# 计算机系统漫游

## 信息就是位+上下文

1、像hello.c这样只由ASCII字符构成的文件称为文本文件

2、基本思想：系统中的所有信息--包括磁盘文件、内存中的程序、内存中存放的用户数据以及网络上传送的数据，都是由一串bit表示的，区分不同数据对象的唯一方法使我们读到这些数据对象时的上下文。例如，在不同的上下文中，一个同样的字节序列可能表示一个整数、浮点数、字符串或者机器指令

3、C语言的起源

1. C语言与Unix操作系统关系密切：C从一开始就是作为一种用于Unix系统的程序语言开发出来的，大部分的Unix内核(操作系统的核心部分)，以及所有支撑工具和函数库都是用C语言编写的
2. C语言小而简单：C语言的设计师由一个人而非一个协会掌控的，因此这是一个简洁明了、没有什么冗赘的设计
3. C语言是为实践目的的设计：C语言是用来实现Unix操作系统的，后来其他人发现能够用这门语言无障碍地编写他们想要的程序

## 程序被其他程序翻译成不同的格式

1、hello程序的生命周期是从一个高级C语言程序开始的，因为这种形式能够被人读懂。为了在系统上运行hello.c程序，每条C语句都必须被其他程序转化为一系列低级机器语言指令。然后这些指令按照一种称为可执行目标程序的格式打包，并以二进制磁盘文件的形式存放起来。目标程序也成为可执行文件

2、在Unix系统上，从源文件到目标文件的转化是由编译器驱动程序完成的

gcc -o hello hello.c

* GCC编译器驱动程序读取源程序文件hello.c，并把它翻译成一个可执行目标文件hello。这个翻译过程可分为四个阶段完成，执行这四个阶段的程序(预处理器、编译器、汇编器、链接器)一起构成了编译系统

1. 预处理阶段：预处理器(cpp)根据以字符#开头的命令，修改原始的C程序。比如hello.c中的第一行的#include<stdio.h>命令告诉预处理器读取系统头文件stdio.h的内容，并把它直接插入程序文本中。结果得到了另一个C程序，通常以.i作为文件扩展名
2. 编译阶段：编译器(ccl)将文本文件hello.i翻译成文本文件hello.s，它包含一个汇编语言程序
3. 汇编阶段：汇编器(as)将hello.s翻译成机器语言指令，把这些指令打包成一种叫做可重定位目标程序(relocatable object program)的格式，并将结果保存在目标文件hello.o中。hello.o是一个二进制文件
4. 链接阶段：hello程序调用了printf函数，它是每个C编译器都提供的标准C库中的一个函数。printf函数存在于一个名为printf.o的单独预编译好了的目标文件中，而这个文件必须以某种方式合并到我们的hello.o程序中。链接器(ld)就负责处理这种合并，结果就得到hello文件，它是一个可执行目标文件，可以被加载到内存中，由系统执行

## 了解编译系统如何工作是大有益处的

1、优化程序性能

2、理解链接时出现的错误

3、避免安全漏洞

## 处理器读并解释存储在内存中的指令

1、在Unix系统上运行可执行文件，将它的文件名输入到称为shell的应用程序中

### 系统的硬件组成

1、总线：

* 贯穿整个系统的是一组电子管道，称为总线，它携带信息字节并负责在各个部件间传递
* 通常总线被设计成传送定长的字节块，也就是字(word)。字中的字节数(即字长)是一个基本的系统参数，各个系统不尽相同。现在大多数机器字长要么是4个字节(32位)，要么是8个字节(64位)

2、I/O设备：

* I/O(输入/输出)设备是系统与外部世界的联系通道
* 每个I/O设备都通过一个控制器或适配器与I/O总线相连
* 控制器与适配器之间的区别主要在于它们的封装方式
* 控制器是I/O设备本身或者系统的主印制电路板(通常称为主板)上的芯片组
* 而适配器则是一块插在主板插槽上的卡
* 无论如何，它们的功能都是在I/O总线和I/O设备之间传递信息

3、主存

* 主存是一个临时存储设备，在处理器执行程序时，用来存放程序和程序处理的数据
* 物理上来说，主存是由一组动态随机存取存储器(DRAM)芯片组成的
* 从逻辑上来说，存储器是一个线性的字节数组，每个字节都具有唯一的地址(数组索引)，这些地址是零开始的
* 一般来说，组成程序的每条机器指令都由不同数量的字节构成
* 与C程序变量相对应的数据项的大小是根据类型变化的

4、处理器

* 中央处理单元(CPU)，简称处理器，是解释(或执行)存储在主存中指令的引擎
* 处理器的核心是一个大小为一个字的存储设备(或寄存器)，称为程序计数器(PC)。在任何时候，PC都指向主存中的某条机器语言指令(即含有该条指令的地址)
* 从系统通电开始，直到系统断电，处理器一直不断地执行程序计数器指向的指令，再更新程序计数器，使其指向下一条指令。处理器看上去是按照一个非常简单的指令执行模型来操作的，这个模型是由指令集架构决定的。在这个模型中，指令按照严格的顺序执行，而执行一条指令包含执行一系列的步骤。处理器从程序计数器指向的内存处读取指令，解释指令中的位，执行该指令指示的简单操作，然后更新PC，使其指向下一条指令，而这条指令并不一定和在内存中刚刚执行的指令相邻
* 以下是CPU在指令的要求下可能执行的操作

1. 加载：从主存复制一个字节或一个字到寄存器，以覆盖寄存器原来的内容
2. 存储：从寄存器复制一个字节或一个字到主存的某个位置，以覆盖这个位置上原来的内容
3. 操作：把两个寄存器的内容复制到ALU，ALU对这两个字做算数运算，并将结果存放到一个寄存器中，以覆盖该寄存器中原来的内容
4. 跳转：从指令本身中抽取一个字，并将这个字复制到程序计数器(PC)中，以覆盖PC中原来的值

## 高速缓存至关重要

1、系统花费了大量的时间把信息从一个地方挪到另一个地方

* hello程序的机器指令最初存放在磁盘上，当程序加载时，它们被复制到主存
* 当处理器运行程序时，指令又从主存复制到处理器
* 数据串"hello,world\n"开始在磁盘上，然后被复制到主存，最后从主存上复制到显示设备

2、从程序员的角度看，这些复制就是开销，减慢了程序"真正"的工作，因此系统设计者的一个主要目标就是使这些复制操作尽可快地完成

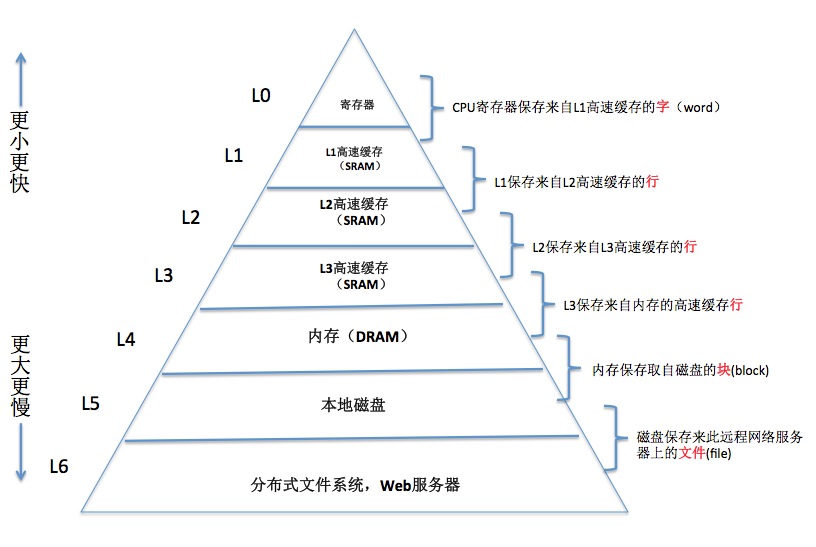
3、根据机械原理

* 较大的存储设备要比较小的存储设备运行得慢，而快速设备的造价远高于同类的低速设备。例如，一个典型系统上的磁盘驱动器可能比主存大1000倍，但是对于处理器而言，从磁盘驱动器上读取一个字的时间开销要比从主存中读取的开销大1000万倍
* 类似地，一个典型的寄存器文件只存储几百字节的信息，而主存里可存放几十亿字节，然而，处理器从寄存器文件中读取数据比从主存中读取几乎要快100倍
* 更麻烦的是，随着半导体技术的进步，这种处理器与主存之间的差距还在持续增大。加快处理器运行速度比加快主存运行速度要容易得多

