**3 Machine-Level Representation of Programs**

# 英语

|  |  |
| --- | --- |
| 1. shield屏蔽 2. reference引用 3. consistent一致的 4. inefficiency低效率 5. malware恶意软件 6. involve涉及、调用 7. thereby从而 8. recursive循环的，iterative迭代的 9. delude oneself自欺欺人 10. arcane神秘的 11. feasible可行的   economically feasible and technically desirable经济可行、技术可实现   1. compromise妥协 2. safeguards保障措施】 3. elementary基本的 4. aggregate聚合的 5. guildeline指导方针 6. strip away剥去 7. annotated带解释的 8. ambiguity歧义 9. owing to由于 10. impose提出 11. excerpt摘录 12. successive连续的 | 1. specify指定 2. manipulate操作 3. underlying潜在的 4. get a sense of感觉到 5. infest侵扰 6. nuance细微差别 7. eliminate消除 8. prerequisite先决条件 9. corporation公司 10. proceed继续前进 11. legacy遗产 12. colloquially通俗地讲 13. overall最初的 14. elaborate复杂的 15. dictated被……制定的 16. whereas虽然 17. ever-changing不断改变的 18. indented缩进的 19. explanatory解释性的 20. dedicated专门的 21. enthusiasts爱好者 22. compact最紧凑的 23. reminiscent令人怀念的 24. outperform优于 25. adjacentl邻近的 |

**|** *Best of all, a program written in a high-level language can be compiled and executed on a number of different machines, whereas assembly code is highly machine specific.*

*When programming in a hign-level language such as C,and even more so in Java,we are shielded from the detailed machine-level implementation of our program*.*In contrast, when writing programs in assembly code (as was done in the early days of computing) a programmer must specify the low-level instructions the program uses to carry out a computation.*

# 3.1 *A Historical Perspective*

**|** 摩尔定律

预测芯片上的晶体管数量每一年半翻一番

# 3.2 *Program Encodings*

假设有两个C语言文件：p1.c和p2.c，然后使用gcc命令行进行编译生成可执行文件

gcc -0g -o p p1.c p2.c

命令gcc表示使用gcc这种C编译器进行编译，而且因为Linux上的默认C编译器就是gcc所以也可以简单地使用cc进行调用

-0g表示使用一种优化方式，产生一种遵守最初的原始C代码的机器代码。

为什么选择-0g是因为使用更高级别的优化将会生成经过大量转换的代码，以致于生成的机器代码和原始源代码之间的关系难以理解。因此，我们将使用 -Og 优化作为学习工具，然后看看当我们提高优化级别时会发生什么

但是在实践中，一般是使用更高级别的优化，比如-01，-02

和之前提到的编译系统一致，gcc命令经过“**预处理、编译、汇编、链接**”四步来产生最终的可执行程序

预处理：扩展源代码中的#include所引入的文件和#define所定义的宏，.c→.i

编译：将修改后的C文件.i使用编译器生成汇编文件p1.s和p2.s

汇编：使用汇编器将汇编文件转换为二进制可重定向文件

可重定向文件是机器代码的一致形式，它包含着所有指令的二进制形式，但是并没有填充全局变量的地址

链接：将多个可重定向文件p1.o和p2.o以及所调用的库函数功能重定向文件合并并生成最后的可执行文件p(-o p)

可执行文件是机器代码的第二种形式，是处理器执行的代码的确切形式。

## 3.2.1 *Machine-Level Code*

和1.9.3所描述的类似，计算机系统采用几种不同形式的抽象注释1，通过使用更简单的抽象模型来隐藏实现的细节

对于机器级编程而言，这些抽象层次中有两个是比较重要的

* 1. 指令集体系结构ISA：定义了处理器的状态，指令格式以及每条指令对状态的影响。大多数ISA定义了程序的指令序列是串行执行的
  2. 虚拟存储：提供了一个非常大的字节数组的存储模型

x86-64机器代码和C语言原始代码差别很大，一些对于C程序员来说不可见的处理器状态对于x86-64机器代码来说是可见的

* 1. PC：程序计数器，**在x86-64中标识为%rip，其中存储着下一条要执行的指令的地址**
  2. **整数寄存器文件包含着存储着64位值的16个命名位置**注释2
  3. **条件码寄存器保存着最近执行的算术或者逻辑指令的状态信息**
  4. 一组向量寄存器可以存放一个或多个整数或浮点数值

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。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来表示。即使是对标量数据类型，汇编代码也不区分有符号或无符号整数，不区分各种类型的指针，甚至于不区分指针和整数。

在任何时候，只有有限的一部分虚拟地址被认为是合法的。例如x86-64虽然是64位的虚拟地址，但是在实践中高16位必须设置为0，从而实际能够访问的是范围内的一个字节

## 3.2.2 Code Examples

**|** gcc -0g -S mstore.c中的"-S"是表示只执行到编译阶段生成汇编代码文件

**|** gcc -0g -c mstore.c中的"-c"是表示只执行到汇编阶段生成可重定位的目标文件

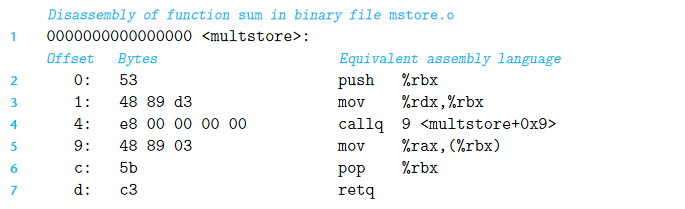
#### 反汇编器 disassembler

为了查看该机器代码文件的内容，需要使用一类称为反汇编器的程序

反汇编器将会从机器代码中生成一种和汇编代码格式类似的文件

在Linux/Windows系统中，使用命令objdump和-d命令参数来对机器代码文件进行解析。类似的解析结果如下：

objdump -d mstore.o



在左侧，有 14 个十六进制字节值，按前面显示的字节序列列出，分为 1 到 5 个字节的组。这些组中的每一个都是一条指令，其汇编语言等效项显示在右侧

可以从中总结到几个特征：

* 1. x86-64的指令长度从1-15字节不等，采用哈夫曼编码，使用频度高/操作数少的指令字节数少；使用频度低/操作数多的指令字节长
  2. 设计指令格式的方式是：**对于某种指令都给一个字节唯一用来标识此处开始为该指令**。例如上图中“push以53起始，mov以48起始，pop以5b起始，retq以c3起始，callq以e8起始”
  3. 反汇编器只是基于程序的机器代码文件中的字节序列来确定，不需要访问程序的源文件和汇编文件
  4. 反汇编器生成的汇编代码和编译器生成的汇编代码在指令的命名规则上有细微差别，例如指令后缀加不加“q”

