表的内容由操作系统管理。

9.2 地址空间

1) 地址空间 address space
   地址空间是一个非负整数地址的有序集合{0, 1, 2, 3…}，如果地址是整数连续的，则称为它是一个线性地址空间 linear address space。
2) 虚拟地址空间和 n 位地址空间
   CPU 从 2^n 的地址空间中产生虚拟地址空间，一个地址空间的大小由表示最大地址所需要位数来描述。所以数字 n 意味着该空间是一个 n 位地址空间
3) 物理地址空间
   物理地址空间对应于系统中物理内存的 M 个字节。这里的 M 不要求是 2 的幂。

9.3 虚拟内存作为缓存的工具

1) 由于虚拟内存的概念出现在 SRAM 概念之前，那时候的“页”即是现在“块”的概念。在磁盘和内存之间传送页面的活动叫做交换 swapping 或者页面调度 paging。当有页不命中时，才换入页面的这种策略称为按需页面调度 demand paging。
2) VM 系统通过将虚拟内存分割为虚拟页 Virtual Page, VP 的大小固定的块来解决这个缓存的问题。每个虚拟页的大小为 $P=2^p$ 字节。类似的，物理内存也分割为物理页 Physical Page, PP，大小也是 P 字节，被称为页帧，page frame。
3) 任意时刻，虚拟空间都是由以下三个不相交的子集组成
   - 未分配的
   - 缓存的
   - 未缓存的
4) 页表
   在内存中负责被 MMU 用来把虚拟地址翻译成物理地址。页表由页表条目 page table entry, PTE 的数组组成。每个 PTE 由一个有效位 valid bit 和一个 n 位地址字段组成。有效位表明这是否一个空地址或者没有被分配。也只有页表知道 VP 的字节有没有被缓存。
5) 物理地址在内存上；虚拟地址由 CPU 生成，但储存在磁盘中 N 个连续地址

6) 分配页

如：使用 malloc 命令

首先在磁盘上分配空间储存虚拟地址 VP，然后更新主存中的页表条目，使他指向在磁盘中新创建的页面。

7) 页命中

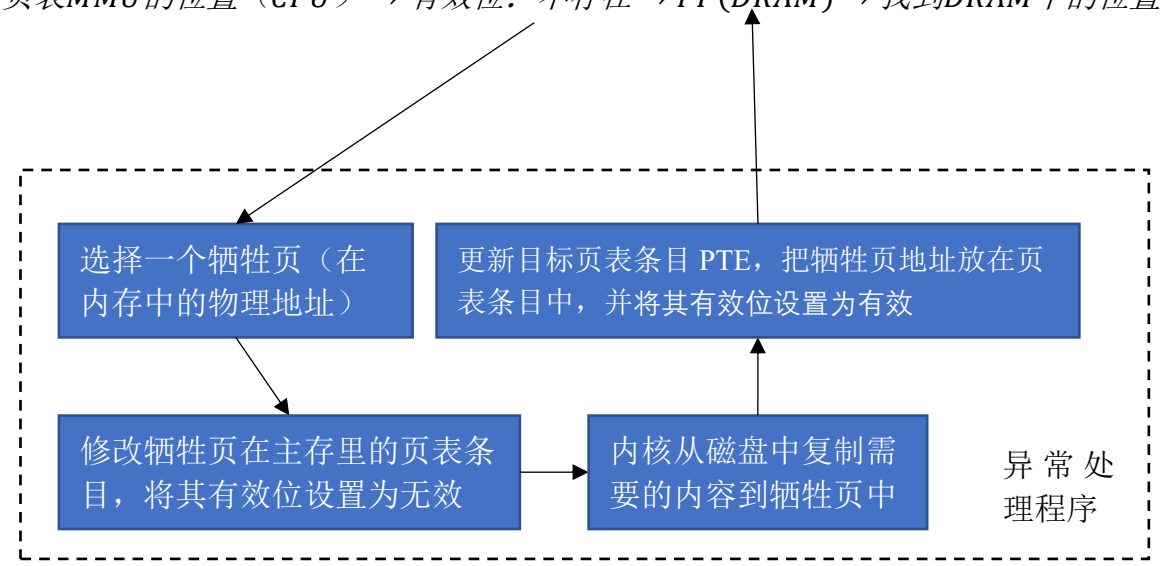
$\text{VP}(\text{CPU 指令生成，但储存在磁盘中 N 个连续地址}) \rightarrow PTE索引$

$\rightarrow 在页表MMU的位置（CPU）\rightarrow 有效位：存在 \rightarrow PP(DRAM) \rightarrow 找到DRAM中的位置$

8) 缺页

$\text{VP}(\text{CPU 指令生成，但储存在磁盘中 N 个连续地址}) \rightarrow PTE索引$

$\rightarrow 在页表MMU的位置（CPU）\rightarrow 有效位：不存在 \rightarrow PP(DRAM) \rightarrow 找到DRAM中的位置$