4、针对这种处理器与主存之间的差异，系统设计采用了更小更快的存储设备，称为高速缓存存储器(cache memory，简称为cache或高速缓存)，作为暂时的集结区域，存放处理器近期可能会需要的信息

## 存储设备形成层次结构

1、在处理器和一个较大较慢的设备(例如主存)之间插入一个更小更快的存储设备(例如高速缓存)的想法已经成为一个普遍的观念。实际上，每个计算机系统中的存储设备都被组织成了一个存储器层次结构



2、存储器层次结构的主要思想是上一层存储器作为低一层存储器的高速缓存。因此，寄存器文件就是L1的高速缓存，L1是L2的高速缓存，L2是L3的高速缓存，L3是主存的高速缓存，而主存又是磁盘的高速缓存。在某些具有分布式文件系统的网络系统中，本地磁盘就是存储在其他系统中磁盘上的数据的高速缓存

## 操作系统管理硬件

1、shell和hello程序都没有直接访问键盘、显示器、磁盘或者主存。取而代之的是，它们依靠操作系统提供的服务。我们可以把操作系统看成是应用程序和硬件之间插入的一层软件，所有程序对硬件的操作尝试都必须通过操作系统

2、操作系统的基本功能

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

3、操作系统通过几个基本抽象概念来实现这两个功能

1. 进程
2. 虚拟内存
3. 文件

4、计算机系统分层图以及操作系统抽象表示如下图所示





* 文件是I/O设备的抽象表示
* 虚拟内存是对主存和磁盘I/O设备的抽象表示
* 进程则是对处理器、主存和I/O设备的抽象表示

### 进程

1、像hello这样的程序在现代操作系统上运行时，操作系统会提供一种假象，就像系统上只有这个程序在运行，程序看上去独占使用处理器、主存和I/O设备。处理器看上去像不间断地一条接一条地执行程序中的命令，即该程序代码和数据是系统内存中唯一的对象。这些假象是通过进程的概念来实现的，进程是计算机科学中最重要和最成功的的概念之一

2、进程是操作系统对一个正在运行的程序的一种抽象，在一个系统上可以同时运行多个进程，而每个进程都好像独占地使用硬件。而并发运行，则是说一个进程的指令和另一个进程的指令是交错执行的。在大多数系统中，需要运行的进程是多于可以运行它们的CPU个数的

3、传统系统在一个时刻只能执行一个程序，而先进的多核处理器同时能够执行多个程序。无论在单核还是多核系统中，一个CPU看上去都像是在并发地执行多个进程，这是通过处理器在进程间切换来实现的。操作系统实现这种交错执行的机制称为上下文切换

4、操作系统保持跟踪进程运行所需的所有状态信息。这种状态，就是上下文，包括许多信息，比如PC和寄存器文件的当前值，以及主存的内容。当操作系统决定要把控制权从当前进程转移到某个新进程时，就会进行上下文切换，即保存当前进程的上下文、恢复进程的上下文，然后将控制权传递到新进程，新进程就会从它上次停止的地方开始

5、从一个进程到另一个进程的转换时由操作系统内核(kernel)管理的，内核是操作系统代码常驻主存的部分。当应用程序需要操作系统的某些操作时，比如读文件，它就执行一条特殊的系统调用(system call)指令，将控制权传递给内核，然后内核执行被请求的操作并返回给应用程序。

6、注意，内核不是一个独立的进程，相反，它是系统管理全部进程所用代码和数据结构的集合

### 线程

1、通常我们认为一个进程只有单一的控制流，但在现代系统中，一个进程实际上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，并共享同样的代码和全局数据。

2、由于网络服务器中对并行处理的需求，线程成为越来越重要的编程模型，因为多线程之间比多进程之间**更容易**共享数据(进程间也能共享数据)，也因为线程一般来说比进程更高效

3、到底什么是线程和进程呢

### 虚拟内存

1、虚拟内存是一个抽象概念，它为每个进程提供了一个假象，即每个进程都在独占地使用主存，每个进程看到的内存都是一致的，称为虚拟地址空间



* 图中地址从下往上是增大的

2、逐个介绍每个内存区域

1. 程序代码和数据

* 对所有进程来说，代码是从同一固定地址开始，紧接着是和C全局变量相对应的数据位置。代码和数据区是直接按照可执行目标文件的内容初始化的

1. 堆

* 代码和数据区后紧随着是运行时堆，代码区和数据区在进程一开始运行时就被指定了大小
* 与此不同，当调用像malloc和free这样的C标准库函数时，堆可以在运行时动态地扩展和收缩

1. 共享库

* 大约在内存地址空间中间部分是一块用来存放像C标准库和数学库这样的共享库的代码和数据的区域
* 共享库的概念非常强大，也相当难懂，第七章有详细介绍

1. 栈

* 位于用户虚拟地址空间顶部的是用户栈，编译器用它来实现函数调用
* 和堆一样，用户栈在程序执行期间可以动态地扩展和收缩
* 每次调用一个函数时，栈就会增长
* 每次从一个函数返回，栈就会收缩
* 第三章将会学习到编译器如何使用栈

1. 内核虚拟内存

* 地址空间顶部的区域是为内核保留的
* 不允许应用程序读写这个区域的内容或者直接调用内核代码定义的函数，相反，它们必须调用内核来执行这些操作

### 文件

1、文件就是字节序列，仅此而已

2、每个I/O设备，包括磁盘、键盘、显示器，甚至网络，都可以看成是文件

3、系统中的所有输入输出都是通过一小组称为Unix I/O的系统函数调用读写文件来实现的

4、文件这个简单而精致的概念是非常强大的，因为它向应用程序提供了一个统一的视图，来看待系统中可能含有的所有各式各样的I/O设备

## 系统之间利用网络通信

1、实际上，现代系统经常通过网络和其他系统连接到一起，从一个单独的系统来看，网络可视为一个I/O设备，当系统从主存复制一串字节到网络适配器时，数据流经过网络到达另一台机器，而不是比如说到达本地磁盘驱动器，相似地，系统可以读取从其他机器发来的数据，并把数据复制到自己的主存

2、客户端和服务器之间交互的类型在所有的网络应用中是非常典型的

## 重要主题

### Amdahl定律

1、想要显著加速整个系统，必须提升全系统中相当大的部分的速度

### 并发和并行

1、两个术语

1. 并发：指一个同时具有多个活动的系统
2. 并行：指的是用并发来使系统运行得更快

2、并行可以在计算机系统的多个抽象层次上运用

1. 线程级并发
2. 指令级并发
3. 单指令多数据并行

### 计算机系统中抽象的重要性

1、抽象的使用时计算机科学中最为重要的概念之一。例如，为一组函数规定一个简单的应用程序接口(API)就是一个很好地编程习惯，程序员无需了解它内部的工作便可以使用这些代码