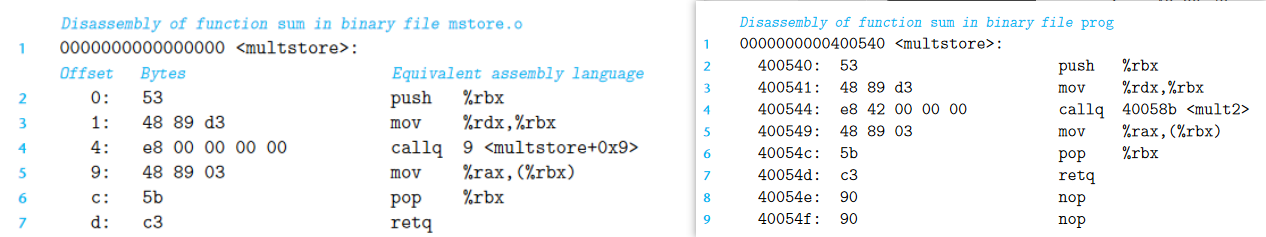
#### 链接main文件

生成实际可执行的代码需要对一组目标代码文件注释3运行链接器，而这一组目标代码文件中必须含有一个main函数

gcc -0g -o prog main.c mstore.c即预处理、编译、汇编、链接main和mstore两文件，最后生成prog可执行程序

#### 链接后反汇编与编译后反汇编的对比

链接之后的可执行文件的反汇编和编译之后的可重定向目标文件的反汇编示例如下：



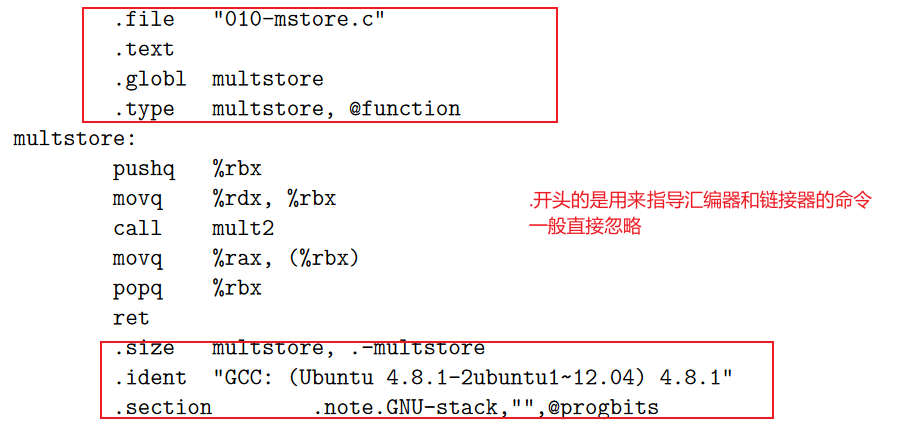
不同的点主要有：

* 1. 链接后的反汇编代码还包括用于启动和终止程序以及与操作系统交互的代码
  2. 链接后的反汇编代码已经给代码分配好了具体的地址范围，并直接指明了要跳转到的地址
  3. 插入的nop只是为了将函数代码变为16字节注释4，就存储器系统性能而言可以更好地放置下一个代码块

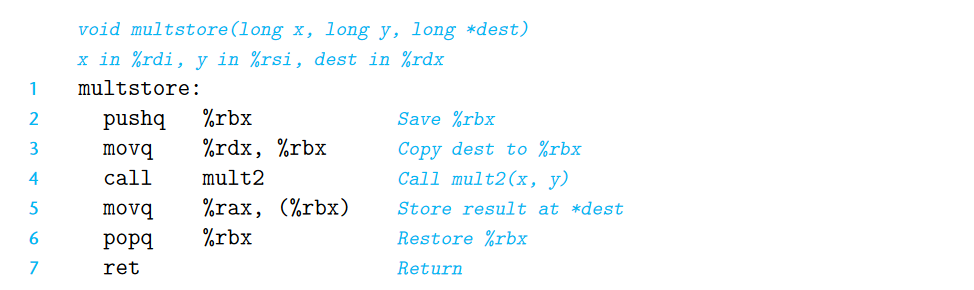
## 3.2.3 *Notes on Formatting*

经GCC编译器生成的汇编代码如下，并不容易阅读。有两方面的原因：

* 1. 该汇编代码中包含一些程序员不太需要关注的信息
  2. 该汇编代码并未提供任何关于程序或是它怎么工作的描述



下面是带有注释的、省略掉"."指令的汇编代码



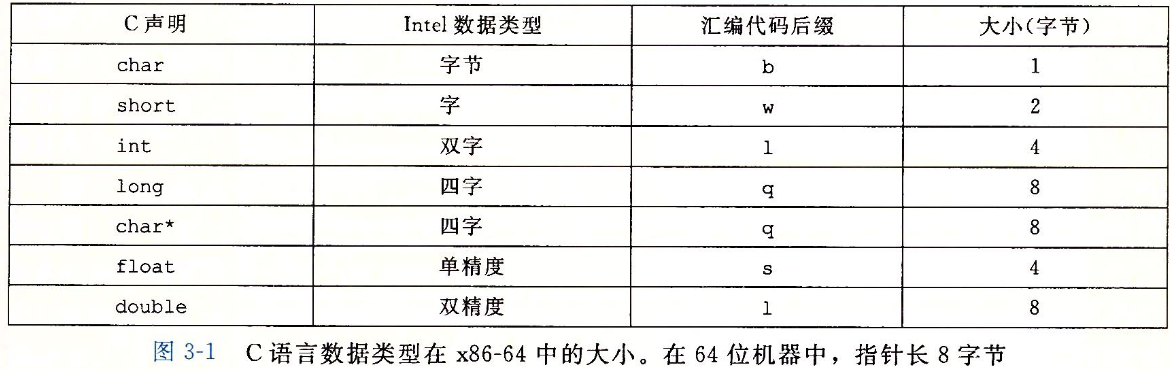
🐳对于一些应用程序，程序员必须用汇编代码来访问机器的低级特性。

一种方法是用汇编代码编写整个函数，在链接阶段把它们和C函数组  
合起来。  
另一种方法是利用GCC的支持，利用asm伪指令直接在C程序中嵌入汇编代码。

# 3.3 *Data Formats*

由于x86-64起源为 16 位架构，后来扩展到 32 位架构，Intel 使用术语“字Word”来指代 16 位数据类型。基于此，将 32 位数据量称为“双字DW”，将 64 位数据量称为“四字QW”

下图是C语言基本数据类型对应的x86-64表示



int占4字节，指针8字节，long占8字节

**大多数GCC生成的汇编代码指令都有一个字符后缀“b/w/l/q”，表示操作数的大小**。例如，数据传送指令有4个变种：movb、movw、movl、movq。注意float、double后缀是s、l。这不会引起歧义，因为浮点数与整数使用不同的指令和寄存器文件

# 3.4 *Accessing Information*

x86-64 中央处理单元 (CPU) 包含一组 16 个通用寄存器，用于存储 64 位值。这些寄存器用于存储整数数据和指针。如下图所示：



这16个寄存器名均以“%r”起始，但是对于32位、16位、8位的寄存器数据则有不同的命名规则。这是由于指令集的变革

* 1. 刚开始8086只有8个16位的寄存器，即ax~sp，根据它们的实际作用进行命名——高位h低位l
  2. 之后扩展到IA32，寄存器命名为eax~esp表示扩展到32位
  3. 再之后扩展到IA64，前8个寄存器命名为rax~rsp。并添加了剩下8个寄存器，命名r8~r15，32位为双字加后缀d，16位为字加后缀w，8位为字节加后缀b

因为指令允许对这16个寄存器的不同数据量位注释5进行操作，那么当将这些寄存器作为目的寄存器且操作数据大小少于8字节，就会遇到高位字节怎么办的情况：

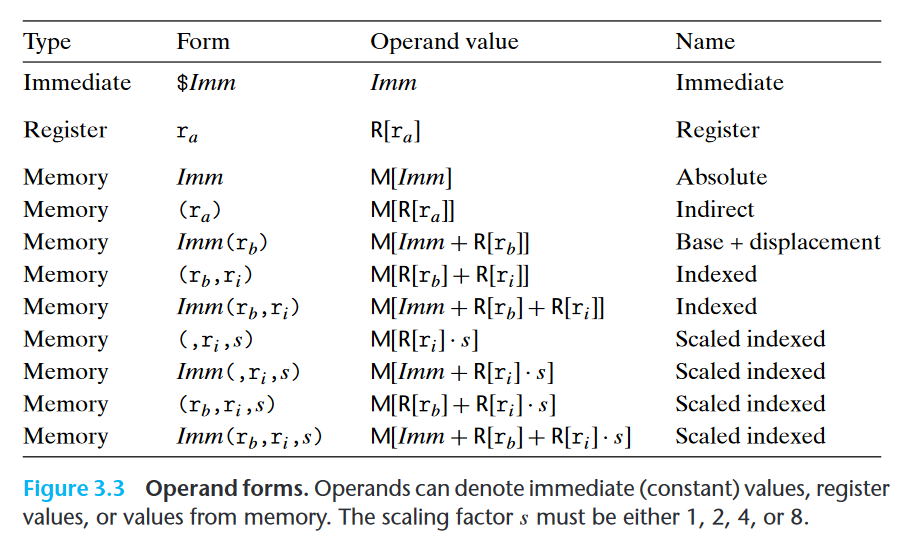
* 1. 操作1字节和2字节（字）的指令保持高位字节数据不变
  2. 操作4字节（双字）的指令，赋值高位4字节为0注释6

Figure3.2右侧的注释表示不太的寄存器在程序中充当的不同的作用

除了%rsp专用表示运行时栈的结束位置，并且有专门的指令来读写这个寄存器。其他15个寄存器的使用都较为灵活

## 3.4.1 *Operand Specifiers*

大多数指令都有一个或多个操作数指示符来指出执行一个操作所需要的源数据值以及结果要存放的目标位置。x86-64支持如下图所示的操作数格式：



各种不同操作数的可能性可分为三种类型：

* 1. 立即数

立即数表示常数值，书写格式为：$+C语言标准所表示的整数

eg:$12,$0x12

* 1. 寄存器数

寄存器数表示的是数据来源于x86-64的16个通用寄存器，且数据大小可以是字节、字、双字、四字。书写格式为：

* 1. 存储器数

根据存储器地址的表示方法可以分为：存储器立即寻址、寄存器间接寻址、寄存器相对寻址，变址寻址（带立即数和不带立即数），比例变址寻址(4种)

