

汇编语言程序设计

汇编语言程序设计

1. 指令集与计算机系统结构
2. 信息的表示和处理
 - 2.1 整数
 - 2.1.0 预备知识
 - 2.1.1 逻辑运算
 - 2.1.2 数的机器表示
 - 2.1.2.1 机器字内的字节序:
 - 2.1.2.2 C语言中基本数据类型的大小(in Bytes):
 - 2.1.2.3 计算机中整数的二进制编码方式
 - 2.1.3 整数运算
 - 2.1.3.1 补码加法公式
 - 2.1.3.2 C语言中的无符号数与带符号数
 - 2.1.3.3 无符号数的加法
 - 2.1.3.4 移位 (除以2的k次幂)
 - 2.2 浮点数
 - 2.2.1 浮点数的标准
 - 2.2.1.1 规格化浮点数
 - 2.2.1.2 非规格化浮点数
 - 2.2.1.3 特殊值
 - 2.2.1.4 浮点数示例
 - 2.2.2 浮点数的舍入
 - 2.2.3 浮点数的运算
 - 2.2.4 C语言中的浮点数
3. 程序的机器级表示
 - 3.1 程序编码
 - 3.1.0 程序进行编码的过程
 - 3.1.1 机器级代码
 - 3.1.2 汇编代码示例
 - 3.2 信息的访问
 - 3.2.1 操作数
 - 3.2.2 数据传送指令
 - 3.2.3 压入和弹出栈数据
 - 3.3 算术和逻辑操作
 - 3.3.1 加载有效地址
 - 3.3.2 一元和二元操作
 - 3.3.3 移位操作
 - 3.3.4 特殊的算术操作
 - 3.4 控制
 - 3.4.1 条件码
 - 3.4.2 跳转指令
 - 3.4.3 实现条件分支
 - 3.4.3.1 用条件控制实现条件分支
 - 3.4.3.2 用条件传送来实现条件分支
 - 3.4.4 循环
 - 3.4.4.1 *do - while* 循环
 - 3.4.4.2 *while* 循环
 - 3.4.4.3 *for* 循环
 - 3.4.5 *switch* 语句
 - 3.5 过程
 - 3.5.1 运行时栈
 - 3.5.2 转移控制

- 3.5.3 数据传送
- 3.5.4 栈上的局部存储
- 3.5.5 寄存器中的局部存储
- 3.5.6 递归过程
- 3.6 数组分配和访问
 - 3.6.1 基本原则与指针的运算
 - 3.6.2 数组
 - 3.6.2.1 嵌套的数组
 - 3.6.2.2 定长数组和变长数组
- 3.7 异质的数据结构
 - 3.7.1 结构
 - 3.7.2 联合
 - 3.7.3 数据对齐
- 3.8 机器级程序中控制与数据的结合
 - 3.8.1 GDB 调试器
 - 3.8.2 内存越界引用和缓冲区溢出
 - 3.8.3 对抗缓冲区溢出攻击
 - 3.8.3.1 栈随机化
 - 3.8.3.2 栈破坏检测
 - 3.8.3.3 限制可执行代码区域
 - 3.8.4 支持变长栈帧
- 3.9 浮点体系结构
 - 3.9.1 浮点传送和转换操作
 - 3.9.2 过程中的浮点代码
 - 3.9.3 浮点的运算与比较操作
 - 3.9.3.1 浮点的运算操作
 - 3.9.3.2 浮点的比较操作
 - 3.9.4 定义和使用浮点数常数

1. 指令集与计算机系统结构

通用计算机冯诺依曼系统架构：

- IO Device
- Control Unit
- Arithmetic/Logic Unit
- Memory Unit

指令系统：

- CISC（复杂指令系统） 代表：X86
 - RISC（精简指令系统） 代表：MIPS/ARM/RISC-V
-

2. 信息的表示和处理

2.1 整数

2.1.0预备知识

1 Byte = 8 bit;

X86下, 1 Word = 2 Byte, MIPS或者RISC-V下, 1 Word = 4 Byte。

2.1.1逻辑运算

四种逻辑运算：**与** (AND, &)、**或** (OR, |)、**非** (NOT, ~)、**异或** (XOR, ^)

逻辑运算按位进行

2.1.2数的机器表示

机器字长：一般指计算机进行一次整数运算所能处理的二进制数据位数。分为32位字长和64位字长。在内存中，相邻机器字的地址相差4 (32-bit) 或者8 (64-bit)。

32位机的地址表示空间大约为4GB。原因是32位机的寻址空间大小为 2^{32} ，如果按照1个字节分配一个内存地址的话，这个大小相当于4GB（一个字节是计算机中最小的可寻址的内存单位）。

2.1.2.1机器字内的字节序：

- 大端 (Big Endian)：低位字节占据高地址。
- 小端 (Little Endian)：高位字节占据高地址。

2.1.2.2C语言中基本数据类型的大小(in Bytes)：

Data Type	Typical 32-bit	X86-32	X86-64
char	1	1	1
short	2	2	2
int	4	4	4
long	4	4	8(Linux)
long long	8	8	8
float	4	4	4
double	8	8	8
pointer	4	4	8

机器字长指明了指针数据的标称大小，可以看到在上表中指针类型变量在32位机和64位机上的大小是不同的。

2.1.2.3计算机中整数的二进制编码方式

- 无符号数：原码表示

注：实际上，原码也可以将第一位设置为符号位，只不过其只是表示正负，不带有权重。

- 带符号数：补码表示。 $B2T(X) = -x_{w-1} * 2^{w-1} + \sum_{i=0}^{w-2} x_i * 2^i$ ，其中w表示字长。对于补码，最左端的bit (Most Significant Bit) 是符号位。**非负数的补码就是它的原码，负数的补码则是它的相反数的补码按位取反再加一。**

注：反码：一般用来在原码和补码间进行过渡。正数的反码就是它本身，负数的反码则是对其相反数逐位取反，同时保持符号位为1。因此，负数的补码就是它的反码+1。

补码的一个好处在于，在表示有符号的数时，与原码或是反码不同，其表示中不存在+0和-0。

- 无符号数与有符号数的转换： $u(x) = \begin{cases} x & x \geq 0 \\ x + 2^w & x < 0 \end{cases}$ 其中， $u(x)$ 表示无符号数， x 表示相同二进制串表示的有符号数， w 表示字长。特别的，对于负数 x ，在二进制意义下有 $[x]_{\text{补}} = 2^w + x \ (x < 0)$ 。

注：由于 $[x]_{\text{补}} = (-x)_{\text{反}} + 1 \ (x < 0)$ ，因此 $[x]_{\text{补}} - 1 + (-x) = 2^w - 1$ ，也就是 $[x]_{\text{补}} = 2^w + x \ (x < 0)$ 。

- 带符号整数的取值范围（以字长=8为例）：

Type	Decimal	Hex	Binary
UMax	255	FF	1111 1111
TMax	127	7F	0111 1111
TMin	-128	80	1000 0000
-1	-1	FF	1111 1111
0	0	00	0000 0000

2.1.3 整数运算

2.1.3.1 补码加法公式

$$[x]_{\text{补}} + [y]_{\text{补}} \equiv [x + y] \pmod{2^w}$$

注：对 x 、 y 分别讨论大于或小于零的情况即可。

意义：负整数用补码表示后，可以和正整数一样来处理，这样，处理器里的运算器只需要一个加法器就可以实现。

2.1.3.2 C语言中的无符号数与带符号数

对于常数，C语言默认为带符号数；而如果无符号数和带符号数混合使用，则**带符号数默认转换为无符号数**。

示例：

```
/*this code is buggy*/
#define DELTA sizeof(int)
int i;
for (i = cnt; i - DELTA >= 0; i -= DELTA)
    ...
```