2、在处理器里的抽象

1. 指令集架构提供了对实际处理器硬件的抽象，使用这个抽象，机器代码程序表现的就好像运行在一个一次只执行一条指令的处理器上
2. 底层的硬件远比抽象描述的要复杂精细，它并行地执行多条指令，但又总是与那个简单有序的模型保持一致，只要执行模型一样，不同的处理器实现也能执行相同的机器码，而又能提供不同的开销和性能

3、操作系统里的抽象

1. 文件：对I/O设备的抽象
2. 虚拟内存：对程序存储器的抽象
3. 进程：对一个正在运行的程序的抽象
4. 虚拟机：它提供对整个计算机的抽象

# 信息的表示和处理

1、二进制信号能够容易地被表示、存储和传输。例如可以表示为穿孔卡片上的有洞或无洞、导线上的高压电和低压电，或者顺时针或逆时针的磁场

2、二进制信号进行存储和执行计算的电子电路非常简单和可靠，制造商能够在一个单独的硅片上集成数百万甚至数亿个这样的电路

3、孤立地讲，单个位不是非常有用，当把位组合在一起，再加上某种解释，即赋予不同的可能位模式以含义，我们就能够表示任何有限集合的元素

## 信息存储

1、大多数计算机使用8位的块，或者字节，作为最小的可寻址的内存单位，而不是访问内存中单独的位

2、机器级程序将内存视为一个非常大的字节数组，称为虚拟内存

3、内存的每个字节都由一个唯一的数字来标识，称为它的地址，所有可能地址的集合就成为虚拟地址空间

### 十六进制表示法

1、十六进制采用0~9，以及A~F来表示16个可能的值

### 字数据大小

1、每台计算机都由一个字长(word size)，指明指针数据的标称大小(nominal size)。因为虚拟地址是以这样的一个字来编码的，所以字长决定的最重要的系统参数就是虚拟地址空间的最大大小。对于一个字长为w位的机器而言，虚拟地址的范围为0~2w-1，即最多访问2w个字节

2、大多数64位机器可以运行32位机器编译的程序，这是一种向后兼容

3、32位程序或64位程序，区别在于该程序是如何编译的，而不是其运行的机器类型

### 寻址和字节顺序

1、对于跨多字节的程序对象，我们必须建立两个规则：这个对象地址是什么，以及在内存中如何排列这些字节(注意本小结讨论的是字节顺序，而非一个字节中的位的顺序)

2、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

3、排列表示一个对象的字节由两个通用的规则

* 考虑一个w位的整数，其为表示为[xw-1,xw-1,...,x1,x0]，其中xw-1是最高有效位，x0是最低有效位。假设w是8的倍数，这些位就能被分组成为字节，其中最高有效字节包含位[xw-1,xw-1,...,xw-8]，最低有效字节包含[x7,x6,..,x0]
* 某些机器选择在内存中按照从最低有效字节到最高有效字节的顺序存储对象，而另一些机器则按照从最高有效字节到最低有效字节的顺序存储
* 最低有效字节在最前面的方式，称为小端法(little endian)
* 最高有效字节在最前面的方式，称为大端法(big endian)

例如整型变量x，其值为0x01234567

大端法为：01 23 45 67

小端法为：67 45 23 01

* 注意，无论大端还是小端，单个字节中的顺序都是一样的，不同的是多个字节的排列顺序。因此char类型，无论是大端法还是小端法都是相同的，因为只有一个字节，无所谓字节顺序

4、大多数Intel兼容机都只用小端模式

5、对于大多数应用程序员来说，其机器所使用的的字节顺序是完全不可见的，无论为哪种类型的机器所编译的程序都会得到相同的结果

6、不过有时候，字节顺序会造成问题

* 首先是在不同类型的机器之间通过网络传送二进制数据时，一个常见的问题是当小端法机器产生的数据被发送到大端法机器或者反过来说时，接受程序会发现，字里的字成了反序的，为了避免这种问题，网络应用程序的代码编写必须遵守已建立的关于字节顺序的规则，以确保发送方机器将它的内部表示转换成网络标准，而接收方机器则将网络标准转换为它的内部表示
* 当阅读表示整数数据的字节序列时字节顺序也很重要

### 表示字符串

1、C语言中字符串被编码为一个以null(置为0，即'\0')字符结尾的字符数组，每个字符都由某个标准编码来表示，最常见的是ASCII码

2、使用ASCII码作为字符码的任何系统上都将得到相同的结果，与字节顺序与字大小规则无关(因为char就一个字节，无所谓字节顺序)，因此，文本数据比二进制数据更具有更强的平台独立性

### 表示代码

1、不同的机器类型使用不同的且不兼容的指令和编码方式，即使是完全一样的进程，运行在不同操作系统上也会有不同的编码规则。因此二进制代码是不兼容的，二进制代码很少可能在不同机器和操作系统组合之间移植

### 布尔代数简介

1、二进制值是计算机编码、存储和操作信息的核心，所以围绕数值0和1的研究已经演化出了丰富的数学知识体系

2、最简单的布尔代数是在二元集合{0，1}基础上定义的

3、位向量：位向量一个很有用的应用就是表示有限集合，我们可以用位向量[aw-1,...,a1,a0]编码任何子集A⊆{0,1,...,w-1}，其中ai=1当且仅当i∈A。aw-1写在左边，将a0写在右边，例如位向量a=[01101001]表示集合A={0,3,5,6}

### C语言中的位级运算

1、位运算

1. |
2. &
3. ^

2、神奇的inplace\_swap

void inplace\_swap(int& x ,int& y){

y=x^y;

x=x^y;//x=x^(x^y)=0^y=y

y=x^y;//y^(x^y)=y^(y^x)=0^x=x

}

### C语言中的逻辑运算

1、逻辑运算

1. ||
2. &&
3. !

### C语言中的移位运算

1、移位运算，对于一个位表示为[xw-1,xw-2,...,x0]的操作数x

1. <<：x向左移动k位，丢弃最高k位，并在右端补k个0

[xw-k-1,xw-k-2,...,x0,0,...,0]

1. >>(逻辑右移)：逻辑右移在左端补k个0

[0,...,0,xw-1,xw-2,...,xk]

1. >>(算数右移)：算数右移在左端补k个最高有效位的值

[xw-1,...,xw-1,xw-1,xw-2,...,xk]

2、C语言标准并没有明确定义对于有符号数应该使用哪种类型的右移---算数右移和逻辑右移都可以，这就意味着任何假设一种或者另一种右移形式的代码都可能遇到可移植性的问题。实际上，几乎所有的编译器/机器组合都对有符号数使用算数右移，且许多程序员也都假设机器会使用这种右移(不对啊，VS上默认是符号右移啊)

## 整数表示

1、用位来编码整数的两种不同方式：一种只能表示非负数，一种能够表示负数、零和正数

2、符号说明

* B2Tw：二进制转补码
* B2Uw：二进制转无符号数
* U2Bw：无符号数转二进制
* U2Tw：无符号数转补码
* T2Bw：补码转二进制
* T2Uw：补码转无符号数
* TMinw：最小补码值
* TMaxw：最大补码值
* UMaxw：最大无符号数
* ：补码加法
* ：无符号数加法
* ：补码乘法
* ：无符号数乘法
* ：补码取反
* ：无符号数取反

### 整型数据类型

1、C语言支持多种整型数据类型---表示有限范围的整数

2、整数类型

1. char
2. short
3. int
4. long

3、C和C++都支持无符号数，Java只支持有符号数

4、C语言只规定了整形数据类型的最小取值范围，至于正真的取值范围与机器有关，例如long就是一个与机器相关的类型，32位机器就是32位，64位机器就是64位

1. [signed]char [-127,127]
2. unsigned char [0,255]
3. short [-32767,32767]
4. unsigned short [0,65535]
5. int [-32767,32767]
6. unsigned [0,65535]
7. long [-2147483647,2147483647]
8. unsigned long [0,4294967295]
9. int32\_t [-2147483648,2147483647]
10. uint32\_t [0,4294967295]
11. int64\_t [-9223372036854775808,9223372036854775807]
12. uint64\_t [0,18446744073709551615]