计算出的存储器地址称之为有效地址

🐳在这些寻址方式中，最常用的形式是其中**s必须是1、2、4、8，必须是64位的寄存器**。这个形式经常用来引用数组元素，其余寻址形式都可以省略这个形式的某部分来形成。

## 3.4.2 *Data Movement Instructions*

将许多不同的指令分为指令类，其中类中的指令执行相同的操作，但操作数大小不同

#### MOV类

|  |  |
| --- | --- |
|  | MOV类指令可以不需要任何转变的数将据从源位置复制到目的位置  MOV类包括movb、movw、movl、movq四类指令，它们执行的效果是相同的唯一的区别在于操作的数据大小不同：1、2、4、8 |

🐳源位置可以是一个立即数、寄存器数或者存储器数；目的位置可以是寄存器数或者存储器数。但是不允许源位置、目的位置都是存储器数

因此从存储器某位置复制数据到另一个位置需要使用两条指令：先mov到某寄存器，再由该寄存器mov到主存

mov指令使用的寄存器操作数可以是16个通用寄存器中的任意一个，但是需要保证使用寄存器片段的大小和指令的最后一个字符指令的大小相同(b、w、l、q)。且**对b、w、q指令的情况，mov指令只会更新目标操作数指示的特定区域，除了movl指令**注释7**会使得寄存器高4B为0**

下面是mov类指令的一些示例：

movl $0x4050,%eax %Immediate--Register,4 bytes  
movw %bp,%sp %Register--Register,2 bytes  
movb (%rdi,%rcx),%al %Memory--Register,1 byte  
movb $-17,(%esp) %Immediate--Memory,1 byte  
movq %rax,-12(%rbp) %Register--Memory,8 bytes

NASM

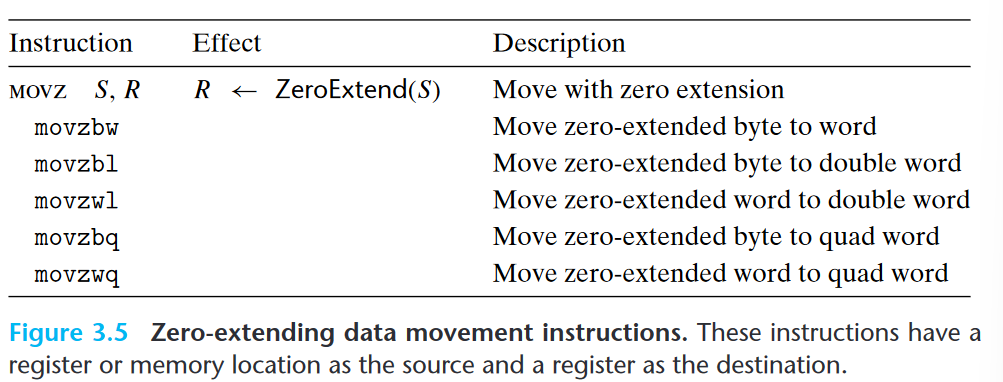
**|** Figure3.4中的最后一条指令movabsq是用来处理64位立即数的注释8。它可以直接使用任意的**64位立即数值**作为源操作数，但是**只能使用寄存器作为目的地址**

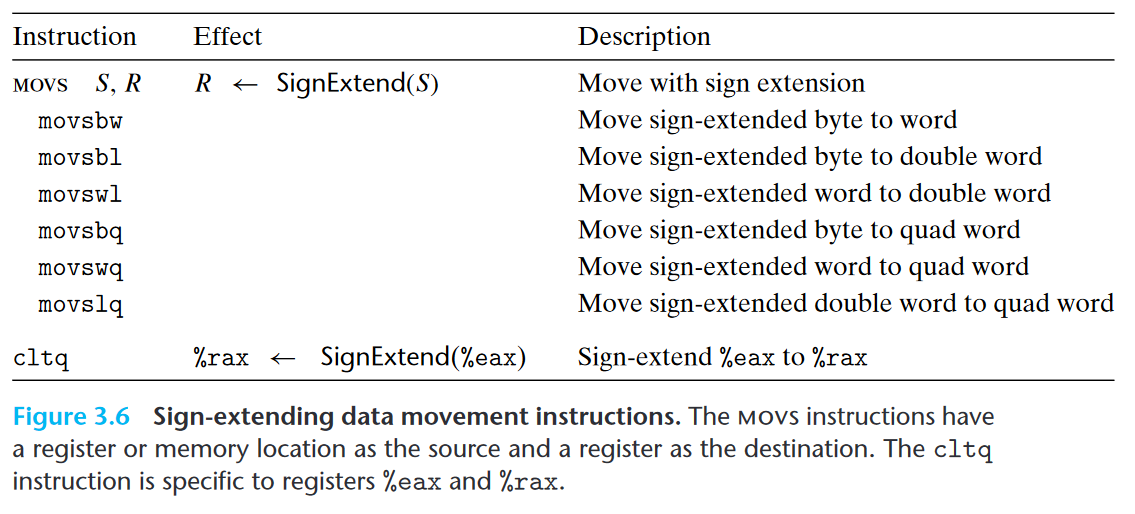
#### MOVZ、MOVS

🐳MOVZ和MOVS指令是用来将一个较小的数移动到一个较大的目的位置上。源位置可以是寄存器或者存储器，但是目的位置只能是寄存器。

1. MOVZ指令是将目的位置上的剩余字节填充0  
   MOVS指令是将目的位置上的剩余字节填充源操作数的MSB
2. 对于b、w、l、q四类数据大小，总共的指令类别有种

但是由于x86-64既有标准——对于任何产生32位结果的指令，若将结果放置在64位的目的处，则高位部分需设置为0。因此不存在movzlq



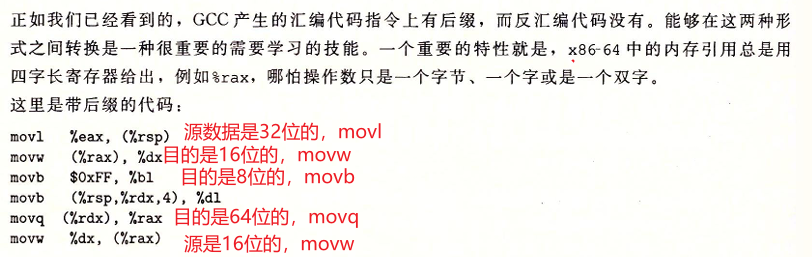


**|** cltq指令

cltq指令没有显式指定操作数，它隐含源操作数%eax以及目的操作数%rax，执行的功能是。等价于

但是相较于movslq指令需要指明操作数，cltq并不需要因此指令字长更紧凑

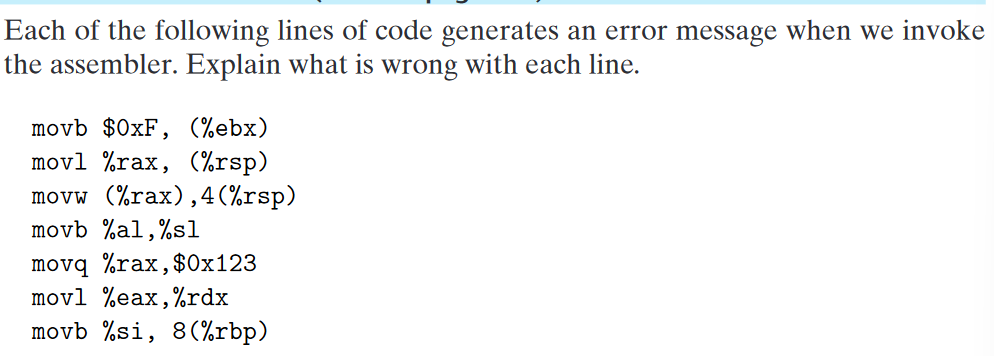
**|** 根据源、目数据类型填充mov指令



需要注意的是，**内存地址总是用64位的**

判断依据：**若目的位置是寄存器操作数，则需要根据寄存器所使用的大小决定mov的后缀；但如果使用的是存储器操作数，则需要根据源位置操作数大小来决定**

**|** 错误的指令示例：



1. 内存引用需要使用64位
2. movl是32位，但是%rax是64位
3. mov指令不能用于内存和内存之间的数据移动
4. 没有%sl这个寄存器，应该是%sil
5. 不能将立即数作为目的寄存器
6. movl是332位的，但是%rdx为64位
7. %si寄存器为16位，但是使用的是movb指令

