# Computer Systems

# A Programmer’s Perspective(3rd)

正式开始看龙书了，有点儿小兴奋。据说这本书的讲述的是一个应用的骑士跟一条恶龙斗争的故事……我瞎编的。事实上应该是将软件与硬件交接的部分。看完应该会很有收益，那事不宜迟，正式开始吧^< ^。

## 计算机系统漫游

### 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。

### 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。

### 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。

1. I/O设备
2. 主存

即内存，是由一组动态随机存取存储器（DRAM）芯片组成。

1. 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；

### 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。

### 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。

1. 虚拟内存

对主存和磁盘I/O设备的抽象表示。

1. 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。

### 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：

性能提升最好的表示方法是 ，如果性能有提升则大于1，用“×”表示比例，写作“”，都作“倍”。

## 程序结构和执行

### 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0~2w-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

### 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位，或者用k mod w的结果，又或者用k%w的余数表示。

### 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

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：

也就是说，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：

无符号数转换为补码：

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

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

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

的形式，就是为了凸显补码的不确定性。

### 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

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。

### 整数运算

1. 无符号加法

**我的直觉**：减去其实就是相当于把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

当的时候表示溢出。

**推导**：当发生溢出的时候，

即

这跟事实

不符，所以当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**；

1. 补码加法；

溢出检测：当时，发生负溢出。

当时，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：

为什么这样呢？因为补码是非对称的，比少1，在正数或者负数下都可以找到自己的相反数，而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。

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。

1. 无符号乘法

将一个无符号截断为w位等价于计算该值的模。

1. 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

1. 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

所以

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。

1. 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

这样加上等同于加上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

1. 浮点数的计算

## 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。

### 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。

1. 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。
2. 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。

### 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

### 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

1. 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ra表示任意寄存器，用引用R[ra]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。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

1. 寻址方式：

|  |  |  |
| --- | --- | --- |
| **类型** | **格式** | **操作数** |
| 立即数寻址 | $Imm | Imm |
| 寄存器寻址 | ra | R[ra] |
| 绝对寻址 | Imm |  |
| 间接寻址 | (ra) |  |
| （基址+偏移量）寻址 |  |  |
| 变址寻址 |  |  |
| 变址寻址 |  |  |
| 比例变址寻址 |  |  |
| 比例变址寻址 |  |  |
| 比例变址寻址 |  |  |
| 比例变址寻址 |  |  |

1. 数据传送指令
2. 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
3. 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。

1. 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）。
2. 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。
3. ret是返回return的意思
4. movsbq：在复制值的时候把多余的高位设置成1；
5. movzbq：在复制值的时候把多余的高位设置成0；

### 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)

1. 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.

1. 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。

1. 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

1. 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

1. 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

1. 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

1. 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。

### 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，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；

### 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

1. 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

1. 编码例子：

movq $0,%rax

jmp .L1

.L1:

popq %rdx

1. 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。

1. 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:

1. 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。

1. 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。

### 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;

1. while循环
2. for循环

### 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

//

……………

### 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

1. 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；

1. 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；

1. 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。

1. 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。

### 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。

1. 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

1. 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

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

1. 定长和变长数组
2. 异质的数据结构：

结构struct

联合union

1. 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。

1. 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。

1. 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。

## 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

### 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；

1. 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。
* 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。

1. 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。

1. 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。

1. 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。
2. 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。
3. ；
4. ；
5. Y

这一章实在太难了，先战术性放弃，以后有时间再看。