### 无符号数的编码

1、假设有一个整数数据类型有w位，可以将向量写成，表示整个向量，或者写成[xw-1,xw-2,...,x0]

2、无符号数编码的定义，我们用函数B2Uw(Binary to Unsigned的缩写，长度w)来表示



3、无符号编码满足唯一性：函数B2Uw是一个双射

### 补码编码

1、对于许多应用，我们还希望表示负数值，最常见的有符号数的计算机表示方式就是补码(two's-complement)，在这个定义中，将字的最高有效位解释为负权(negative weight)，我们用B2Tw(Binary to Two's-complement的缩写，长度为w)来表示

2、补码的定义



3、补码所能表示的范围

1. 最小值：设置负权位，清除其他所有位，TMinw=-2w-1
2. 最大值：清除负权位，设置其他所有位，TMaxw=

4、补码编码满足唯一性：函数B2Tw是一个双射

5、有符号数的其他表示方法

1. 反码：
2. 原码：

* 这两种表示方式都有一个奇怪的属性，那就是对于数字0有两种不同的编码方式
* 几乎所有现代机器都采用补码来表示有符号数

6、注意补码(Two's complement)和反码(Ones' complement)中撇号的位置是不同的

* 术语补码来源于这样一个情况，对于非负数x，我们用2w-x(这里只有一个2)来计算-x的w位表示
* 术语反码来源于这样一个情况，我们用[111...1]-x(这里有很多个1)来计算-x的反码表示

### 有符号数和无符号数之间的转换

1、C语言允许在各种不同的数字数据类型之间做强制类型转换

2、**强制类型转换的结果保持位值不变，只是改变了解释这些位的方式**

3、对于大多数C语言的实现，处理同样字长的有符号数和无符号数之间相互转换的一般规则是：数值可能会改变，但是位模式不变。为了用更数学的形式来描述这个规则，我们定义函数U2Bw和T2Bw，它们将数值映射为无符号数和补码形式的表示

* 给定0≤x≤UMaxw，函数U2Bw(x)会给出x的唯一w位无符号表示
* 给定TMinw≤x≤TMaxw，函数T2Bw(x)会给出x的唯一w位补码表示

4、现在将函数T2Uw定义为T2Uw=B2Uw(T2Bw(x))

5、补码转换为无符号数：对满足TMinw≤x≤TMaxw的x有



6、无符号数转为补码：对于满足0≤u≤UMaxx的u



### C语言中的有符号数和无符号数

1、C语言支持所有整型数据类型的有符号和无符号运算。尽管C语言标准并没有指定有符号数要采用某种表示，但是几乎所有的机器都使用补码

2、通常，要创建一个无符号常量，必须加上后缀字符'U'或者'u'，例如

12345U或者0x1A2Bu

3、当一种类型的表达式被赋值给另一种类型的变量时，转换是隐式发生的(转换被允许的话，对于基本类型由编译器决定，对于类类型由单个参数且非explicit的构造函数决定)

4、C语言对同时包含有符号和无符号数表达式，有奇特的行为

1. 当执行一个运算时，如果它的一个运算数是有符号的而另一个时无符号的，C语言会隐式地将有符号参数强制转换成无符号数，来执行这个运算

5、注意TMin的写法

1. 我们将TMin32写成-2147483647-1，而不是写成-2147483648或0x80000000

* 0x80000000大于INT\_MAX，int装不下，因此会被当成unsigned int
* 这是因为int的最小值是-2147483648，最大值是2147483647，但是我们不能用int n = -2147483648。因为编译器(VS2015)在看到int n = -2147483648;的时候，首先判断2147483648 > INT\_MAX，知道int装不下，于是决定使用unsigned int。然后发现前面还有个负号，但是编译器会拒绝无符号数的取反运算，因此错误

### 扩展一个数字的位表示

1、要将一个无符号数转换成一个更大数据类型时，我们只需要简单地在表示的开头添加0，这种运算被称为零扩展(zero extension)

[uw-1,uw-2,...,u0]--->[0,...,0,uw-1,uw-2,...,u0]

2、要将一个补码数字转换成一个更大的数据类型，可以执行一个符号扩展(sign extension)，在表示中添加最高有效位的值

[xw-1,xw-2,...,x0]--->[xw-1,...,xw-1,xw-1,xw-2,...,x0]

* 这样做是为了抵消由于符号位权值变大带来的增益

### 截断数字

1、当将一个w位的数截断为一个k位数字时，我们会丢弃高w-k位，截断一个数字可能会改变它的值，这是溢出的一种形式

2、无论是有符号数和无符号数

[xw-1,xw-2,...,x0]---> [xk-1,xk-2,...,x0]

### 关于有符号数和无符号数的建议

1、有符号数到无符号数的隐式强制类型转换导致了某些非直观的行为

2、避免这类错误的一种方法就是绝不使用无符号数。实际上除了C语言外，很少有语言支持无符号数，因为设计者认为他们带来的麻烦要比利益多得多。例如Java只支持有符号整数，并且要求以补码运算来实现。正常的右移运算符>>被定义为执行算数右移。特殊的运算符>>>被定义为执行逻辑右移

3、当我们想要把字仅仅看做是位的集合而没有任何数字意义时，无符号数值是非常有用的

## 整数运算

### 无符号数加法

1、无符号数加法，两个w位的无符号整数相加，其和是w+1位的整数



* 当多出的最高位是0，丢弃没有任何影响
* 当多出的最高位是1，丢弃会造成影响

2、当执行C程序时，不会将溢出作为错误而发出信号

3、检测无符号数加法中的溢出

在对范围0≤x，y≤UMaxw中的x和y，令，则对计算s，当且仅当s<x(或者等价地s<y)时，发生了溢出

### 补码的加法

1、对于补码加法，我们必须确定当结果太大(为正)或者太小(为负)

2、对于在给定范围-2w-1≤x,y≤2w-1-1之内的整数值x和y，它们的和就在范围  
-2w≤x+y≤2w-2之内，要想准确表示，可能需要w+1位

3、定义为整数和x+y被截断为w位的结果，并将这个结果看做是补码数

对于满足-2w-1≤x,y≤2w-1-1的整数x和y



* 正溢出由于xw-2位的和向xw-1产生了进位，导致符号位从0变成1
* 负溢出由于符号位xw-1向更高位产生了进位(被丢弃)，导致符号位从1变成0

### 补码的非

1、对于满足TMinw≤x≤TMaxw的x，其补码的非



* TMinw+TMinw=-2w-1+(-2w-1)=-2w，这将导致负溢出结果为0，因此TMinw的求反就是其本身

2、补码非聪明方法

-x=~x+1

### 无符号数乘法

1、对于在给定范围0≤x,y≤2w-1之内的整数值x和y，它们的乘积就在范围0到(2w-1)2=22w-2w-1+1之间，这就需要2w位来表示。但是C语言中的无符号数乘法被定义为产生w位的值，就是2w位的整数乘积的低w位表示的值

2、对于满足0≤x,y≤UMaxw的x和y



### 补码乘法

1、对于在给定范围-2w-1≤x,y≤2w-1-1之内的整数值x和y，它们的乘积就在范围  
-2w-1\*(2w-1-1)=-22w-2+2w-1到-2w-1\*-2w-1=-22w-2。想要用补码来表示这个乘积，就需要2w位。在C语言中有符号数的乘法是通过2w位的乘积截断为w位来实现的

2、未完成

### 乘以常数

1、在大多数机器上，整数乘法指令相当慢，需要10个或更多时钟周期，而其他整数运算(例如加法、减法、位级运算和移位)只需要一个时钟周期，即便在i7上，整数乘法也需要3个时钟周期