9.4 虚拟内存作为内存管理的工具

1) 实际上，操作系统为每个进程提供了一个独立的虚拟地址空间。
2) 按需页面调度和独立的虚拟地址空间的结合，对系统中的内存的使用和管理产生深远的影响。特别地，VM 简化了链接和加载，代码和数据共享，以及应用程序的内存分配。
   - 简化链接

     对于 64 位系统来讲，代码段总是从虚拟地址 0x400000 开始，数据段在代码段之后，中间有一段符合要求的对齐空白。这样的一致性极大地简化了链接器的设计和实现，使得这些可执行文件可以独立于物理内存中代码和数据的最终位置。
   - 简化加载

     Linux 加载器为为代码和数据在磁盘中分配虚拟页，把页表条目指向相应的代码和数据，并把他们标记为无效（即未被缓存）。有趣的是，加载

器并不执行调度的工作。只有被需要的时候，虚拟内存系统会按照需要自动调入数据页。

另外，将一组连续的虚拟页映射到任意一个文件的任意位置的表示法称作内存映射 memory mapping。Linux 提供一个称为 mmap 的系统调用，允许应用程序自己做内存映射。

- 简化共享

  操作系统为每个进程安排一个单独私有的连续的虚拟页面，另外，对于共享的代码，操作系统允许不同的虚拟页映射到相同的物理页面作为这部分代码的副本，如内核和 C 标准库的副本。
- 简化内存分配

  当一个进程需要额外的堆空间的时候，操作系统分配一个适当的数字 k 个连续的虚拟内存页面，并且将他映射到物理内存中任意位置的 k 个任意的物理页面。

9.5 虚拟内存作为内存保护的工具

1) 提供独立的地址空间使得区分不同进程的私有内存变得容易，除非所有的共享者都显式地允许它这么做（通过调用明确的进程间通信系统调用）
2) 通过在 PTE 添加一些允许位，来控制访问他的权限。如 SUP，READ 和 WRITE 的允许位。
3) 如果一条指令违反了上述的允许位规定，就会触发系统异常——故障。Linux Shell 把这个异常报告称为“段错误 segmentation fault”。

9.6 地址翻译

1) 地址翻译是一个 N 元素的虚拟地址(VAS)和一个 M 元素的物理地址(PAS)中元素之间的映射

$$\text{MAP：VAS} \rightarrow \text{PAS} \cup \emptyset$$

这里

$$\text{MAP} = A' \textit{ 如果虚拟地址} A \textit{ 中的数据在} PAS \textit{ 的物理地址} A' \textit{ 处}$$

$$= \emptyset \textit{ 如果虚拟地址} A \textit{ 中的数据不存在} PAS \textit{ 之中}$$

2) CPU 中有一个控制寄存器——页表基址寄存器 Page Table Base Register，PTBR 指向当前页表。
3) 基本参数

表 9-5 地址翻译的基本参数

种类	基本参数	含义
虚拟地址 VA	VPO	虚拟地址偏移量（字节）
	VPN	虚拟页号
	TLBI	TLB 索引
	TLBT	TLB 标记
物理地址 PA	PPO	物理地址偏移量（字节）
	PPN	物理页号
	CO	缓存块内的字节偏移量
	CI	高速缓存索引
	CT	高速缓存标记

4) 虚拟地址由 p 位的虚拟页偏移量 VPO 和 n-p 位虚拟页号 VPN 组成，页表基址寄存器读取 VPO 值到达相应的页表条目 PTE，然后根据访问权限获得里面的物理页号 PPN，而物理页偏离量 PPO=VPO，所以得到 PPN+PPO，从而获得物理全地址。
5) 处理命中页的具体步骤（页面命中完全是由硬件来处理的）
   - 处理器生成一个虚拟地址，并把它传送给 MMU
   - MMU 生成 PTE 地址，并从高速缓存/主存请求想得到 PTE
   - 高速缓存/主存向 MMU 返回 PTE
   - MMU 构造物理地址，并把它传送给高速缓存/主存
   - 高速缓存/主存返回所请求的数据字给处理器。
6) 处理缺页的具体步骤（页面命是由硬件和操作系统内核来处理的）
   - 处理器生成一个虚拟地址，并把它传送给 MMU
   - MMU 生成 PTE 地址，并从高速缓存/主存请求想得到 PTE
   - 高速缓存/主存向 MMU 返回 PTE
   - PTE 的有效位是 0，所以 MMU 触发了一次异常，传递 CPU 中的控制到操作系统内核中缺页异常的处理程序、
   - 缺页处理程序确定出物理内存中的牺牲页，如果这个页面已经被修改了，则把他换出到硬盘
   - 缺页处理程序页面调入新的页面，并更新内存中的 PTE，包括原来指向牺牲页的 PTE 和新的 PTE。
   - 缺页处理程序返回到原来的进程，再次执行导致缺页的指令，
   - 执行处理命中页的具体步骤
7) 结合高速缓存和虚拟地址寻址