## 3.4.3 *Data Movement Example*

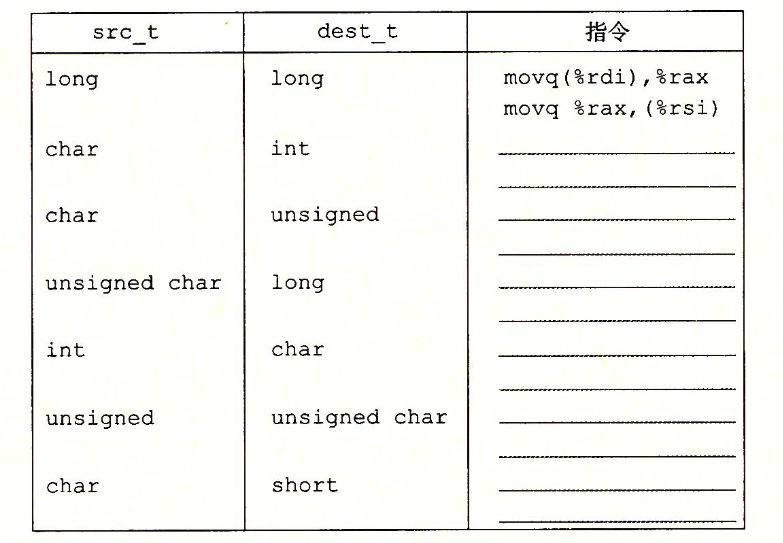
以Figure3.7中的交换值的代码示例为例进行分析

|  |  |
| --- | --- |
| 函数中的局部变量是被保存在寄存器中而不是内存中的。因为访问寄存器远快于访问主存 | 1. 返回值保存在%rax中；参数保存在%rdi、%rsi中   %rdi中存储的是xp指针的值 %rsi中存储的是y的值   1. 指针的解引用过程如下：    1. 首先将内存中指针的值赋给某个寄存器    2. 通过寄存器间接寻址的方式形成内存的有效地址，从而获得对应的值 |

**|** 使用MOV、MOVZ、MOVS类指令完成类型强制转换

\**dp=(dest\_t)*\*sp,sp存放在%rdi中，dp存放在%rsi中

中间变量使用%rax、%eax、%ax、%al



1. long→long：64位到64位

读数据：movq (%rdi),%rax

写数据：movq %rax,(%rsi)

1. char→int：8位到32位

读数据：movsbl (%rdi),%eax 因为写数据要求使用%eax则存数据也要使用%eax，所以需要进行符号扩展，使用movsbl指令

写数据：movl %eax,(%rsi) 这里必须要使用%eax因为数据大小由源来决定

1. char→unsigned：8位到32位

对于既要进行位扩展也要符号转换的情形：**C语言规定先进行位扩展再进行符号转换**

因此实际的调用指令同2

1. unsigned char→long：8位到64位

无符号数进行零扩展使用movzbl指令

读数据：movzbl (%rdi),%eax

写数据：movq %rax,(%rsi)

1. int→char：32位到8位

读数据：movl (%rdi),%eax

写数据：movb %al,(%rsi)

1. unsigned→unsigned char：32位到8位

同5

1. char→short：8位到16位

读数据：movsbw (%rdi),%ax

写数据：movw %ax,(%rsi)

## 3.4.4 *Pushing and Popping Stack Data* %rsp

|  |  |
| --- | --- |
|  | 栈顶采用“高到低”**向下增长**的形式且**栈顶有数据**  入栈：先减栈顶指针8，然后入数据 出栈：先取数据，再加栈顶指针8 |

|  |  |
| --- | --- |
|  | 在x86-64中，栈存放在内存中的某个区域。采用**向下增长的原则使得栈顶元素的地址是所有栈中元素地址最低的**——但是根据惯例，画栈时栈顶在最下端，采用向上增长 |

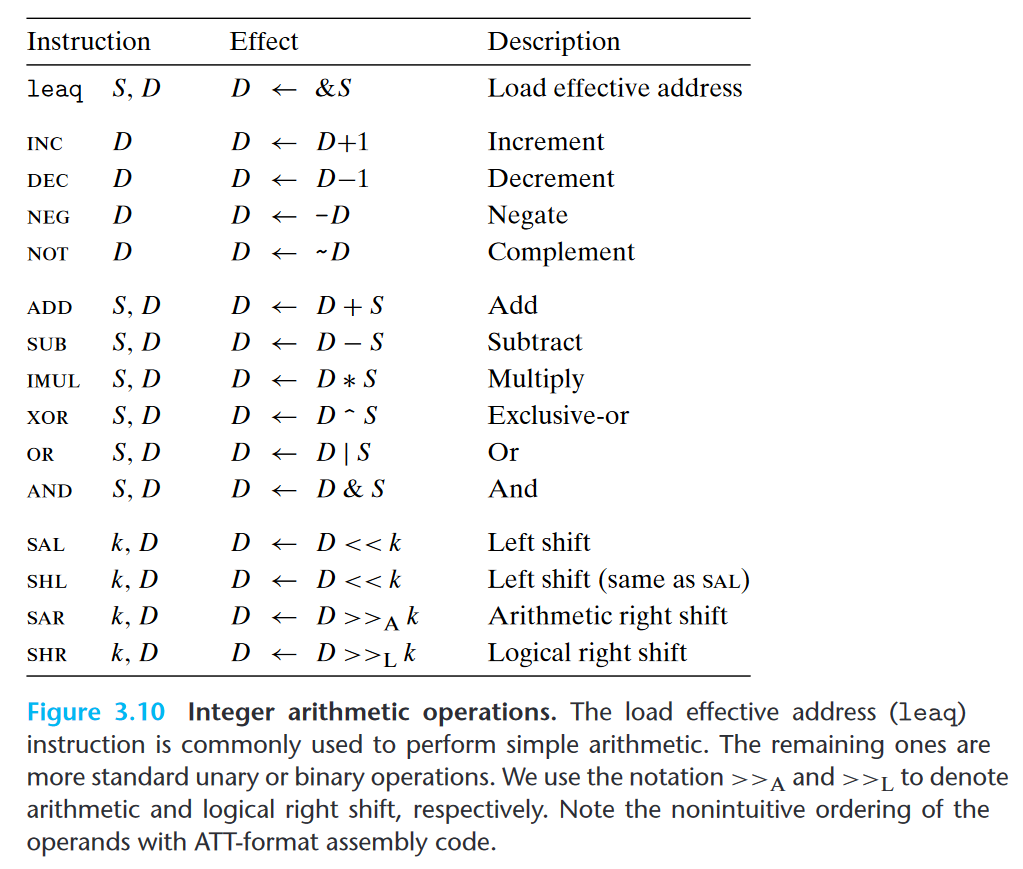
**出栈后，只是栈顶%rsp的值发生了变化，但是原栈顶位置上的元素还在直到被重新push覆盖**

**因为栈也存放在内存中，所以可以直接使用寄存器间接寻址的方式访问栈元素**

比如movq 8(%rsp),rdx则是将栈顶的下一个四字元素复制到%rax中

# 3.5 *Arithmetic and Logical Operations*

算术逻辑运算指令根据操作类型分为：加载有效地址、一元操作、二元操作和移位

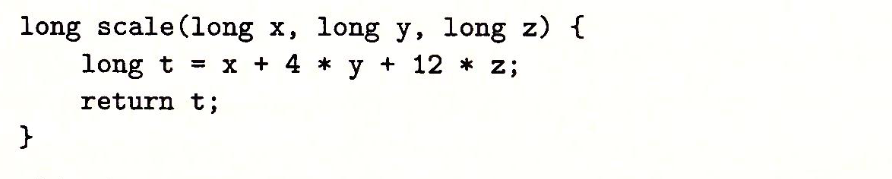


## 3.5.1 *Load Effective Address*

加载有效地址leaq指令实际上是movq的变形，它限制了源操作数必须是个内存引用，目的操作数必须是寄存器，但是并不从内存指定位置读取数据，而是将内存引用的有效地址写寄存器。leaq指令可以得到内存引用指针的值也可以进行一些利用缩放变址寻址方式实现一些简单的计算

eg：若%rdx值为x，则leaq 7(%rdx,%rdx,4),%rax即%rax=5x+7

因此编译器可以使用leaq指令来完成一些优化操作，如下：



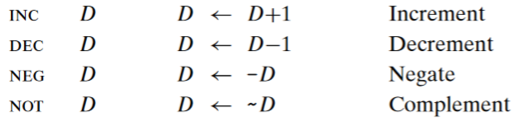
则若*x in %rdi,y in %rsi,z in %rdx*，那么x+4yi即leaq (%rdi,%rsi,4),%rax  
12z即3z\*4，leaq (%rdx,%rdx,2),%rdx,leaq(%rax,%rdx,4),%rax