2、因此，编译器采用了一项重要优化，试着用位移和加法运算的组合来代替乘以常数因子的乘法

3、乘以2的幂：左移k位(无论是无符号数还是有符号数，对于溢出情况也能得到一致的结果，因为对于溢出的位都是采用截断处理)

4、乘以任意整数的幂

* 首先将整数分解成2的幂次求和，就是那些位为1的幂之和
* 然后在进行移位运算

### 除以2的幂

1、在大多数机器上，整数除法比整数乘法更慢---需要30个或者更多的时钟周期

2、除以2的幂也可以用位移运算来实现，只不过用的是右移，而不是左移

3、无符号数用逻辑移位来进行处理，有符号数用算数移位来处理

## 浮点数

### 二进制小数

1、理解浮点数的第一步是考虑含有小数值的二进制数字



2、我们仅考虑有限长度的编码，那么大部分十进制并不能准确表达，只能近似的表达，增加二进制表示的长度可以提高表示的精度

### IEEE浮点表示

1、我们希望通过给定x和y的值，来表示形如的数

2、IEEE浮点标准用

1. 符号(sign)：s决定这个数是负数(s=1)还是正数(s=0)，而对于数值0的符号位解释为作为特殊情况处理
2. 尾数(significand)：M是一个二进制小数，它的范围，或者是
3. 阶码(exponent)：E的作用是对浮点数加权，这个权重是2的E次幂(可能是负数)

3、将浮点数表示划分为三个阶段，分别对这些值进行编码

1. 一个单独的符号位s直接编码符号s
2. k位的阶码字段exp=ek-1...e1e0编码阶码E
3. n位小数字段frac=fn-1...f1f0编码尾数M，但是编码出来的值也依赖于阶码字段的值是否等于0

4、单精度和双精度如下



5、给定位表示，根据exp的值，被编码的值可以分为三种不同情况(最后一种情况有两个变种)



#### 情况1：规格化的值

1、当exp的位模式既不是全0(数值0)也不是全1(单精度255，双精度2047)

2、在这种情况下，阶码字段被解释为以偏置(biased)形式表示的有符号整数。阶码的值时E=e-Bias，其中e是假定阶码字段是一个无符号整数时所表示的值，其位表示为ek-1...e1e0，而Bias是一个等于2k-1-1(单精度是127，双精度是1023)的偏置值，由此产生指数的取值范围

* 对于单精度是-126~127
* 对于双精度是-1022~1023

3、小数字段frac被解释为描述小数值f，其中0≤f<1，其二进制表示为0.fn-1...f1f0，也就是二进制小数点在最高有效位的左边。尾数定义为M=1+f

* 有时，这种方式也叫作隐含的以1开头的(implied leading 1)表示，因为我们可以把M看成一个二进制表达式为1.fn-1...f1f0的数字
* 既然我们总是能够调整阶码E，使得尾数M在范围1≤M<2之中，那么这种表示方法是一种轻松获得一个额外精度位的技巧。既然第一位总是等于1，那我们就不需要显式地表示它

#### 情况2：非规格化的值

1、当阶码域全为0时，所表示的数是非规格化形式。在这种情况下，阶码值时E=1-Bias，而尾数的值时M=f，也就是小数字段的值，不包含隐含的开头1

2、为什么要这样设置偏置值

* 使阶码值为1-Bias而不是简单的-Bias似乎是违反直觉的(因为按照规格化值得定义，E=e-Bias=0-Bias=-Bias)
* 这种方式提供了一种从非规格化值平滑转换到规格化值得方法(详见P80)

3、非规格化数有两个用途

* 首先，它们提供了一种表示数值0的方法，因为使用规格化数，我们必须总是使M≥1，因此我们就不能表示0
* +0.0的浮点表示的位模式为全0：符号位是0，阶码字端全为0，小数域全为0，这就得到了M=f=0
* -0.0的浮点表示的位模式为：符号位1，阶码字段全0，小数域全0
* 根据IEEE的浮点格式，值+0.0和-0.0在某些方面是不同的，而在其他方面是相同的
* 非规格化数的另外一个功能是表示那些非常接近于0.0的数，它们提供了一种属性，称为逐渐溢出(gradual underflow)

#### 情况3：特殊值

1、当阶码全为1，小数域全为0时，得到的值表示无穷大

* 当s=0表示正无穷大
* 当s=1表示负无穷大

2、当阶码全为1，小数域不为0时，得到的值称为"NaN"，即不是一个数(Not a number)的缩写，一些运算的结果不能是实数或无穷，就会返回这样的NaN值

### 数字示例

1、未完成

### 舍入

1、因为表示方法限制了浮点数的范围和精度，所以浮点运算只能近似地表示实数运算

2、因此，对于值x，我们一般想用一种系统的方法，能够找到"最接近的"匹配值x'，它可以用期望的浮点数形式表示出来。这就是舍入(rounding)运算的任务

3、一个关键的问题是在两个可能值得中间确定舍入方向。例如我有1.50美元，想把它舍入到最接近的美元数，应该是1还是2呢。一种可选的方法是维持实际数字的上届和下届。例如，我们可以确定可表示的值x-和x+，使得x的值位于它们之间：x-≤x≤x+

4、四种舍入方式

1. 向偶数舍入：也被称为向最接近的值舍入
2. 向零舍入
3. 向上舍入
4. 向下舍入

5、向偶数舍入初看上去好像是个相当随意的目标，但是向偶数舍入可以避免统计偏差

* 例如，如果采用向上舍入，会在计算平均值数中引入统计偏差，得到的平均值会略大一些
* 而向偶数舍入在50%的时间里，向上舍入，在50%的时间里，向下舍入

### 浮点运算

1、IEEE标准制定了一个简单的规则，来确定诸如加法和乘法这样的算数运算的结果。把浮点值x和y看成实值，而某个运算定义在实数上，计算将产生Round(xy)，这是对实际运算的精确结果进行舍入后的结果

2、IEEE标准中制定浮点运算行为方法的一个优势在于，它可以独立于任何具体的硬件实现或软件实现。因此，我们可以检查他的抽象数学属性，而不必考虑它实际上是如何实现的

3、浮点加法满足了单调性属性，如果a≥b，那么对于任何a、b以及x值，除了NaN，满足x+a≥x+b

4、浮点乘法遵循通常乘法的许多属性，我们定义

1. 这个运算在乘法中是封闭的(虽然可能产生无穷大或者NaN)，它是可交换的，而且它的乘法单元为1.0
2. 由于可能发生溢出，或者由于舍入而失去精度，它不具有可结合性。例如(1e20\*1e20)\*1e-20=+∞，而1e20(1e20\*1e-20)将得出1e20
3. 浮点乘法在加法上不具有分配性。例如1e20\*(1e20-1e20)求值为0.0，而1e20\*1e20-1e20\*1e20会得出NaN
4. 另一方面，对于任何a、b和c，并且a、b和c都不等于NaN，浮点乘法满足下列单调性



1. 只要a不为NaN，就有，而无符号数或补码的乘法没有这些单调性属性



5、未完成

### C语言中的浮点数

1、所有C语言版本提供了两种不同的浮点数据类型：float和double

# 程序的机器表示

## 历史观点

## 程序编码

1、编译例子

gcc -Og -o p p1.c p2.c

1. 编译选项Og告诉编译器使用会生成符合原始C代码整体结构的机器代码的优化等级。使用较高级别的优化产生的代码会严重变形，以至于产生的机器代码和初始源码之间的关系非常难以理解