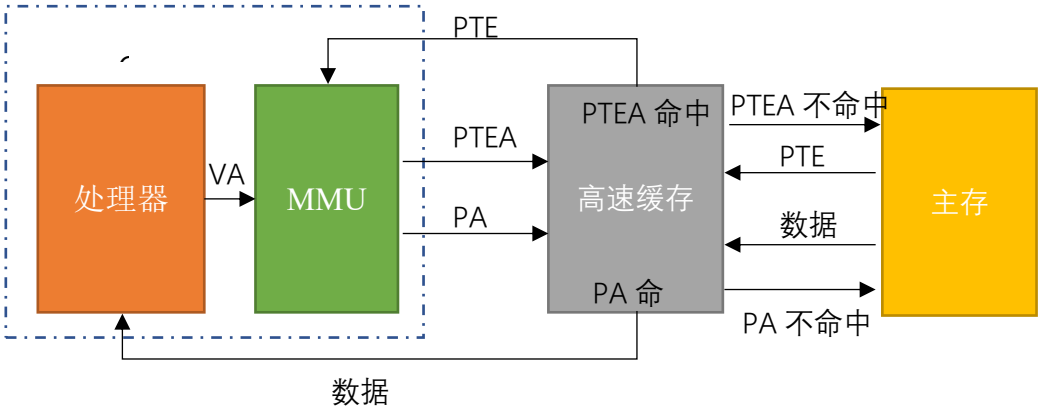


图 9-1 高速缓存和虚拟地址寻址结合

8) 利用 TLB 加速地址翻译

看着图 9-1，可以发现每次寻址都需要经过一次 PTE 的查询，搞不好多一次的内存寻址，这开销很大，为了减少开销，许多系统都想消除这样的开销。于是，他们在 MMU 中做了一个关于 PTE 的缓存，叫翻译后备缓存器 Translation Lookaside Buffer，TLB

用于组选择和行匹配的索引和标记字段是从虚拟地址中的虚拟页号 VPN 中提取出来的，如果 TLB 有 2^t 个组，那么 TLB 索引 TLBI 是由 VPN 的低 t 位组成的，其余的位组成 TLB 标记 TLBT

9) 多级页表

从以下两个方面减少内存的要求

如果一级页表中的一个 PTE 是空的，那么相应的二级页表就根本不会出现，这代表着一种巨大的潜在节约，因为对于一个典型的程序，4GB 的虚拟内存地址空间大部分都是未分配的。

只有一级页表才需要总在主存中，虚拟内存系统可以在需要的时候创建，页面调入或者调出二级页表，这就减少了主存的压力；只有最常用的二级页表才需要缓存在主存中。

第 j 级的页表中的每个 PTE，都指向第 j+1 级的页表的基址

10) 案例研究

- Intel Core i7
- Linux

  有空再看吧

9.7 内存映射

Linux 通过将一个虚拟内存区域与一个磁盘上的对象 object 关联起来，已初始化这个虚拟内存区域的内容，这个过程称为内存映射 memory mapping

1) 映射的对象包括

- Linux 文件系统中的普通文件
- 匿名文件

  由内核创建的，包含的全是二进制零

2) 一个对象可以被映射到虚拟内存中的一个区域，要么作为共享对象，要么作

为私有对象。

3) 进程对一个共享对象进行任何的写操作，其他的进程都会看得到，而且这些操作会反映到磁盘上的原始对象。
4) 即使对象映射到共享区域，物理内存中也只需要存放共享对象的一个副本。这个私有对象的副本可以被不同的进程映射到自己虚拟内存当中，私有对象的每个页面都会被标记为只读。如果进程想要写私有对象的某个页面，则会触发一个保护故障：
   进程试图写私有的写时复制区域时，它就会在物理内存中创建这个页面的一个新副本，更新页表条目指向这个新的副本，然后恢复这个页面的可写权限，然后返回控制。
5) 私有对象使用一种称为写时复制 copy-on-write 的巧妙技术被映射到虚拟内存中
6) fork 函数
7) execve 函数
   execve(“a.out”, NULL, NULL) //在当前进程中加载并执行 a.out
   步骤如下：
   - 删除已存在的用户区域；
   - 映射私有区域
     为新程序的代码，数据，bss 和栈区域创建新的区域结构
     加载运行时堆和 bss 是私有的，请求二进制零的匿名文件
   - 映射共享区域
   - 设置程序计数器 PC
8) 使用 mmap 和 munmap 创建和删除虚拟内存区域和内存映射
9) 动态内存分配
   显式内存分配，malloc
   隐式内存分配，也叫垃圾收集器 garbage collector
10) 后面实现分配器的问题跟数据结构的内容有点像，这里就不进行学习了，有空再学。