注意，由于 `sizeof(int)` 返回的为为无符号整数，因此 `i-DELTA` 会被自动转换为无符号整数，从而循环永远不会终止。

2.1.3.3 无符号数的加法

$$UAdd_w(u, v) = \begin{cases} u + v & u + v < 2^w \\ u + v - 2^w & u + v \geq 2^w \end{cases}$$

补码的加法与无符号数的加法一致，但是存在正溢出和负溢出的问题。

2.1.3.4 移位（除以2的k次幂）

对于无符号数，在移位时会采用逻辑位移；而对于带符号数，在移位时会采用算术位移（移位时在左侧补充的数码与原先的符号位数码相同）。

需要注意的是，对于带符号整数中的负数，在向右移位时会产生舍入错误（其需要向零舍入，而非向下舍入）。但是，我们可以在移位前设置偏置值来修正这种不合适的舍入。因此，当待操作的 $x < 0$ 时，需要给 x 加上 $2^w - 1$ ，其中 w 为将要移位的位数。C表达式的一种写法如下：

```
return (x < 0 ? x + (1 << w) - 1 : x) >> w;
```

计算机执行的“整数”运算实际上是一种模运算的形式，表示数字的有限字长限制了可能的值的取值范围。

2.2. 浮点数

2.2.1 浮点数的标准

IEEE在1985年建立浮点数的标准，用二进制分别表示浮点数的整数部分和小数部分。但也有很大局限性，原因是它只能够表示类似于 $x/2^k$ 这种类型的数据。

二进制小数

考虑一个二进制小数 $b = b_m b_{m-1} \cdots b_1 b_0 . b_{-1} b_{-2} \cdots b_{-n-1} b_{-n}$ $b_i \in \{0, 1\}$

这种表示方法表示的数 b 的定义如下：

$$b = \sum_{i=-n}^{i=m} 2^i * b_i$$

浮点数的数字形式：

IEEE浮点数标准用 $x = (-1)^s M 2^E$ 来表示一个数：

- **符号** (sign) : s 决定这个数是负数还是正数，而对于数值0的符号位解释会作为特殊情况处理。
- **尾数** (significand) : M 是一个二进制小数，它的范围是 $1 \sim 2 - \epsilon$ ，或者是 $0 \sim 1 - \epsilon$ 。
- **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）。

将浮点数的位表示划分为三个字段，分别对这些值进行编码：

- 一个单独的符号位 s 直接编码符号 s 。
- k 位的阶码字段 $exp = e_{k-1} \cdots e_1 e_0$ 编码阶码 E 。
- n 位小数字段 $frac = f_{n-1} \cdots f_1 f_0$ 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。

2.2.1.1 规格化浮点数

这是最普遍的情况。当 exp 的位表达既不全为0，也不全为1时，都属于这种情况。

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。也就是说，阶码的值是 $E = e - Bias$ ，其中 e 是无符号数，而 $Bias$ 是一个等于 $2^{k-1} - 1$ （单精度是127，双精度是1023）的偏置值。由此产生的指数的取值范围，对于单精度是 $-126 \sim 127$ ，对于双精度则是 $-1022 \sim 1023$ 。

小数字段 $frac$ 被解释为描述小数值 f ，其中 $0 \leq f < 1$ 。尾数 M 就定义为 $M = 1 + f$ 。这种方式也叫做隐含的以1开头的表示，原因在于我们总可以通过调整阶码，使得尾数 M 在范围 $[1, 2)$ 之间，这样我们就可以轻松获得一个额外的精度位。

2.2.1.2非规格化浮点数

当阶码域的位表达为全0时，所表示的数为**非规格化**形式。在这种情况下，阶码值是 $E = 1 - bias$ ，而尾数的值是 $M = f$ ，不包含隐含的开头的1。

后文将会提到，这里将阶码值设置为 $E = 1 - bias$ 而不是 $E = -bias$ 是为了让非规格化值平滑转换到规格化值。

非规格化值有两个主要用途：

- 首先，它们提供了一种表示数值0的方法（因为在规格化浮点数中，尾数M始终大于1）。需要注意的是，这种方法表示出来的0由于符号位的不同会产生+0和-0。
- 其次，非规格化浮点数的重要功能在于表示那些非常接近于0.0的数。

2.2.1.3特殊值

这一类数值在阶码的位表达中**全为1**时出现。当小数域全为0时，得到的值表示无穷，符号位为0时表示 $+\infty$ ，符号位为1时表示 $-\infty$ 。当我们把两个非常大的数相乘，或者除以0时，无穷就能够表示这种情况下**溢出**的结果。而当小数域非零时，这一结果被称为“NaN”，即“Not a Number”的缩写。这用来表示一些不被允许的运算（比如 $\sqrt{-1}$ ）以及一些未初始化的数据。

2.2.1.4浮点数示例

以8位浮点数格式为例，其中有4位阶码位和3位小数位，偏置量 $Bias = 2^{4-1} - 1 = 7$ 。

描述	位表示	e	E	2^E	f	M	$2^E \times M$	值	十进制
0	0 0000 000	0	-6	$\frac{1}{64}$	$\frac{0}{8}$	$\frac{0}{8}$	$\frac{0}{512}$	$\frac{0}{512}$	0.0
最小正非规格化数	0 0000 001	0	-6	$\frac{1}{64}$	$\frac{1}{8}$	$\frac{1}{8}$	$\frac{1}{512}$	$\frac{1}{512}$	0.001953
最大非规格化数	0 0000 111	0	-6	$\frac{1}{64}$	$\frac{7}{8}$	$\frac{7}{8}$	$\frac{7}{512}$	$\frac{7}{512}$	0.013672
最小规格化数	0 0001 000	1	-6	$\frac{1}{64}$	$\frac{0}{8}$	$\frac{8}{8}$	$\frac{8}{512}$	$\frac{1}{64}$	0.015625
1	0 0111 000	7	0	1	$\frac{0}{8}$	$\frac{8}{8}$	$\frac{8}{8}$	1	1.0
最大规格化数	0 1110 111	14	7	128	$\frac{7}{8}$	$\frac{15}{8}$	$\frac{1920}{8}$	240	240.0
无穷大	0 1111 000	—	—	—	—	—	—	∞	—

这里即可观察到最大非规格化数到最小规格化数的平滑转变，同时，当数值超过最大规格化数所能表示的值时，就会溢出到 ∞ 。

这种表示具有一个有趣的属性，那就是假如我们将表中的值的位表达式解释为无符号整数，它们就是按照升序排列的，**恰好与它们所表示的浮点数一样**。而当处理负数时，由于位表达式开头有1，因此它们是按照降序出现的。

举例：将整数12345转换为单精度浮点数形式

注意到12345具有二进制表达 $[11000000111001]_2$ ，通过将小数点左移13位，就可以构建这个数的一个规格化表示： $12345 = 1.1000000111001_2 \times 2^{13}$ 。于是其浮点数形式的小数部分应为 $[10000001110010000000000]_2$ 。为了构造阶码，我们将移位数13加上偏置量 $2^7 - 1 = 127$ ，得到140，其二进制表达为 $[10001100]_2$ 。再加上符号位0，即可得到12345的单精度浮点数形式 $[01000110010000001110010000000000]_2$ 。