🐳一定要注意leaq指令能够执行加法和有限形式的乘法，在编译一些简单的表达式时是十分有用的

## 3.5.2 *Unary and Binary Operations*

### 3.5.2.1 Unary Operations

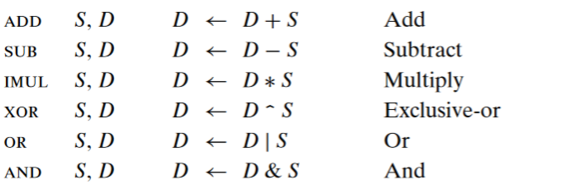
一元操作只有一个操作数，这个操作数既是源操作数也是目的操作数，**操作数的类型只能是寄存器或者存储器**。例如C语言中的自增、自减运算符



### 3.5.2.2 Binary Operations

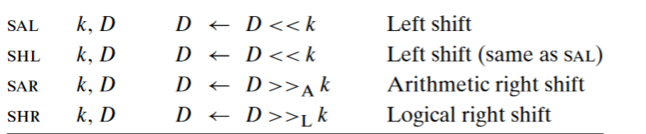
二元操作有两个操作数，其中**第二个操作数既是源操作数也是目的操作数**。例如subq %rax,%rdx执行的功能是

二元操作的第一个源操作数可以是立即数、寄存器或存储器；但是第二个操作数只能是寄存器/存储器注释9



## 3.5.3 *Shift Operations*

移位操作是二元操作：第一个操作数是移位量，移位量的表示可以使用立即数也可以使用%cl；第二个操作数是要移位的值，类型是寄存器或者存储器

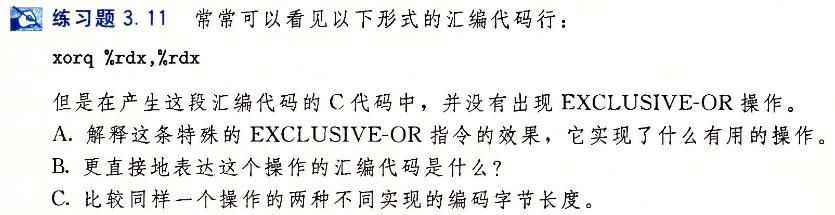


🐳原则上虽然使用%cl表示移位量，范围为0~255。但是在x86-64中，对w位长的数据值进行移位操作时，移位量是由%cl寄存器的低m位决定的，**也就是移位量%w**

例如当%cl=0xff时，salb会移动111b位，salw会移动1111b位，sall会移动11111b位，而salq会移动111111b位

## 3.5.4 *Special Arithmetic Operations*

**|** **使用异或指令实现寄存器的赋0**



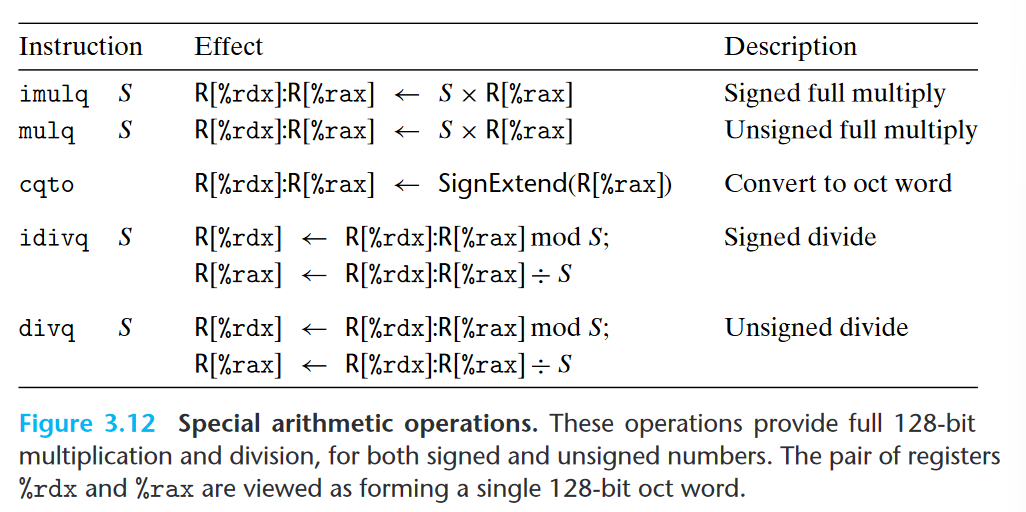
A.这条指令可以将%rdx置为0

B.movq $0,%rdx

C.使用XOR编码长度更少

正如我们在 2.3 节中看到的，两个 64 位有符号或无符号整数相乘可以产生需要 128 位来表示的乘积。 x86-64 指令集对涉及 128 位（16 字节）数字的操作提供有限的支持。继续采用字（2 字节w）、双字（4 字节l）和四字（8 字节q）的命名约定，**Intel 将 16 字节数量称为八进制字**o

下图描述了x86-64指令集中支持128位的整数乘法和除法



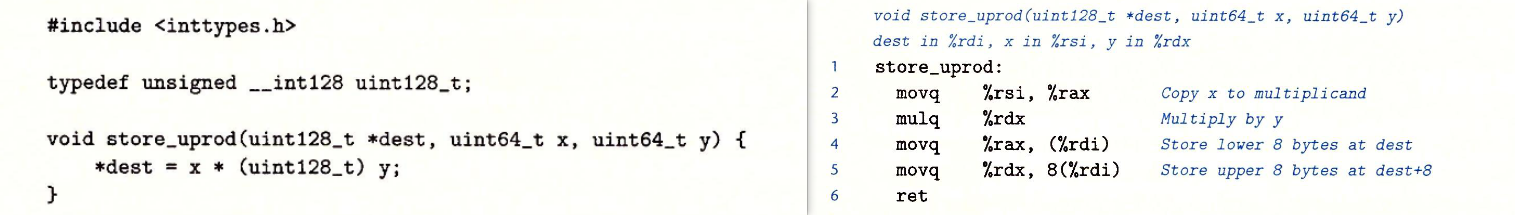
### MUL

imulq指令有两种形式：

* 1. 双操作数形式的IMUL：imul S,D是将S\*D→D
  2. 单操作数形式的IMUL：imulq S隐含操作数%rax为源操作数，是将S\*R[%rax]的高位结果写R[%dx]，低位结果写R[%rax]——**有符号计算**

和单操作数imul有符号乘法类似的还有无符号单操作数乘法mulq

mulq S同样隐含源操作数%rax，也同样是将R[%rax]\*S的高位结果写R[%rdx]，低位结果写R[%rax]



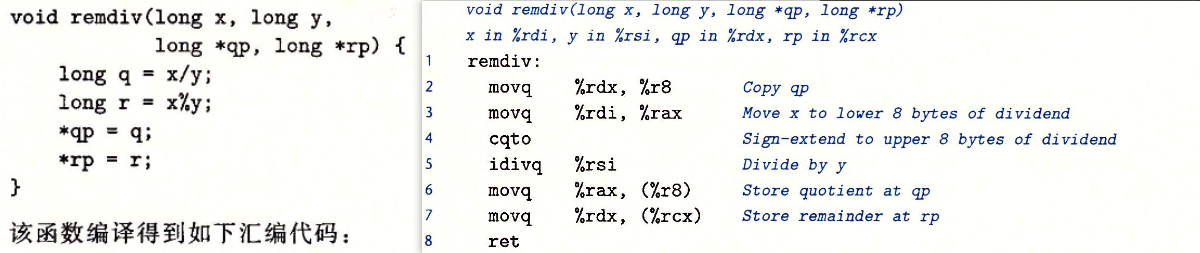
上图是一个计算64位无符号数乘以64位无符号数得到128位结果的函数，其对应的汇编代码如右半图所示。根据汇编代码可以得到如果想保留该128位的结果，需要使用两个movq；且根据机器的字节顺序，来决定先放高位还是先放低位