### 机器级代码

1、计算机系统使用了许多不同形式的抽象，利用更简单的抽象模型来隐藏实现的细节

2、对于机器级编程来说，两种抽象尤为重要

* 第一种抽象：指令集体系结构或指令集架构(Instruction Set Architecture, ISA)来定义机器级程序的格式和行为，它定义了处理器状态、指令格式，以及每条指令对状态的影响
* 第二种抽象：机器级程序使用的内存地址是虚拟地址，提供的内存模型看上去是一个非常大的字节数组。存储系统的实际实现是将多个硬件存储器和操作系统软件组合起来

3、在整个编译过程中，编译器会完成大部分的工作，将把用C语言提供的相对比较抽象的执行模型表示的程序转化成处理器执行的非常基本的命令

4、汇编代码表示非常接近于机器代码，与机器代码的二进制格式相比，汇编代码主要特点是它用可读性更好的文本格式表示。能够理解汇编代码以及它与原始C代码的联系，是理解计算机如何执行程序的关键一步

5、一些通常对C语言程序员隐藏的处理器状态

1. 程序计数器：通常称为"PC"，在x86-64中用%rip表示，给出将要执行的下一条指令在内存中的地址
2. 整数寄存器文件：包含16个命名的位置，分别存储64位的值，这些寄存器可以存储地址(对应于C语言的指针)或整数数据。有的寄存器被用来记录某些重要的程序状态，而其他的寄存器用来保存临时数据，例如过程的参数和局部变量，以及函数的返回值
3. 条件码寄存器：保存着最近执行的算术或逻辑指令的状态信息。它们用来实现控制或数据流中的条件变化，比如说用来实现if和while语句
4. 向量寄存器：一组向量寄存器可以存放一个或多个整数或浮点数值

6、虽然C语言提供了一种模型，可以在内存中声明和分配各种数据类型的对象，但是机器代码只是简单地将内存看成一个很大的、按字节寻址的数组

1. 数组和结构，在机器代码中用一组连续的字节来表示
2. 对于标量数据，汇编代码也不区分有符号和无符号整数，不区分各种类型的指针，甚至不区分指针和整数

7、程序内存包含

1. 程序的可执行机器代码
2. 操作系统需要的一些信息，用来管理过程调用和返回的运行时栈，以及用户分配的内存块(比如malloc库函数分配的)
3. 程序内部用虚拟地址来寻址，在任意给定的时刻，只有有限的一部分虚拟地址被认为是合法的。例如x86-64的虚拟地址由64位的字来表示，在目前的实现中，这些地址的高16位必须设置为0，所以一个地址实际上能够指定的是248即64TB范围内的一个字节。较为典型的程序只会访问几兆字节或者几千兆字节的数据。操作系统负责管理虚拟地址空间，将虚拟地址翻译成实际处理器内存中的物理地址

8、一条指令只执行一个非常基本的操作，例如将放在寄存器中的两个数字相加，在存储器和寄存器之间传送数据，或是条件分支转移到新的指令地址。编译器必须产生这些指令序列，从而实现(像算术表达式求值、循环或过程调用和返回这样的)程序结构

### 代码示例

1、命令

gcc -Og -S mstore.c

2、反汇编器(disassembler)：这些程序根据机器代码产生一种类似于汇编代码的格式

objdump -d mstore.o

3、一些关于机器代码和它的反汇编表示的特性值

1. x86-64的指令长度从1到15个字节不等。常用的指令以及操作数较少的指令所需的字节数较少，那些不太常用或操作数较多的指令所需字节较多
2. 设计指令格式的方式是，从某个给定位置开始，可以将字节唯一地解码城机器指令，例如只有pushq %rbx是以字节值53开头的
3. 反汇编器只是基于及其代码文件中的字节序列来确定汇编代码，他不需要访问该程序的源代码或汇编代码
4. 反汇编器使用的指令命名规则与GCC生成的汇编代码使用的有些细微的差别

4、关于格式的注解

1. GCC产生的汇编代码对我们来说有点难度，它包含了一些我们不需要关心的信息，同时它不提供任何程序的描述或它是如何工作的描述
2. 所有以'.'开头的行都是指导汇编器和链接器工作的伪指令，通常可以忽略

5、ATT和Intel汇编代码格式

1. 我们表述的是ATT(根据"AT&T"命名的，AT&T是运营贝尔实验室多年的公司)格式的汇编代码，这是GCC、OBJDUMP和其他一些我们使用的工具的默认格式
2. 其他一些编程工具，包括Microsoft工具，以及来自Intel的文档，其汇编代码都是Intel格式的
3. Gcc可以产生Intel格式的代码

gcc -Og -S -masm=intel mstore.c

1. Intel和ATT格式的不同之处

* Intel代码省略了指示大小的后缀，例如指令push和mov，而不是pushq和movq
* Intel代码省略了寄存器名字前面的'%'符号，用的是rbx，而不是%rbx
* Intel代码用不同的方式来描述内存中的位置，例如是'QWORD PTR [rbx]'而不是'(%rbx)'
* 在带有多个操作数的指令下，列出操作数的顺序相反，这一点非常令人困惑

6、在C语言中插入汇编代码有两种方法

1. 我们可以编写完整的函数，放进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
2. 我们可以使用GCC的内联汇编(inline assembly)特性，用asm伪指令可以在C语言程序中包含简短的汇编代码

## 数据格式

1、由于从16位体系结构扩展成32位，Intel用术语"字(word)"表示16位数据类型(网上的说法好像是字长是可变的，比如32位的字长就是32位)。因此32位数称为"双字(double words)"，称64位数为"四字(quad words)"

2、C语言数据类型在x86-64中的大小，在64位机器中，指针8个字节

|  |  |  |  |
| --- | --- | --- | --- |
| C声明 | Intel数据类型 | 汇编代码后缀 | 大小 |
| char | 字节 | b | 1 |
| short | 字 | w | 2 |
| int | 双字 | l | 4 |
| long | 四字 | q | 8 |
| char\* | 四字 | q | 8 |
| float | 单精度 | s | 4 |
| double | 双精度 | l | 8 |

3、大多数GCC生成的汇编代码指令都有一个字符后缀，表明操作数的大小。例如数据传送指令有四个变种

1. movb：传送字节(byte)
2. movw：传送字(word)
3. movl：传送双字(long word)
4. movq：传送四字(quad words)

## 访问信息

1、一个x86-64的中央处理单元(CPU)包含一组16个存储位值的通用目的寄存器，这些寄存器用来存储整数数据和指针

1. 这16个寄存器名字都以%r开头，后面还跟着一些不同的命名规则的名字，这是由于指令集历史演化造成的
2. 最初8086中有8个16位的寄存器，即%ax-%bp，每个寄存器都有特殊的用途
3. 扩展到IA32架构时，这些寄存器也扩展成32位寄存器，标号从%aex到%ebp
4. 扩展到x86-64后，原来的8个寄存器扩展成64位，标号从%rax-%rbp，除此之外，还增加了8个新的寄存器，他们的标号是按照新的命名规则制定的，从%r8到%r15



2、如上的嵌套结构表明，指令可以对这16个寄存器的低位字节中存放的不同大小的数据进行操作。字节级操作可以访问最低的字节，16位操作可以访问最低的2个字节，32位操作可以访问最低的4个字节，而64位操作可以访问整个寄存器

3、两条规则

1. 生成1字节和2字节数字的指令会保持剩下的字节不变
2. 生成4字节数字的指令会把高位的4个字节置为0
3. 后面这条规则是作为从IA32到x86-64的扩展的一部分而采用的

4、在常见的程序里，不同的寄存器扮演者不同的角色，其中最特别的是%rsp，用来指明运行时栈的结束位置。有些程序会明确地读写这个寄存器。值得注意的是，有一组标准的编程规范控制者如何使用寄存器来管理栈、传递函数参数、从函数的返回值，以及存储局部和临时数据