2.2.2浮点数的舍入

由于浮点数的表示范围和精度受限，所以浮点运算只能近似地表示实数运算。因此，对于一个值 x ，我们有时只能找到最接近的匹配值 x' 来“代表” x ，这个过程就称为**舍入**（Round）。通常的舍入大致有四种方式：向偶数舍入、向零舍入、向下舍入以及向上舍入。这其中，**向偶数舍入**（Round-to-Even）是默认的方式。

为什么选用向偶数舍入呢？如果采用其他三种方式，很容易在计算这些数的平均值等统计过程中引入统计偏差，而向偶数舍入的方式避免了这种情况的发生。

向偶数舍入，也称为**向最近的值舍入**。顾名思义，它会首先将一个值向离它最近的整数值进行舍入。但是，当某一个值恰好落在这样的两个整数之间时，它会优先向**偶数**进行舍入。

一组数向偶数舍入的结果如下：

数值	1.40	1.60	1.50	2.50	-1.50
向偶数舍入	1	2	2	2	-2

相似的，当向二进制小数来应用向偶数舍入法时，我们将**最低有效位的值0认为是偶数，而将值1认为是奇数**。因此，一般来说，只有形如 $[XX \cdots X.YY \cdots Y100 \cdots]_2$ 的二进制位模式才会产生“向偶数舍入”的特殊效果，其中 X 和 Y 表示任意值，最右边的 Y 就是要舍入的位置。原因在于只有这种位模式表示两个可能的结果的中间值。

2.2.3浮点数的运算

IEEE标准中指定浮点数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或者软件实现。

我们把浮点数 x 和 y 看作实数，而某个运算 \otimes 定义在实数上，则定义 $x \otimes y = \text{Round}(x \otimes y)$ 。

值得注意的是，浮点数的运算满足**交换律**，但不满足**结合律**以及**分配律**。例如，使用单精度浮点，表达式 $(3.14 + 1e10) - 1e10$ 求值得到0.0——因为舍入，值3.14会丢失；而另一方面，表达式 $3.14 + (1e10 - 1e10)$ 会得到正确的结果3.14。因此，必须**非常小心地使用浮点数运算**。此外，由于浮点数在溢出时并不会发生正负的颠倒，而是溢出为 $+/ - \infty$ ，因此浮点数具有一些特殊的单调性（无符号数或是补码的乘法则没有这些单调性）：

$$a \geq b \text{ 且 } c \geq 0 \Rightarrow a *^f c \geq b *^f c$$

$$a \geq b \text{ 且 } c \leq 0 \Rightarrow a *^f c \leq b *^f c$$

2.2.4C语言中的浮点数

- 单精度浮点数`float`：`exp`域宽度为8bits，`frac`域宽度为23bits，共32bits。
- 双精度浮点数`double`：`exp`域宽度为11bits，`frac`域宽度为52bits，共64bits。

不同数据间进行强制类型转换的规则：

- `int`转换为`float`：数字不会溢出，但是可能被舍入。
- `int`或`float`转换为`double`：由于`double`的精度更高且范围更大，能够保留精确的数值。
- `double`转换为`float`：数字可能会溢出，还可能会被舍入。

- *float*或*double*转换为*int*: 首先, 值将会向0舍入; 其次, 值可能会溢出, C语言标准对于这种情况会指定固定的结果。如与Intel兼容的微处理器就会指定 $[10 \dots 00]_2 (TMin_w)$ 为这种情况下的**整数不确定值**。一个从浮点数到整数的转换, 如果不能为浮点数找到一个合理的整数近似值, 就会产生这样一个值。例如, `(int)1e10` 就会得到-2147483648, 即 $TMin_{32}$ 。

3.程序的机器级表示

GCC C语言编译器以**汇编代码**的形式产生输出。**汇编代码**是**机器代码**的文本表示, 给出程序中的每一条指令。然后GCC调用**汇编器**和**编译器**, 根据汇编代码生成可执行的机器代码。

机器级程序和它们的汇编代码表示与C程序的差别很大。其中各种数据类型的差别很小, 程序是以指令序列来表示的。相较于高级语言编写的程序, 汇编代码的编译与执行是与特定的机器密切相关的。但是, 能够阅读和理解汇编代码依然是一项很重要的技能。原因在于我们可以理解编译器的优化能力, 分析代码中隐含的低效率, 并进而实现优化程序的目的。

3.1程序编码

3.1.0程序进行编码的过程

```
# Unix命令行编译代码
linux> gcc -Og -o p p1.c p2.c
```

命令gcc指的就是GCC C编译器。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。实际上, 如果指定较高级别的优化等级 (比如`-O1`、`-O2`等), 其产生的汇编代码会严重变形。

实际上, gcc命令调用了一整套的程序, 将源代码转化为可执行代码。首先, **C预处理器拓展源代码**, 插入所有使用`#include`命令指定的文件, 并扩展所有用`#define`声明指定的宏。其次, **编译器**产生两个源文件的**汇编代码**, 名字分别为`p1.s`和`p2.s`。接下来, **汇编器**会将汇编代码转化成**二进制目标代码文件** `p1.o`和`p2.o`。目标代码是机器代码的一种形式, 它包含所有指令的二进制表示, 但是还没有填入全局值的地址。最后, **链接器**将两个目标代码文件与实现库函数 (例如`printf()`) 的代码合并, 并产生最终的可执行代码文件`p` (由命令行指示符`-o p`指定的)。

3.1.1机器级代码

计算机系统使用了多种不同形式的抽象, 其中有两种抽象尤为重要:

- 由**指令集体系结构或指令集架构 (ISA)**来定义机器级程序的格式和行为, 它定义了处理器状态、指令的格式, 以及每条指令对状态的影响。

大多数ISA, 包括x86-64, 将程序的行为描述成好像每条指令都是按顺序执行的。但是, 处理器的硬件结构远比描述的精细复杂, 它们并发地执行每个指令, 但是可以采取**措施**保证整体行为与ISA指定的顺序执行的行为完全一致。

- 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。程序内存用虚拟地址来寻址, 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。

在x86-64的机器代码中, 一些通常对C语言程序员隐藏的处理器状态都是可见的:

- **程序计数器** (通常称为“PC”) 给出将要执行的下一条指令在内存中的地址, 指令寄存器的名称为`%rip`。

- **整数寄存器文件**包含16个命令的位置，分别存储64位的值。这些寄存器存储着地址或整数数据。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。
- **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，比如用来实现*if*和*else*语句。
- 一组向量寄存器可以存放一个或多个整数或浮点数值。

程序的内存包括：程序的可执行机器代码，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（比如调用*malloc*库函数分配的）。在C语言中所实现的模型、数据结构、对象等等，在机器代码中都被用一组连续的字节来表示。汇编代码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。一条机器指令只能执行一个非常基本的操作，例如**将存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。**

3.1.2 汇编代码示例

C语言代码文件*test.c*如下：

```
long mult2(long, long);

void mulstore(long x, long y, long* dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

命令行使用-S选项可得到汇编代码：