### DIV

有符号数除法指令idivq是单操作数指令，idivq S是将R[%rdx]:R[%rax]组成的128位有符号数除以S计算得到的商放R[%rax]，余数放R[%rdx]

**|** cqto指令——读取R[%rax]的符号位，并复制到R[%rdx]所有位，实现128位的符号扩展

**|** 当被除数是64位值时，这个值应该存放在R[%rax]中，R[%rdx]的所有位应该设为全0(无符号运算)或者是R[%rax]的符号位(有符号运算)——使用cqto指令实现



# 3.6 *Control*

🐳机器代码提供两种基本的低级机制来实现有条件的行为：测试数据值，然后根据测试的结果来改变控制流或者数据流

## 3.6.1 *Conditions Codes*

除了整数寄存器之外，CPU 还维护一组单个位的条件码寄存器，用于描述最近算术或逻辑运算的属性，通过检测这些条件码来进行条件分支。其中最常用的条件码如下：

* 1. CF：进位标志。最近的操作使得最高位发生了进位，可以用于检测无符号数加法的溢出
  2. ZF：零标识。最近的操作使得结果为0
  3. SF：符号位。最近的操作使得结果为负
  4. OF：溢出位。最近的操作导致出现一个补码溢出——可能是正溢出也可能是负溢出

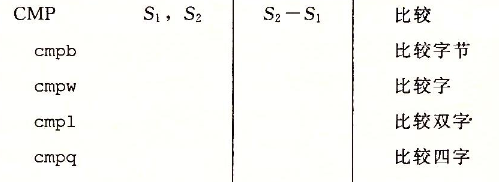
以t=a+b为例，表示上述的条件码如下：

🐳虽然leaq指令可以用来做一些简单的加法和乘法，但是它并不改变条件码。除此之外Figure3.10[行内引用](https://www.wolai.com/meDbLwZrbLCTmTFzF8fkxF#axw5cykhZmBqENv8K7QoH7)中的所有指令均会改变条件码

* 1. 对于逻辑指令OR、AND、XOR，会设置CF和OF为0
  2. 对于移位指令，CF设置为最后一个被移出的位，而OF设置为0
  3. INC、DEC指令会设置OF和ZF，但是不改变CF

除此之外，还有两个指令类只改变条件码而不改变任何其他寄存器

* 1. CMP指令类



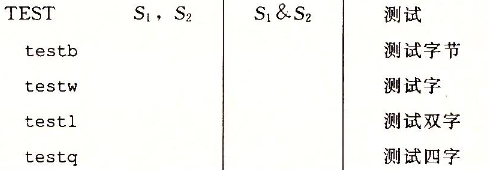
cmp类指令执行的行为同sub指令，但是并不将结果写回，而是只根据结果来设置条件码。且使用的是ATT汇编格式，操作数是倒序的 cmp S1,S2→S2-S1

根据条件码的情况来判断S1和S2的大小关系：S2-S1

有符号数小于：S2<S1→SF^OF  
有符号数小于等于：S2≤S1→SF^OF|ZF  
有符号数大于：S2>S1→~(SF^OF)^~ZF  
有符号数大于等于：S2≥S1→~(SF^OF)

无符号数小于：S2<S1→CF  
无符号数小于等于：S2≤S1→CF|ZF  
无符号数大于：S2>S1→~CF^~ZF  
无符号数大于等于：S2≥S1→~CF

* 1. TEST指令类



test类指令执行的行为同and指令，但是并不将结果写回，而是只根据结果设置条件码

test类指令的作用：

* + - 1. 一般用来判断某个寄存器内的值是负数、整数还是0

test S1,S1操作数重复，则根据结果设置的条件码，若ZF=1则S1为0；若SF=1则结果为负，否则为正

* + - 1. 掩码与

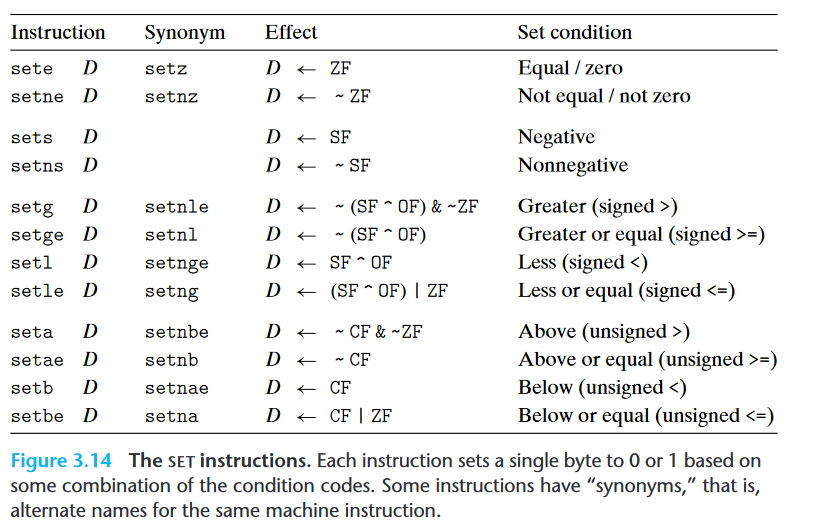
将其中一个操作数设置为掩码来进行计算

## 3.6.2 *Accessing the Condition Codes*

通常不能直接读取条件码，而是使用下面三种方式来使用条件码：

* 1. 可以通过指令根据条件码的某种组合来将一个字节设置为0或1

指令的后缀表示的是所考虑的不同条件码的组合



🐳SET指令只产生8位结果，指令中的D可以是内存某个字节位置也可以是寄存器的低8位。若要产生16/32/64的结果，则需要使用movz指令将高位0扩展

一般比较64位a、b大小的指令序列如下：

//int comp(data\_t a,data\_t b)  
//a in %rdi,b in %rsi  
comp:  
 cmpq %rsi,%rdi //R[%rdi]-R[%rsi]  
 setl %al  
 movzbl %al,%eax//不仅复位32位的前3个字节，也复位64位的前4个字节  
 ret

NASM

* 1. 可以条件跳转到程序的某个其他部分
  2. 可以有条件地传送数据

## 3.6.3 *Jump Instructions*

**|** *A jump instruction can cause the execution to switch to a completely new position in the program. These jump destinations are generally indicated in assembly code by a label*

|  |  |
| --- | --- |
|  | 左图中列出了x86-64所支持的跳转指令。其中包括无条件跳转jmp指令以及结合条件码的某种组合的j型条件跳转指令  jmp指令可以分为直接跳转和间接跳转两种  直接跳转：jmp label 间接跳转：jmp \*Operand  j指令均是直接跳转 |

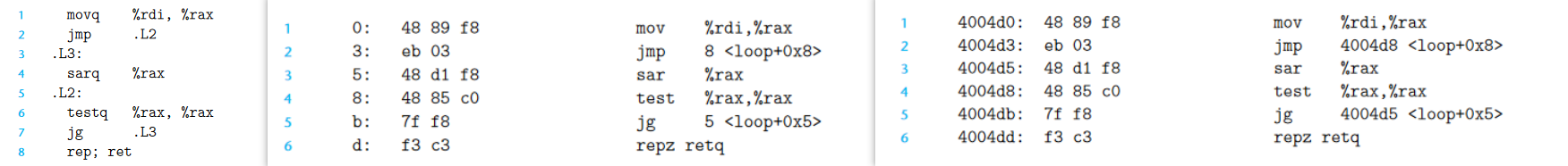
🐳在生成可重定位的目标代码文件时，汇编器会确定所有带标号指令的地址，并将该地址编码为跳转指令的一部分。而采用间接跳转的jmp指令则需要访问寄存器/存储器得到跳转目标地址