### 操作数指示符

1、大多数指令有一个或多个操作数(operand)，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

2、x86-64支持多种操作数格式，源数据值可以以常数形式给出，或是从寄存器或内存中读出。结果可以放在寄存器或内存中

3、各种不同的操作数的可能性被分为三种类型

1. 第一种类型是立即数(immediate)，用来表示常数值。在ATT格式的汇编代码中，立即数的书写方式是$后跟一个用标准C表示法表示的整数。例如$-557，$0x1F。不同的指令允许的立即数值范围不同，汇编器会自动选择最紧凑的方式进行数值编码
2. 第二种类型是寄存器(register)，它表示某个寄存器的内容，16个寄存器的低位1字节、2字节、4字节或8字节中的一个座位操作数，这些字节数分别对应于8位，16位，32位，64位。我们用符号ra表示任意寄存器a，用引用R[ra]来表示它的值，这是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
3. 第三类操作数是内存引用，它会根据计算出来的地址(通常称为有效地址)访问某个内存位置。因为将内存看成一个很大的字节数组，我们用符号Mb[Addr]表示对存储在内存中从地址Addr开始的b个字节值的的引用。为了方便，我们通常省去下标b

4、有多种不同的寻址模式，允许不同形式的内存引用

* 表中底部用语法*Imm*(rb,ri,s)表示的是最常用的形式。这样的引用有四个组成部分：一个立即数偏移*Imm*，一个基址寄存器rb，一个变址寄存器ri和一个比例因子s，这里s必须是1、2、4或者8，基址和变址寄存器都必须是64位寄存器。有效地址被计算为*Imm*+R[rb]+R[ri]\*s

|  |  |  |  |
| --- | --- | --- | --- |
| 类型 | 格式 | 操作数值 | 名称 |
| 立即数 | $*Imm* | *Imm* | 立即数寻址 |
| 寄存器 | ra | R[ra] | 寄存器寻址 |
| 存储器 | *Imm* | M[*Imm*] | 绝对寻址 |
| 存储器 | (ra) | M[R[ra]] | 间接寻址 |
| 存储器 | *Imm*(rb) | M[*Imm*+R[rb]] | (基址+偏移量)寻址 |
| 存储器 | (rb,ri) | M[R[rb]+R[ri]] | 变址寻址 |
| 存储器 | *Imm*(rb,ri) | M[*Imm*+R[rb]+R[ri]] | 变址寻址 |
| 存储器 | (,ri,s) | M[R[ri]\*s] | 比例变址寻址 |
| 存储器 | *Imm*(,ri,s) | M[*Imm*+R[ri]\*s] | 比例变址寻址 |
| 存储器 | (rb,ri,s) | M[R[rb]+R[ri]\*s] | 比例变址寻址 |
| 存储器 | *Imm*(rb,ri,s) | M[*Imm*+R[rb]+R[ri]\*s] | 比例变址寻址 |

### 数据传送指令

1、最频繁使用的指令是将数据从一个位置复制到另一个位置的指令。操作数表示的通用性使得一条简单的数据传送指令能够完成在许多机器中要好几条不同指令才能完成的功能

#### MOV类

1、下标是最简单形式的数据传输指令---MOV类

* 这些指令把数据从源位置复制到目的位置，不做任何变化
* MOV类主要由四条指令组成：movb、movw、movl和movq，这些指令都执行相同的操作，主要区别在于它们操作的数据大小不同，分别是1、2、4、8字节

|  |  |  |
| --- | --- | --- |
| 指令 | 效果 | 描述 |
| MOV S,D | D<--S | 传送 |
| movb | R<--I | 传送字节 |
| movw | 传送字 |
| movl | 传送双字 |
| movq | 传送四字 |
| movabsq I,R | 传送绝对的四字 |

2、规则

1. 源操作数指定的是一个立即数，存储在寄存器中或者内存中
2. 目的操作数指定一个位置，要么是一个寄存器，要么是一个内存地址
3. x86-64增加了一条限制，传送指令的两个操作数不能都指向内存位置，将一个值从一个内存位置复制到另一个内存位置需要两条指令

* 第一条指令将原值加载到寄存器中
* 第二条指令将该寄存器值写入目的的位置

1. 这些指令的寄存器操作数可以是16个寄存器有标号部分的任意一个，寄存器部分的大小必须与指令最后一个字符('b','w','l','q')指定的大小匹配
2. 大多数情况中，MOV指令只会更新目的操作数指定的那些寄存器字节或内存位置。唯一的例外是movl指令以寄存器作为目的时，它会把该寄存器的最高4位字节设置为0，造成这个例外的原因是x86-64的惯例，即任何为寄存器生成32位值的指令都会把该寄存器的高位部分置0

3、示例

movl $0x4050,%eax Immediate-->Register,4bytes

movw %bp,%sp Register-->Register,2bytes

movb (%rdi,%rcx),%al Memory-->Register,1byte

movb $-17,(%rsp) Immediate-->Memory,1byte

movq %rax,-12(%rbp) Register-->Memory,8bytes

4、常规的movq指令只能以表示为32位补码数字的立即数作为源操作数，然后把这个值符号扩展得到64位的值，放到目的位置。movabsp指令能够以任意64位立即数值作为源操作数，并且只能以寄存器作为目的

#### MOVZ类

1、MOVZ类指令把数据从源(寄存器或内存)复制到目的寄存器

2、MOVZ类中的指令把目的中剩余的字节填充为0

3、每条指令名字最后两个字符都是大小指示符：

1. 第一个字符指定源的大小
2. 第二个字符指定目的大小
3. 满足源大小<目的大小

|  |  |  |
| --- | --- | --- |
| 指令 | 效果 | 描述 |
| MOVZ S,R | R<--零扩展(S) | 以零扩展进行传送 |
| movzbw |  | 将做了零扩展的字节传送到字 |
| movzbl | 将做了零扩展的字节传送到双字 |
| movzwl | 将做了零扩展的字传送到双字 |
| movzbq | 将做了零扩展的字节传送到四字 |
| movzwq | 将做了零扩展的字传送到四字 |

4、注意，只有两种较小的源类型(字节，字)有零扩展传送，但是双字并没有零扩展传送，即并没有movzlq，但是将做了零扩展的双字传送到四字可以通过movl指令来实现(特殊规则)

#### MOVS类

1、MOVS类指令把数据从源(寄存器或内存)复制到目的寄存器

2、MOVS类中的指令把目的中剩余的字节通过符号扩展来填充，把源操作的最高位进行复制

3、每条指令名字最后两个字符都是大小指示符：

1. 第一个字符指定源的大小
2. 第二个字符指定目的大小
3. 满足源大小<目的大小

|  |  |  |
| --- | --- | --- |
| 指令 | 效果 | 描述 |
| MOVS S,R | R<--符号扩展(S) | 以零扩展进行传送 |
| movsbw |  | 将做了符号扩展的字节传送到字 |
| movsbl | 将做了符号扩展的字节传送到双字 |
| movswl | 将做了符号扩展的字传送到双字 |
| movsbq | 将做了符号扩展的字节传送到四字 |
| movswq | 将做了符号扩展的字传送到四字 |
| movslq | 将做了符号扩展的双字传送到四字 |
| cltq | %rax<--符号扩展(%eax) | 把%eax符号扩展到%rax |

4、所有三种源类型(字节，字，双字)都有对应的符号扩展传送

5、cltq指令与movslq %eax,%rax完全一致，不过编码更紧凑

### 数据传送示例

1、C代码

long exchange(long \*xp,long y){

long x=\*xp;

\*xp=y;

return x;