```
linux> gcc -Og -S test.c
```

生成的汇编代码入下：

```
.file "test.cpp"
.text
.globl _Z8mulstorellPl
.type _Z8mulstorellPl, @function
_Z8mulstorellPl:
.LFB0:
.cfi_startproc
endbr64
pushq %rbx
.cfi_def_cfa_offset 16
.cfi_offset 3, -16
movq %rdx, %rbx
call _Z5mult2ll@PLT
movq %rax, (%rbx)
popq %rbx
.cfi_def_cfa_offset 8
ret
.cfi_endproc
.LFE0:
.size _Z8mulstorellPl, .-_Z8mulstorellPl
.ident "GCC: (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0"
.section .note.GNU-stack,"",@progbits
.section .note.gnu.property,"a"
.align 8
```

```

.long    1f - 0f
.long    4f - 1f
.long    5
0:
.string  "GNU"
1:
.align  8
.long    0xc0000002
.long    3f - 2f
2:
.long    0x3
3:
.align  8
4:

```

其中，所有以"."开头的行都是指导**汇编器**和**链接器**工作的伪指令，通常我们可以忽略。GCC产生的汇编代码不提供任何程序或是它如何工作的描述，而汇编语言程序员在写代码时通常都会带上注释，简单地描述指令的效果以及它和原始C语言代码中的计算操作的关系。

除以"."开头的行外，每个缩进的行通常都代表一条机器指令。这里：

```

_z8mulstore11P1:
    endbr64
    pushq   %rbx
    movq    %rdx, %rbx
    call    _Z5mult211@PLT
    movq    %rax, (%rbx)
    popq    %rbx
    ret

```

比如，`pushq` 表示将寄存器 `%rbx` 中的内容压入程序栈，`endbr64` 代表64位结束分支，用于确保程序在间接跳转后进入有效位置。

当我们使用“-c”命令行选项 `linux> gcc -Og -c test.c` 时，就会产生二进制目标代码文件 `test.o`。其中的一段18字节序列的十六进制表示“f3 0f 1e fa 53 48 89 d3 e8 00000000 48 80 03 5b c3”就是上面的汇编指令对应的目标代码。

要查看机器代码文件的内容，一种被称为**反汇编器**的程序非常有用。这些程序根据机器代码产生一种类似于汇编代码的格式。Linux系统中，采用带“-d”命令行标志的程序 `OBJDUMP` 可以充当这个角色：