## 3.6.4 *Jump Instruction Encodings*

跳转目标的编码方式有两种：

* 1. PC相对编码注释10：使用跳转目标地址以及跳转指令后面那条指令的地址之间的差作为编码
  2. 绝对地址：用4个字节直接指定跳转目标

以下图的汇编代码为例，从左至右依次是其源汇编代码、可重定向目标代码以及可执行文件

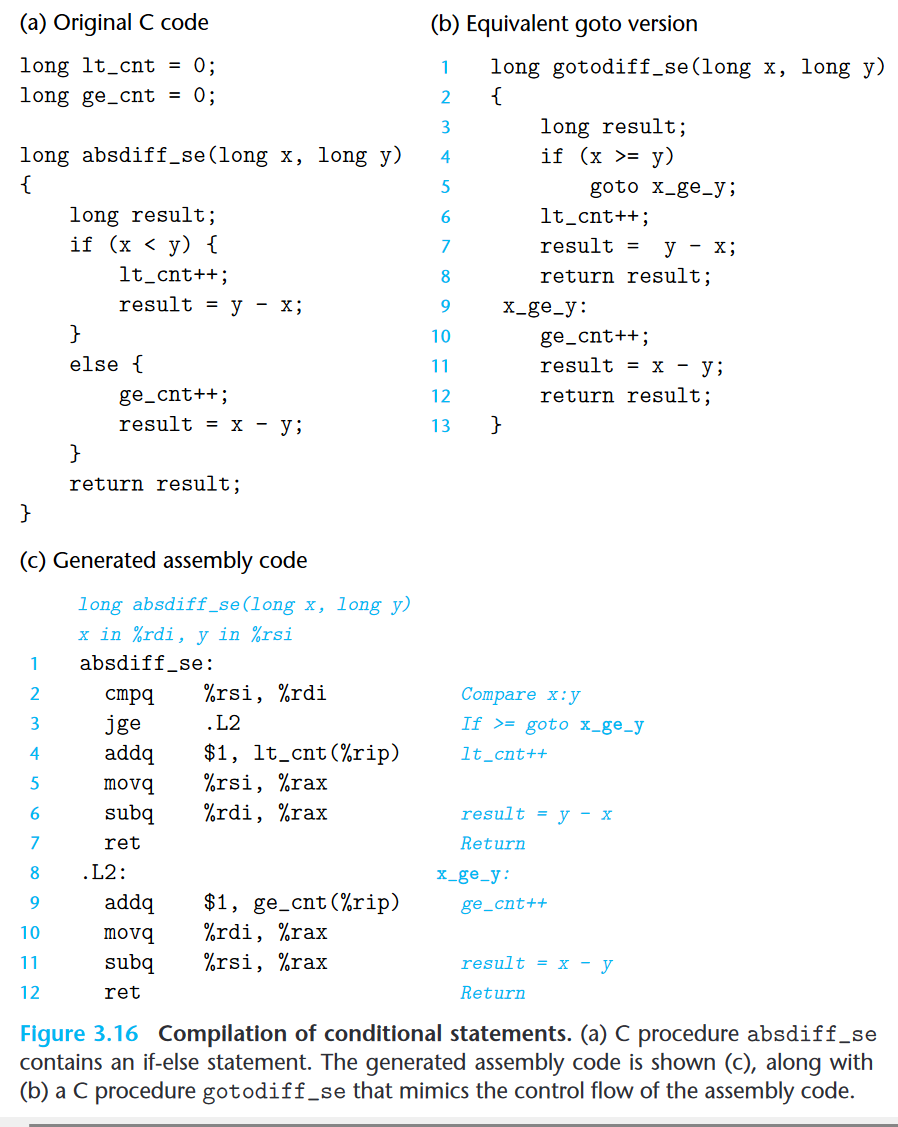


在左图汇编代码中有两条跳转指令，均采用直接跳转的方式。采用PC相对编码

第一个jmp指令要调到.L2即地址8处，而当前跳转指令的下一条地址是5，因此跳转目的编码为03；第二个jg指令要跳转到地址5处，跳转指令的下一条地址是d，因此跳转目的编码为-8，即11111000b

在链接后的可执行文件中，该程序已被装配到内存中的某个位置处。但是指令所编码的目标地址并没有改变

## 3.6.5 *Implementing Conditional Branches with Conditional Control*



以上述代码为例展示C语言和汇编语言中的控制分支转换

通过分析可以得到C中的if-else语句块的汇编表示的控制流的一般形式：

if (test-expr)  
 then-statement  
else  
 else-statement

C

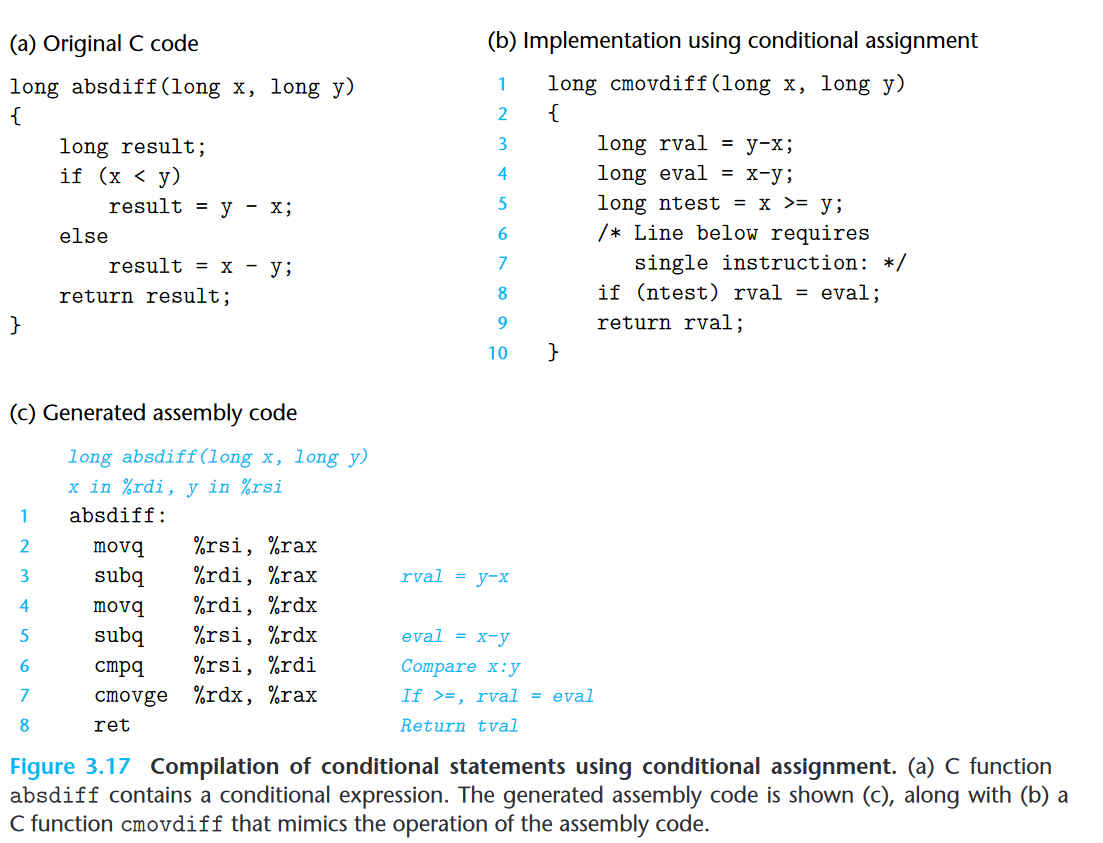
t =test-expr;  
 if (!t)  
 goto false;  
 then-statement  
 goto done;  
false:  
 else-statement  
done:

C

## 3.6.6 *Implementing Conditional Branches with Conditional Moves*

**|** *The conventional way to implement conditional operations is through a conditional transfer of control, where the program follows one execution path when a condition holds and another when it does not. This mechanism is simple and general, but it can be very inefficient on modern processors. An alternate strategy is through a conditional transfer of data.* ***This approach computes both outcomes of a conditional operation and then selects one based on whether or not the condition holds***.This strategy makes sense only in restricted cases, but it can then be implemented by a simple conditional move instruction that is better matched to the performance characteristics of modern processors.