}

*long exchange(long \*xp, long y)*

*xp in %rdi, y in %rsi*

exchange:

movq (%rdi),%rax *Get x at xp，Set as return value*

movq %rsi,(%rdi) *Store y at xp*

ret *Return*

2、注意点

1. C语言中的指针其实就是地址，间接引用指针就是将该指针放在一个寄存器中，然后在内存引用中使用这个寄存器
2. 像x这样的局部变量通常保存在寄存器中，而不是内存中

### 压入和弹出栈数据

1、最后两个数据传送操作可以将数据压入程序栈中，以及从程序栈中弹出数据

2、栈在处理过程中起到至关重要的作用

* 栈是一种数据结构，可以添加或删除值，不过要遵循"先进后出"的原则。通过push操作把数据压入栈中，通过pop操作删除数据
* 弹出的值永远是最近被压入而且仍然在栈中的值
* 栈可以实现为一个数组，总是从数组的一端插入和删除元素，这一端被称为栈顶

3、pushq指令的功能是把数据压入到栈上，而popq指令是弹出数据。这些指令只有一个操作数---压入的数据源和弹出的数据数目

4、将一个四字值压入栈中，首先要将栈指针减8，然后将值写到新的栈顶地址。因此pushq %rbp的行为等价于下面两条指令

subq $8,%rsp

movq %rbp,(%rsp)

5、弹出一个四字操作包括从栈顶位置读出数据，然后将栈指针加8。因此，指令popq %rax等价于下面两条指令

movq (%rsp),%rax

addq %8,%rsp

6、因为栈和程序代码以及其他形式的程序数据都是放在同一内存中，所以程序可以用标准的内存寻址方法访问栈内的任意位置。假设栈顶元素是四字，指令movq 8(%rsp),%rdx会将第二个四字从栈中复制到寄存器%rdx

## 算数和逻辑操作

1、下表列出了x86-64的一些整数和逻辑操作。大多数操作都分成了指令类，即这些指令类有各种带不同大小操作数的变种(只有leaq没有其他大小的变种)。指令类ADD由四条加法指令组成：addb、addw、addl和addq，分别是字节加法、字加法、双字加法和四字加法

|  |  |  |
| --- | --- | --- |
| 指令 | 效果 | 描述 |
| leaq S,D | D<--&S | 加载有效地址 |
| INC D  DEC D  NEG D  NOT D | D<--D+1  D<--D-1  D<-- -D  D<-- ~D | 加1  减1  取负  取补 |
| ADD S,D  SUB S,D  IMUL S,D  XOR S,D  OR S,D  AND S,D | D<-- D+S  D<-- D-S  D<-- D\*S  D<-- D^S  D<--D|S  D<-- D&S | 加  减  乘  异或  或  与 |
| SAL k,D  SHL k,D  SAR k,D  SHR k,D | D<--D<<k  D<--D<<k  D<--D>>Ak  D<--D>>Lk | 左移  左移(等同于SAL)  算数右移  逻辑右移 |

### 加载有效地址

1、加载有效地址(load effective address)指令leaq实际上是movq指令的变形

* 它的指令形式是从内存读取数据到寄存器，但实际上它根本就没有引用内存
* 它的第一个操作数看上去是一个内存引用，但该指令并不是从指定的位置读取数据，而是将有效地址写入到目的操作数。例如如果寄存器%rdx的值为x，那么指令leaq 7(%rdx,%rdx,4),%rax将设置寄存器%rax的值为5x+7

2、leaq的灵活用法：leaq指令能执行加法和有限形式的乘法，在编译如下算数表达式时，很有用处

long scale(long x,long y,long z){

long t=x+4\*y+12\*z;

return t;

}

*long scale(long x,long y,long z)*

*x in %rdi, y in %rsi, z in %rdx*

scale:

leaq (%rdi,%rsi,4),%rax *x+4\*y*

leaq (%rdx,%rdx,2),%rdx *z+2\*z=3z*

leaq (%rax,%rdx,4),%rax *(x+4\*y)+4\*(3\*z)=x+4\*y+12\*z*

ret

### 一元和二元操作

1、第二组中的操作是一元操作，只有一个操作数，既是源又是目的。这个操作数可以是一个寄存器，也可以是一个内存位置。例如指令incq(%rsp)会使栈顶的8字节元素加1，这种语法让人想起C语言中的自增运算符和自减运算符

2、第三组是二元操作，其中第二个操作数既是源又是目的，这种语法让人想起C语言中的符合赋值运算符(+=、-=、\*=、/=、^=、|=、&=)

* 源操作数是第一个，目的操作数是第二个
* 例如指令subq %rax,%rdx使寄存器%rdx的值减去%rax中的值
* 第一个操作数可以使立即数，寄存器或是内存位置
* 第二个操作数可以是寄存器或内存位置。当第二个操作数为内存地址时，处理器必须从内存读出值，执行操作，再把结果写回内存

### 移位操作

1、最后一组是移位操作，先给出移位量，然后第二项给出的是要移位的数，可以进行算数和逻辑右移

2、移位量可以是一个立即数，或者放在单字节寄存器%cl中(这些指令很特别，因为只允许以这个特定的寄存器作为操作数)

* 原则上来说，1个字节的位移量使得移位量的编码范围可以达到28-1=255
* x86-64中，移位操作对w位长的数据进行操作，移位量是由%cl寄存器的低m位决定的，满足2m=w，高位会被忽略。例如当寄存器%cl的十六进制值为0xFF时，指令salb移动7位(23=8，111=7)，salw会移动15位(24=16，1111=15)，sall会移动31位(25=32，11111=31)，salq会移动63位(26=64，111111=63)

3、左移指令有两个名字：SAL和SHL。两者效果一样，都是将右边填上0

4、右移指令不同，SAL执行算数移位(填上符号位)，而SHR执行逻辑移位(填上0)。移位操作的目的操作数可以是一个寄存器或是一个内存位置

### 讨论

1、对于大多数指令，既可以用于无符号运算，也可以用于补码运算，只有右移操作需要区分有符号和无符号数，这个特性使补码运算成为实现有符号整数运算的一种比较好的方法的原因之一

### 特殊的算数操作

1、两个64位有符号或无符号整数相乘得到的乘积需要128位来表示。x86-64指令集对128位(16字节)数的操作提供有限的支持。延续字(2字节)、双字(4字节)、四字(8字节)的命名管理，Intel把16字节的数称为八字(oct word)

2、下表描述的是支持产生两个64位数字的全128位乘积以及整数除法指令

|  |  |  |
| --- | --- | --- |
| 指令 | 效果 | 描述 |
| imulq S  mulq S | R[%rdx]：R[%rax]<--SxR[%rax]  R[%rdx]：R[%rax]<--SxR[%rax] | 有符号全乘法  无符号全乘法 |
| clto | R[%rdx]：R[%rax]<--符号扩展(R[%rax]) | 转换为八字 |
| idivq S | R[%rdx]<--R[%rdx]：R[%rax] mod S  R[%rdx]<--R[%rdx]：R[%rax] ÷ S | 有符号除法 |
| divq S | R[%rdx]<--R[%rdx]：R[%rax] mod S  R[%rdx]<--R[%rdx]：R[%rax] ÷ S | 无符号除法 |

3、乘法指令有两种不同的形式

* 双操作数乘法指令
* 单操作数乘法指令，mulq和imulq，这两条指令都要求一个参数必须放在寄存器%rax中，而另一个作为指令的源操作数给出。乘积结果放在寄存器%rdx(高位)和%rax(低位)中

4、除法或取模操作是由单操作数除法指令来提供的

* 类似于单操作数乘法指令，有符号数除法指令idivl将寄存器%rdx(高64位)和%rax(低64位)中的128位数作为被除数，而除数作为指令的操作数给出
* 指令将商存在寄存器%rax中，将余数存在寄存器%rdx中
* 对于大多数64位除法应用来说，被除数也常常是一个64位的值，这个值应该存放在%rax中，%rdx的位应该设置为全0(无符号运算)或者%rax的符号位(有符号运算)。这个操作可以用cqto来完成，这条指令不需要操作数---它隐含读出%rax的符号位，并将它复制到%rdx的所有位