```
linux> objdump -d test.o
```

结果如下：

```

0000000000000000 <_z8mulstore11P1>:
 0:  f3 0f 1e fa          endbr64
 4:  53                   push   %rbx
 5:  48 89 d3             mov    %rdx,%rbx
 8:  e8 00 00 00 00      callq d <_z8mulstore11P1+0xd>
 d:  48 89 03             mov    %rax, (%rbx)
10:  5b                   pop    %rbx
11:  c3                   retq

```

我们可以看到，按照前面给出的字节顺序排列的18个十六进制字节值，且它们分成了若干组，每组有1~5个字节。每组都是一条指令，右边则是等价的汇编语言。其中一些关于机器代码和它反汇编的特性值得注意：

- x86-64的指令长度从1到15个字节不等。指令越常用，操作数越少，所需要的字节数一般来说越少。
- 设计指令格式的方式是，从某个指定位置开始，可以将字节**唯一地**解码成机器指令。
- **反汇编器只是基于机器代码文件中的字节序列来确定汇编代码，它不需要访问该程序的源代码或是汇编代码。**
- 反汇编器使用的指令命名规则和GCC生成的汇编代码使用的有些细微的区别。许多指令结尾的“q”表示大小指示符，有些情况下可以省略。

大多数GCC生成的汇编代码指令都有一个字符的后缀，表明操作数的大小。

C声明	Intel数据类型	汇编代码后缀	大小（字节数）
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

注：64位机器中，指针长8字节。汇编代码使用后缀“l”来表示4字节整数和8字节双精度浮点数并不会产生歧义，因为浮点数使用的是一组完全不同的指令和寄存器。

实际生成可执行文件时，还需要进行**链接**的操作，生成的可执行文件的代码经反汇编后会产生汇编代码。与不经过链接而只通过编译就生成的汇编代码相比，主要的区别是左边列出的**地址**不同，这是因为链接器将这段代码的地址移到了一段不同的地址范围中，且链接器填上了`callq`指令调用的函数的地址。此外，有时在程序末尾还会多处几行十六进制表示为“90”且显示命令为`nop`的行。通常这种操作是为了使得函数代码变为“整字节”（比如16字节），指令实际上对程序没有影响，但就存储器的系统性能而言，能够更好地放置下一个代码块。

注：本节中使用的是ATT格式的汇编代码，这是GCC等工具的默认格式。此外，还有Intel格式的汇编代码，修改命令为 `linux> gcc -Og -S -masm=intel test.c` 即可生成Intel格式的汇编代码。

由于有些机器特性是C程序访问不到的（比如每次x86-64处理器在执行算术或逻辑运算时，如果得到的运算结果的低8位中有偶数个1，就会自动地将一个名为“PF”（“parity flag”奇偶标志）的1位条件码标志设为1，否则设为0）。因此，在程序中插入几条汇编代码指令就能很容易地完成这项任务。插入汇编代码有两种方法：

- 可以编写完整的函数，放进一个独立的汇编代码文件中，再让编译器和链接器把它和用C语言书写的代码合并起来。
- 可以使用GCC的内联汇编特性，用asm伪指令在C程序中包含简短的汇编代码。

3.2 信息的访问

一个x86-64的中央处理单元（CPU）中包含一组16个存储64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**。它们的名字都以%*r*开头，不过后面还跟着一些不同的命名规则的名字。

编号	64位	32位	16位	8位	用途
1	%rax	%eax	%ax	%al	返回值
2	%rbx	%ebx	%bx	%bl	被调用者保存
3	%rcx	%ecx	%cx	%cl	第4个参数
4	%rdx	%edx	%dx	%dl	第3个参数
5	%rsi	%esi	%si	%sil	第2个参数
6	%rdi	%edi	%di	%dil	第1个参数
7	%rbp	%ebp	%bp	%bpl	被调用者保存
8	%rsp	%esp	%sp	%spl	栈指针
9	%r8	%r8d	%r8w	%r8b	第5个参数
10	%r9	%r9d	%r9w	%r9b	第6个参数
11	%r10	%r10d	%r10w	%r10b	调用者保存
12	%r11	%r11d	%r11w	%r11b	调用者保存
13	%r12	%r12d	%r12w	%r12b	被调用者保存
14	%r13	%r13d	%r13w	%r13b	被调用者保存
15	%r14	%r14d	%r14w	%r14b	被调用者保存
16	%r15	%r15d	%r15w	%r15b	被调用者保存

如上表所示，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，可以访问**最低的字节、最低的两字节、最低的四字节、整个寄存器**。当操作少于8个字节时，有两条规则：

1. 生成1字节或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**。

3.2.1 操作数

大多数指令都有一个或多个**操作数**（operand），指示出执行一个操作中所要使用的**源数据**，以及放置结果的**目的位置**。由于源数据可以以常数形式给出，也可以从寄存器或是内存中读出，因此操作数一般被分为3类：

- **立即数**（immediate）：用来表示常数值。ATT格式的汇编代码中，立即数的书写方式为“\$”后加上一个用**标准C表示法表示的整数**，如\$-577以及\$0x1F。

如果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。

- **寄存器**（register）：用来表示某个寄存器的内容。我们用符号 r_a 来表示任意寄存器a，用**引用** $R[r_a]$ 来表示它的值。这里是将寄存器集合看成一个数组R，用寄存器标识符作为索引。

- **内存引用 (memory)**：根据计算出来的**有效地址**来访问某个内存位置。我们将内存看成一个很大的字节数组，用符号 $M_b[Addr]$ 来表示对存储在内存中从地址 $Addr$ 开始的 b 个字节值的引用。通常可以省略下标 b 。

在实际应用中， $Imm(r_b, r_i, s)$ 是最常用的寻址模式。这样的引用由4个部分组成：一个**立即数偏移** Imm ，一个**基址寄存器** r_b ，一个**变址/索引寄存器** r_i 和一个**比例因子** s 。这里 s 必须是1, 2, 4或者8。基址或是变址寄存器都必须是64位寄存器。有效地址被计算为 $Imm + R[r_b] + R[r_i] \times s$ 。引用数组元素时，就会用到这种通用形式。其他形式都是省略了某些部分的特殊情况。

特别注意，对于大部分指令，当**操作数中包含括号，或是仅仅有一个立即数（前面没有“\$”符号）**，都表示在对应的内存中读取值。当操作数为单独的一个寄存器时，表示寄存器寻址，取出对应寄存器中的值。

基址寄存器和变址寄存器可以理解为，基址寄存器中保存一个数组的起始位置，变址寄存器中保存某一元素与起始位置的相对偏移（下标），二者相加得到真实的元素地址。

3.2.2 数据传送指令

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。最简单形式的传送指令为MOV类。

MOV类由四条指令组成，包括movb、movw、movl和movq（注意这里的MOV均为拷贝粘贴，而非剪切粘贴）。这些指令所能执行的操作都是相同的，主要区别在于它们操作的数据大小不同，分别是1, 2, 4, 8和8字节。值得注意的是，只有最后一条指令是处理64位立即数数据的。常规的movq只能以表示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。movabsq则能够以任意64位立即数值作为源操作数，并且只能以寄存器作为目的地。

在MOV指令中，**源操作数**指定的值是一个立即数，存储在寄存器中或者内存中。**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。此外，x86-64加入了一条限制：**传送指令的两个操作数不能都指向内存地址**。因此，将一个值从一个内存地址复制到另一个内存地址需要两条指令。当然，寄存器部分的大小必须与指令最后一个字符所指定的大小匹配。需要注意的是，movl指令在以寄存器作为目的时，它会把该寄存器的高位4字节设置为0。

x86-64的惯例：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为0。

在将较小的源值复制到较大的目的时，还可以使用另两类数据移动指令，分别是MOVZ类和MOVS类。二者的区别在于，前者把目的中剩余的字节填充为0，后者则通过符号位扩展来填充。MOVS类有一条特殊的命令：cltq，它没有操作数，总是以寄存器%eax作为源，%rax作为符号扩展结果的目的。它的效果与movslq %eax %rax 完全一致，不过编码更加紧凑。

在数据的传输比如函数调用过程中，比如函数的调用与返回，**局部变量通常是保存在寄存器中，而不是内存中**（访问寄存器比访问内存要快得多）；并且C语言中所谓的“指针”实际上就是地址，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。

3.2.3 压入和弹出栈数据

栈是一种遵循“**后进先出**”规则的数据结构，通过push操作将数据压入栈中，通过pop操作删除数据；它具有一个属性：弹出的值始终是最近被压入且仍然在栈中的值。插入或删除元素的一端称为**栈顶**，栈顶元素的地址是所有栈中元素地址中**最低的**。**栈指针%rsp**保存着栈顶元素的地址。

入栈和出栈指令如下（以插入、弹出四字数据为例）：

指令	效果	描述	等价汇编代码
pushq S	$R[\%rsp] \leftarrow R[\%rsp] - 8$ $M[R[\%rsp]] \leftarrow S$	将四字压入栈	<code>subq \$8, %rsp</code> <code>movq %rbq, (%rsp)</code>
popq D	$D \leftarrow M[R[\%rsp]]$ $R[\%rsp] \leftarrow R[\%rsp] + 8$	将四字弹出栈	<code>movq (%rsp), %rax</code> <code>addq \$8, %rsp</code>

此外，由于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，所以程序可以通过标准的内存寻址方式来访问栈中的任意位置上的元素。例如，上表中，指令“`movq 8(%rsp), %rdx`”就会将第二个四字从栈中复制到寄存器%rdx中。

3.3 算术和逻辑操作

算术和逻辑操作被分为四组，分别为**加载有效地址**、**一元操作**、**二元操作**和**移位**。所有指令类都有各种带有不同大小操作数的变种（只有leaq没有其他大小的变种），例如addb、addw、addl、addq分别表示字节加法、字加法、双字加法和四字加法。各种整数算术运算的指令如下：

指令	效果	描述
<code>leaq S, D</code>	$D \leftarrow \&S$	加载有效地址
<code>INC D</code>	$D \leftarrow D + 1$	加一
<code>DEC D</code>	$D \leftarrow D - 1$	减一
<code>NEG D</code>	$D \leftarrow -D$	取负
<code>NOT D</code>	$D \leftarrow \sim D$	取补
<code>ADD S, D</code>	$D \leftarrow D + S$	加
<code>SUB S, D</code>	$D \leftarrow D - S$	减
<code>IMUL S, D</code>	$D \leftarrow D * S$	乘
<code>XOR S, D</code>	$D \leftarrow D \wedge S$	异或
<code>OR S, D</code>	$D \leftarrow D S$	或
<code>AND S, D</code>	$D \leftarrow \&S$	与
<code>SAL k, D</code>	$D \leftarrow D \ll k$	左移
<code>SHL k, D</code>	$D \leftarrow D \ll k$	左移（等同于SAL）
<code>SAR k, D</code>	$D \leftarrow D \gg_A k$	算术右移
<code>SHR k, D</code>	$D \leftarrow D \gg_L k$	逻辑右移

3.3.1 加载有效地址

加载有效地址的指令leaq实际上是movq指令的变形。该指令可以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。此外，它还可以**简洁地**描述普通的算术操作。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 $5x + 7$ 。

注意，如果这里的指令为`movq 7(%rdx, %rdx, 4), %rax`，会将%rax的值设为 $M[5x + 7]$ 。这即为movq指令与leaq指令的区别。

3.3.2 一元和二元操作

第二组的操作是一元操作，即**只有一个操作数**，既是源又是目的。

第三组是二元操作，即**有两个操作数**，其中第二个操作数既是源又是目的。例如，指令 `subq %rax, %rdx` 将使寄存器 `%rdx` 的值减去 `%rax` 的值。值得注意的是，当第二个操作数为内存地址时，处理器必须从内存中读出值，执行操作，再把结果写回内存，这样就增加了时间上的消耗。

3.3.3 移位操作

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。移位量可以是一个**立即数**，或者放在**单字节寄存器** `%cl` 中。注意，虽然一个字节的移位量使得移位量的编码范围可以得到 $2^8 - 1 = 255$ ，但是 x86-64 中，当移位操作对 w 位长的数据值进行操作时，移位量是由 `%cl` 寄存器的低 m 位决定的，这里 $2^m = w$ ，高位会被忽略。

左移指令 `SAL` 和 `SHL` 分别表示**算术左移**与**逻辑左移**，从效果来说没有区别，都是在右侧补零。而右移指令 `SAR` 和 `SHR` 分别执行**算术移位**和**逻辑移位**，`SAR` 在左侧补的是符号位。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。

可以看到，大部分的指令既可以用于无符号运算，也可以用于补码运算，只有右移操作要求区分有符号数和无符号数。这个特性使得补码运算成为实现有符号整数运算的一种比较好的方法。

3.3.4 特殊的算术操作

我们知道，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，Intel 把 16 字节的数称为八字（oct word）。

一些特殊的算术操作如下：

指令	效果	描述
<code>imulq S</code>	$R[\%rdx] : R[\%rax] \leftarrow S \times R[\%rax]$	有符号全乘法
<code>mulq S</code>	$R[\%rdx] : R[\%rax] \leftarrow S \times R[\%rax]$	无符号全乘法
<code>cqto</code>	$R[\%rdx] : R[\%rax] \leftarrow \text{符号扩展}(R[\%rax])$	转换为八字
<code>idivq S</code>	$R[\%rdx] \leftarrow R[\%rdx] : R[\%rax] \bmod S$ $R[\%rax] \leftarrow R[\%rdx] : R[\%rax] \div S$	有符号除法
<code>divq S</code>	$R[\%rdx] \leftarrow R[\%rdx] : R[\%rax] \bmod S$ $R[\%rax] \leftarrow R[\%rdx] : R[\%rax] \div S$	无符号除法

由前文我们可以看到，`imulq` 有一种形式是作为 `IMUL` 指令类中的一种，这种形式的 `imulq` 指令是一个**“双操作数”**乘法指令，从两个 64 位操作数产生一个 64 位乘积。而在这里，x86-64 指令集还提供了两条不同的**“单操作数”**乘法指令，以计算两个 64 位值的全 128 位乘积——一个是无符号数乘法（`mulq`），而另一个是补码乘法（`imulq`）。这两条指令都要求**一个参数必须在寄存器 `%rax` 中**，而**另一个作为指令的源操作数**给出，然后把乘积存放在寄存器 `%rdx`（高 64 位）和 `%rax`（低 64 位）中。汇编器可以通过 `imulq` 后的操作数数目来分辨想用哪条指令。对于除法指令同理，`idivq` 和 `divq` 会将寄存器 `%rdx`（高 64 位）和 `%rax`（低 64 位）中的 128 位数作为被除数，而除数则由指令中的操作数来给出。指令将商存储在寄存器 `%rax` 中，而将余数存储在寄存器 `%rdx` 中。

当然，对于大部分的除法操作来说，被除数应当是一个 64 位的值，因此寄存器 `%rdx` 中的值应置为**全零**（无符号）或是**符号位**（有符号）。这个操作是通过指令 `cqto` 来完成。指令 `cqto` 不需要操作数，它会隐含读出 `%rax` 的符号位。对于无符号数，这一操作通常通过指令 `movq $0, %rdx` 来实现。

乘法的示例：代码如下：