下图是一个可以用条件传送编译的示例代码：

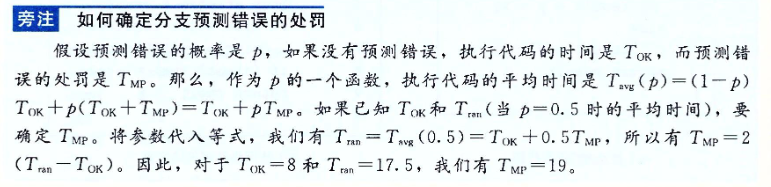


这种数据流的风格和控制流的风格不同点在于：数据流风格同时计算了x-y和y-x；然后根据x和y的大小关系，选择是否更改其中%rax所存储的值

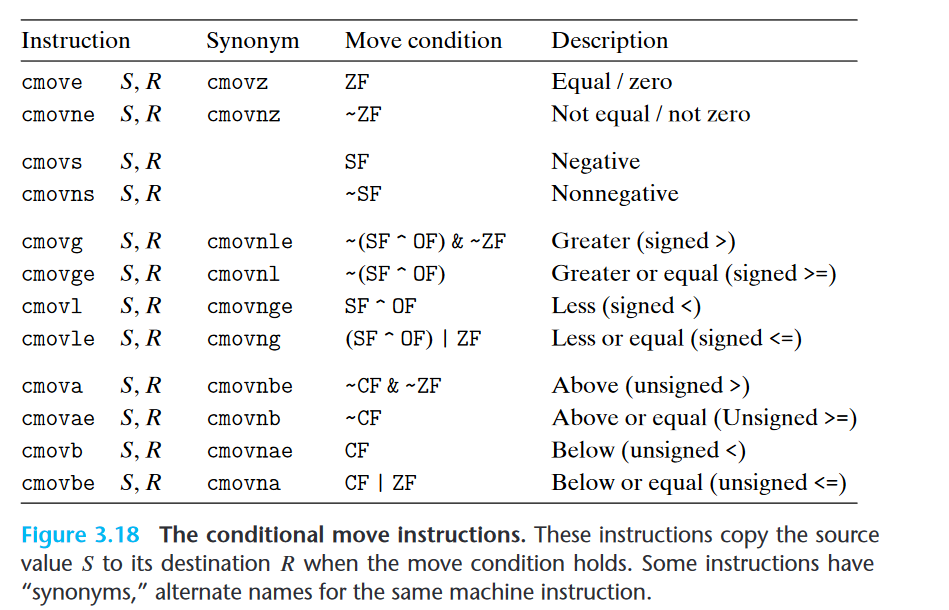
为了实现这种机制，采用了cmovge指令，即若上一条cmp指令结果是大于等于，那么就进行赋值

### 为什么要使用条件传送

至于基于条件数据传输的代码为什么是优于基于条件控制传输的代码的原因，必须理解现代处理器的工作原理——使用流水来实现高性能。对于流水遇到条件分支时采用分支预测技术。当分支预测成功时，会提高性能；但是当分支预测失败时，性能会严重降低。而如果使用条件传送编译，那么无论测试的数据是什么，所需的时间都约是8个时钟周期



### 条件传送指令cmov类



每个条件传送指令都有两个操作数：S和R。其中S可以是寄存器或者存储器，R只能是寄存器

源和目的值可以是16、32、64位，但不能是单字节。和无条件传送数据大小以后缀的形式表明不同，条件传送的数据大小取决于目的寄存器。**同条件跳转不同，处理器无需预测测试的结果就可以执行条件传送；处理器只是读源值（可能是从内存中），检查条件码，然后要么更新目的寄存器，要么保持不变**

### 条件传送理解

以条件表达式v=test-expr?then-expr:else-expr;为例：

1. 转变为条件控制形式

t=test-expr;  
 if(!t)  
 goto false;  
 v=then-expr;  
 goto done  
false:  
 v=else-expr;  
done:

C

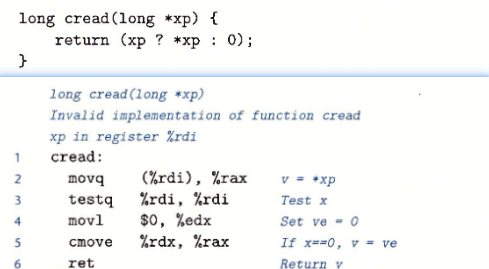
1. 转变为条件传送形式

v=then-expr;  
vn=else-expr;  
t=test-expr;  
if (!t) v=vn;

C

#### 条件传送的缺点

不是所有的都可以转换为条件传送的形式。比如，前面求值的表达式出现错误条件或者非法引用的情况，从而导致非法行为。



如上图，当rdi为空时，在第2行已出现对空的解引用，是非法行为

而且条件传送并不是总会提高代码效率。比如如果计算then-expr和else-expr太过复杂，那么编译器就必须考虑浪费的计算和由于分支预测错误所造成的性能处罚之间的相对性能。

一般根据经验，只有then-expr和else-expr都很容易计算时才采用条件传送；即使在分支错误预测的成本超过更复杂的计算的许多情况下，gcc 也使用条件控制传输。

## 3.6.7 Loops

**|** *C provides several looping constructs—namely, do-while, while, and for.No corresponding instructions exist in machine code. Instead, combinations of conditional tests and jumps are used to implement the effect of loops. Gcc and other compilers generate loop code based on the two basic loop patterns*

### Do-While Loops

Do-while格式循环的一般格式如下，它会重复执行body-statement直到test-expr为false。因此Do-While循环至少执行一次

do  
 body-statement   
while (test-expr);

C

将Do-While的C格式翻译为类汇编语言的格式如下：

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

NASM

### While Loops

while循环的一般格式如下，与do-while不同的是它首先检测test-expr是否成立。GCC一般采用两种方式注释11来将While循环转换为机器代码

while(test-expr)  
 body-statement

C

1. 跳转到中间注释12

跳转到中间的方法是通过执行无条件跳转到循环末尾的测试注释13来执行初始测试。其一般结构如下：

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

NASM

1. 受保护的do注释14

受保护的do的方法是通过在do-while前加一个条件分支来先判断是否满足循环条件，然后再do-while。其一般结构如下：

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

NASM

### For Loops

for循环的通过C语言格式如下，且C语言标准规定for循环可以等价为其下的while循环(有一个例外)

for(init-expr;test-expr;update-expr){  
 body-statement;  
}  
  
init-expr;  
while(test-expr){  
 body-statement;  
 update-expr;  
}

C

GCC编译for循环根据所采用的优化规则遵循While循环的“跳转到中间”/“受保护的do”的方法之一，其对应的goto代码如下：

//跳转到中间  
 init-expr  
 goto test  
loop:  
 body-statement  
test:   
 t=test-expr;  
 if(t)  
 goto loop  
   
//受保护的do  
 init-expr  
 t=test-expr;  
 if(!t)  
 goto done;  
loop:  
 body-statement  
test:   
 t=test-expr;  
 if(t)  
 goto loop;  
done:

NASM

## 3.6.8 Switch Statements

**|** *A switch statement provides a multiway branching capability based on the value of an integer index. They are particularly useful when dealing with tests where there can be a large number of possible outcomes. Not only do they make the C code more readable, but they also allow an efficient implementation using a data structure called a jump table.*

与一长串 if-else 语句相比，使用跳转表的优点是执行切换所需的时间与切换情况的数量无关。 GCC 根据 case 的数量和 case 值的稀疏性来选择翻译 switch 语句的方法。当存在多种情况（例如四种或更多）并且它们跨越较小范围的值时，使用跳转表

# 3.7 Procedures

[注释1] 虚拟机、进程(ISA+虚拟内存)、虚拟内存、文件

[注释2] x86-64只有16个整数通用寄存器

[注释3] main文件、库函数文件

[注释4] 2的n次幂

[注释5] B、W、DW、QW

[注释6] x86-64的惯例：任何为寄存器生成 32 位值的指令也会将该寄存器的高位部分设置为 0

[注释7] x86-64规定任何产生32位结果的指令，若要将结果放置在64位寄存器中，均需要将寄存器的高位设置为0

[注释8] 因为movq只能处理32位立即数，这个立即数然后会被符号扩展为64位

[注释9] 因为第二个操作数也要做目的操作数

[注释10] 这些例子说明，当执行P℃C相对寻址时，程序计数器的值是跳转指令后面的那条指令的地址，而不是跳转指令本身的地址。这种惯例可以追溯到早期的实现，当时的处理器会将更新程序计数器作为执行一条指令的第一步。

[注释11] 两种方式都使用与 do-while 循环相同的循环结构，但在如何实现初始测试方面有所不同

[注释12] GCC使用-0g时使用这种策略

[注释13] 意思是dowhile的变体，dowhile的测试在循环的末尾

[注释14] Gcc 在使用更高级别的优化进行编译时遵循此策略，例如使用命令行选项 -01