```
#include <inttypes.h>

typedef unsigned __int128 uint128_t;

void store_uprod(uint128_t *dest, uint64_t x, uint64_t y)
{
    *dest = x * (uint128_t) y;
}
```

生成的汇编代码如下：

```
# dest in %rcx, x in %rdx, y in %r8
_Z11store_uprodPoyy:
    movq    %rdx, %rax    #copy x to multiplicand
    mulq    %r8           #multiply x by y
    movq    %rax, (%rcx)  #store lower 8 bytes at dest
    movq    %rdx, 8(%rcx) #store upper 8 bytes at dest+8(Little Endian)
    ret
```

3.4控制

对于C语言中的某些结构，比如条件语句、循环语句等，要求有条件的执行，**根据测试的结果来决定操作执行的顺序**。机器代码提供两种基本的低级机制来实现有条件的行为。

通常，C语言中的语句和机器代码中的指令都是按照它们在程序中出现的次序**顺序执行**。用*jump*指令即可以改变一组机器代码指令的执行顺序，指定控制被传递到程序的某个其他部分。

3.4.1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**，它们描述了最近的算术或逻辑操作的属性。最常用的条件码有：

- *CF*：进位标志。最近的操作使最高位产生了进位。可用来检查无符号数的溢出。
- *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。
- *OF*：溢出标志。最近的操作导致一个补码溢出（包括正溢出和负溢出）。

由于*leaq*指令是用来进行地址计算的，因此它不会改变任何条件码。除此之外，涉及整数运算的所有指令都会设置条件码。对于逻辑操作，例如*XOR*，进位标志和溢出标志会设置成0；对于移位操作，进位标志将设置为最后一个被移出的位（左移时，储存的为最低位；右移时相反），而溢出标志将被设置为0；*INC*和*DEC*会设置溢出和零标志，但是不会改变进位标志。

除了涉及整数运算的指令之外，还有两类指令*CMP*和*TEST*，它们**只设置条件码而不改变任何其他寄存器**。它们的指令如下

$$CMP \ S_1, S_2 \quad \text{基于 } S_2 - S_1 \text{ 进行比较}$$

$$TEST \ S_1, S_2 \quad \text{基于 } S_1 \& S_2 \text{ 进行测试}$$

注意到，在ATT格式中，列出的操作数的顺序是相反的，如果两个操作数相等，这些*CMP*指令会将零标志设置为1，而其他的标志可以用来确定两个操作数之间的大小关系。*TEST*指令的行为与*AND*指令一样，除了它只改变条件码。典型用法是，设置两个操作数一样，利用*TEST*可判断操作数为正值、0还是负值。

条件码一般不会直接访问，常用的方法有三种：

- 可以根据条件码的某种组合，将一个字节设置为0或1。
- 可以条件跳转到程序的某个其他的部分。

- 可以有条件地传送数据。

这一整类指令被称为 *SET* 指令，这些指令名字的不同后缀指明了它们所考虑的**条件码的组合**，而不是操作数大小。一条 *SET* 指令的目的操作数必须是**低位单字节寄存器元素之一**，或是一个字节的内存位置，指令会将这个字节设置为0或1。需要特别注意的是，在进行大小比较时，对于有符号与无符号整数，指令是不同的。例如，指令 *setl* 表示有符号数的“小于”比较，而指令 *setb* 表示无符号数的“低于”比较。而对于有符号和无符号整数通用的指令有 *sete*、*setne*、*sets*、*setns* 等，分别用来判断相等、不等、负数、非负数。

注：（一个例子）指令 *setl*（用于比较有符号数的小于）的效果如下

$$D \leftarrow SF \wedge OF$$

当没有发生溢出时，OF为0（OF设置为0就表示无溢出），我们有 $a - {}^t_w b < 0$ 时 $a < b$ ，当且仅当SF被设置为1时成立；另一方面，当发生溢出时，OF为1，显然这时只有下溢才满足 $a < b$ ，而发生溢出且为下溢时当且仅当 $a - {}^t_w b > 0$ ，也即SF被设置为0。因此，溢出和符号位的条件码的异或提供了 $a < b$ 是否为真的测试。

3.4.2 跳转指令

跳转指令会导致执行切换到程序中一个全新的位置，在汇编代码中，这些跳转的目的地通常用一个**标号**指明。

跳转指令分为**条件跳转**和**无条件跳转**。*jmp*指令就是无条件跳转，但是它可以是**直接跳转**，即跳转目标是作为指令的一部分编码的；或是**间接跳转**，即跳转目标是从寄存器或内存位置中读出的。直接跳转的跳转目标是直接作为标号给出的，例如 *jmp .L1*；间接跳转的写法是“*”后跟一个操作数指示符，**按照操作数格式进行寻址**。例如，指令“*jmp *%rax*”指用寄存器 *%rax* 中的值作为跳转目标，而指令“*jmp *(%rax)*”则以 *%rax* 中的值作为读地址，从内存中读出跳转目标。

条件跳转的指令有很多，它们根据条件码的某种组合，或者**跳转**，或者**继续执行代码序列中的下一条指令**。这些指令的**名字和跳转条件**与 *SET* 指令的名字和设置条件时相匹配的。**条件跳转必须为直接跳转**。

跳转指令的编码方式有两种，分别为**PC相对的**和**绝对地址编码**。前者会将**目标指令的地址与紧跟在跳转指令后面那条指令的地址之间的差值**作为编码，地址偏移量可以编码为1, 2或4字节。后者则是给出绝对地址，用4个字节直接给出目标。PC相对的进行编码的优势在于，其指令编码很简洁，且程序链接后，目标代码可以不做改变就移到内存中的不同位置。

例：下面是跳转指令的一个示例，跳转目标的编码是PC相对的，并且是一个按照小端法编码的4字节补码数。

```
4005eb: e9 73 ff ff ff      jmpq
4005ed: 90                  nop
```

则跳转目标的地址应为0x400560。

3.4.3 实现条件分支

3.4.3.1 用条件控制实现条件分支

C语言中的 *if - else* 语句的通用形式模板如下：

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

对于这种通用形式，汇编是按通常会使用下面这种形式，这里，使用C语法来描述控制流：

```

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

```

也就是说，编译器为`then-statement`和`else-statement`产生各自的代码块，并插入条件和无条件分支。

3.4.3.2用条件传送来实现条件分支

在3.4.3.1中，程序采用“切换路径”的方式来实现条件分支。这种机制简单通用，但是在现代处理器上可能会非常低效。因此，可以采用一种替代的策略：**数据的条件转移**。简单来说，这种方法计算条件操作的两种结果，然后再根据条件是否满足来从中选取一个。只有在一些受限制的情况中，这种策略才可行，但是如果可行，就可以用一条**简单的条件传送指令**来实现它，其更符合现代处理器的性能特征。

条件传送的一个示例如下：

```

long absdiff(long x, long y){
    long result;
    if (x < y)
        result = y - x;
    else
        result = x - y;
    return result;
}

```

对应汇编代码：

```

# x in %rdi, y in %rsi
absdiff:
    movq    %rsi, %rax
    subq    %rdi, %rax        # rax = y - x
    movq    %rdi, %rdx
    subq    %rsi, %rdx        # rdx = x - y
    comq    %rsi, %rdi        # Compare x:y
    cmovge  %rdi, %rax        # if x >= y, rax = rdx
    ret

```

汇编代码中的关键指令`cmovge`即实现了**条件赋值**，并且注意到，汇编代码会**按照顺序**先执行第一个分支的指令再执行第二个分支的指令。

如果使用C语法来描述控制流，就应该为：

```

long absdiff(long x, long y){
    long result = y - x;
    long rresult = x - y;
    if (x >= y)
        result = rresult;
    return result.
}

```

注：为什么**基于条件数据传送的代码**会比**基于条件控制转移的代码**性能好？

这是因为，现代处理器通过**流水线**来获得高性能。在流水线中，一条指令的处理要经过一系列的阶段，每个阶段执行所需操作的一小部分（例如，从内存中取指令、确定指令类型、从内存中读取数据、执行算术运算、向内存写数据、更新程序计数器等等）。这种方法通过**重叠连续指令**的步骤来获得高性能。例如，在取一条指令的**同时**，执行它前面一条指令的算术运算。但是，要做到这一点，就要求能够**事先确定要执行的指令序列**，这样才能保持流水线中充满了待执行的指令。当机器遇到（条件）分支时，只有当条件分支求值完成后，才能决定分支往哪走。现代处理器采用非常精密的**分支预测逻辑**来猜测每条跳转指令是否会执行。现代微处理器设计试图达到90%以上的正确率，只要它的猜测比较可靠，指令流水线中就会充满着指令；另一方面，如果错误预测一个跳转，就要求处理器丢掉它为该跳转指令后所有指令已做的工作，然后再开始用从正确位置处起始的指令去填充流水线，这样就会浪费大约15~30个时钟周期，导致程序性能严重下降。而基于条件数据传送的控制流不依赖于数据，这使得处理器更容易保持流水线是满的。

与跳转指令类似，x86-64上的**条件传送指令**的名字和传送条件与**SET**指令的名字和设置条件是相匹配的，指令的结果取决于对应条件码的值。每条条件传送指令都有两个操作数，分别为源寄存器或者内存地址S，以及目的寄存器R。这其中，源和目的的值可以是16位、32位或64位长，但是不支持单字节的条件传送，**汇编器可以从目标寄存器的名字推断出条件传送指令的操作数长度**，因此对所有的操作数长度可以共用同一个指令名字。

当然，前面已经提到，**不是所有的条件表达式都可以用条件传送来进行编译**。因为，无论测试结果如何，我们会对“两个分支”的结果都进行求值。如果这两个表达式的其中一个可能产生错误条件或是副作用，就会导致非法的行为。例如，考虑这个代码 `long cread(long *xp) {return (xp ? *xp : 0);}`。此外，使用条件传送也不总是能提高代码的效率。如果两个表达式中需要进行大量的计算，就会导致浪费，因此，编译器必须考虑**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**。对于GCC的实验表明，只有当两个表达式都很容易计算时，例如都只是一条加法指令，编译器才会使用条件传送。

3.4.4 循环

C语言程序中提供了多重循环结构，即 `do-while`、`while` 和 `for`。汇编中没有相应的指令存在，但可以用**条件测试和跳转**组合起来实现循环的效果。

3.4.4.1 `do - while` 循环

C语言示例如下：

```
long fact_do(long n){
    long result = 0;
    do{
        result *= n;
        n -= 1;
    } while(n > 1);
    return result;
}
```

GCC产生的汇编代码如下：

```
fact_do
    movl    $1, %eax        # set result = 1
.L2:                                # loop:
    imulq   %rdi, %rax       # compute result *= n
    subq    $1, %rdi        # compute n -= 1
    cmpq    $1, %rdi        # compare n : 1
    jg      .L2             # if >, goto loop
    rep; ret                # return
```

注：汇编代码中的 `rep`（或 `repz`）是为了避免 `ret` 指令成为条件跳转指令的目标。根据Intel和AMD的文档中的说明，当 `ret` 指令是通过跳转指令到达时，处理器不能正确预测 `ret` 指令的目的。这里的 `rep` 指令就作为一种**空操作**，因此作为跳转目的而插入它。

汇编代码中，倒数第二行的 `jg` 指令是实现循环的关键指令，它决定了是需要继续重复还是退出循环。

注：编译器的“奇怪”行为：C语言编译器常常会重组计算，因此有些C代码中的变量在机器代码中没有对应的值；而有时，机器代码中又会引入源代码中不存在的新值；此外，编译器还常常试图将多个程序值映射到一个寄存器上，以此来**最小化寄存器的使用率**。

3.4.4.2 *while* 循环

有很多种方法将 `while` 循环翻译成机器代码，GCC在代码生成中使用其中的两种方法。

1. 第一种翻译方法，被称为“**跳到中间**”，它执行一个无条件跳转到循环结尾处的测试，以此来执行**初始的测试**。可以用以下模板来表达：

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

2. 第二种翻译方法，被称为“**guarded-do**”。它首先用条件分支，如果初始条件不成立就跳过循环，把代码变换为***do-while***策略。当使用较高优化等级编译时，例如使用命令行选项 `-O1`，GCC就会采取这种策略。**利用这种实现策略，编译器常常可以优化初始的测试，例如在确保初始测试满足的情况下将其省去**。可以用以下模板来表达：

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

3.4.4.3 *for* 循环

由于任意一个 `for` 循环相当于在 `while` 循环前加上一个初始化表达式，在每次循环结束时加上一个更新表达式，因此GCC为 `for` 循环产生的代码时 `while` 循环的两种翻译之一，这取决于优化的等级。特别地，在C语言中执行 `continue` 语句将会导致程序直接跳到当前循环的末尾（注意不可直接跳过更新表达式），执行 `break` 语句则会导致程序直接跳出循环。

3.4.5 *switch* 语句

`switch`（开关）语句可以根据一个**整数索引值**进行**多重分支**。在处理具有多种可能结果的测试时，这种语句不仅提高了C代码的可读性，而且通过使用**跳转表**这种数据结构使得实现更加高效。和使用一组很长的 `if-else` 语句相比，使用跳转表的优点是**执行开关语句的时间与开关情况的数量无关**。GCC会根据开关情况的数量和开关情况值的稀疏程度来翻译开关语句。当开关情况比较多（4个以上）并且值的范围跨度比较小时，就会使用跳转表。

下面是一个带有 `switch` 语句的C代码示例：

```

void switch_eg(long x, long n, long *dest){
    long val = x;
    switch(n)
    {
        case 100:
            val *= 13;
            break;
        case 102:
            val += 10;
            /* Fall through */
        case 103:
            val += 11;
            break;
        case 104:
        case 106:
            val *= val;
            break;
        default:
            val = 0;
    }
    *dest = val;
}

```

GCC生成的汇编代码如下：

```

switch_ed:
    subq    $100,%rsi
    cmpq    $6,%rsi
    ja      .L8
    jmp     *.L4(,%rsi, 8)
# jump table
    .section .rodata
    .align  8    # align address to multiple of 8
.L4:
    .quad   .L3
    .quad   .L8
    .quad   .L5
    .quad   .L6
    .quad   .L7
    .quad   .L8
    .quad   .L7
# end jump table
.L3:
    leaq    (%rdi,%rdi,2),%rax
    leaq    (%rdi,%rax,4),%rdi
    jmp     .L2
.L5:
    addq    $10,%rdi
.L6:
    addq    $11,%rdi
    jmp     .L2
.L7:
    imulq   %rdi,%rdi
    jmp     .L2
.L8:
    movl    $0,%edi

```



```
.L2:
    movq    %rdi, (%rdx)
    ret
```

可以看到，编译器首先将 n 减去100，把取值范围移到0和6之间（通常情况下，编译器可能会尝试将最小取值转化为0），并且利用无符号值，进一步简化了分支的可能性。执行`switch`语句的关键步骤是通过跳转表来访问代码位置，通过汇编代码中的 `jmp *.L4(,%rsi, 8)` 来实现。这是一个**间接跳转**，操作数指定一个内存位置，索引由寄存器`%rsi`给出，这个寄存器保存着 $(n - 100)$ 的值。跳转表的声明在代码中由注释标出，这段声明表示，在叫做`.rodata`（只读数据，*Read - Only Data*）的目标代码文件的段中，应该有一组7个“四”字（8个字节），每个字的值都是与指定的汇编代码标号相关联的指令地址。标号`.L4`标记出这个分配地址的起始。与这个标号相对应的地址会作为间接跳转的基地址。值得注意的是，由于C代码中 `case 102` 中没有 `break` 语句，因此汇编代码中对应部分（.L5）就没有跳转语句，当执行完后其会继续执行下一个块。

3.5过程

过程是软件中一种很重要的**抽象**。它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了某种功能。过程的形式多样，包括**函数**（function）、**方法**（method）、**子例程**（subroutine）、**处理函数**（handler）等等。要提供对过程的机器级支持，在例如“P过程调用Q，Q执行后返回到P”这一动作中包括下面一个或多个机制：

1. **传递控制**：进入过程Q时，**程序计数器**（寄存器`%rip`）必须被设置为Q的代码的**起始地址**，然后在返回时，要把程序计数器设置为P中调用Q后面那条指令的地址。
2. **传递数据**：P必须能够向Q提供一个或多个**参数**，Q能够向P返回一个**值**。
3. **分配和释放内存**：在开始时，Q可能需要为局部变量**分配空间**，而在返回前，又必须**释放**这些存储空间。

3.5.1运行时栈

C语言过程调用机制的一个关键特性在于使用了栈数据结构提供的**后进先出**的内存管理原则，这一原则恰好与C语言标准的**调用/返回机制**相吻合。x86-64的栈向**低地址**方向增长，而**栈指针`%rsp`**指向**栈顶元素**。可以用`pushq`或`popq`指令将数据存入栈中或是从栈中取出。将栈指针**减少**一个量即可以为数据在栈上分配空间；类似地，可以通过**增加**栈指针来释放空间。

当x86-64过程需要的存储空间超过寄存器能够存放的大小时，就会在栈上分配空间，这个部分称为过程的**栈帧**。在这个空间中，过程可以保存寄存器的值、分配局部变量空间、为它调用的过程设置参数。大多数过程的栈帧都是定长的，在过程的开始就分配好了。而实际上，许多函数甚至根本不需要栈帧。当所有的局部变量都可以保存在寄存器中（通过寄存器，过程可以传递最多6个整数值，也就是指针和整数），而且该函数不会调用任何其他函数时，就可以这么处理。

3.5.2转移控制

将控制从函数P转移到函数Q只需要简单地把**程序计数器**（PC）设置为Q的代码的起始位置。不过，稍后从Q返回时，处理器必须记录好它需要继续P的执行的代码位置。这个信息是用指令`call Q`调用过程Q来记录的。**该指令会将返回地址压入栈中，并将PC设置为Q的起始地址**。这里的返回地址是紧跟在`call`指令后面的那条指令的地址。对应的函数返回指令`ret`会**从栈中弹出返回地址，并将PC设为它**。注意，由于返回地址中是与P相关的状态，因此我们把这个返回地址当作**P栈帧的一部分**。

同跳转一样，调用可以是**直接**的，也可以是**间接**的。在汇编代码中，直接调用的目标是一个标号，而间接调用的目标是`*`后面跟一个操作数指示符。

3.5.3 数据传送

x86-64中，大部分过程间的数据传送是通过**寄存器**实现的。例如，在函数调用时，参数可以在寄存器`%rdi`、`%rsi`和其他寄存器中传递。当过程P调用过程Q时，P的代码必须首先把参数复制到适当的寄存器中；当Q返回到P时，P的代码可以访问寄存器`%rax`中的返回值。x86-64中最多可以通过寄存器传递6个整型变量，且寄存器的使用是有特殊顺序的，一般来说会按照以下顺序进行分配：`%rdi`、`%rsi`、`%rdx`、`%rcx`、`%r8`、`%r9`（操作数大小为64位）。

如果一个函数有大于6个整型参数，则超过6个的部分就要通过栈来传递。假设过程P调用过程Q，且需要传递的参数个数为 $n > 6$ 。首先，P的栈帧必须要能够容纳7到n号参数的存储空间。函数调用时，首先要把参数1~6复制到对应的寄存器，把参数7~n放到栈上，而参数7位于栈顶。**当通过栈传递参数时，所有的数据大小都向8的倍数对齐。**

一个参数传递的示例：

```
void proc(long a1, long *a1p,
          int a2, int *a2p,
          short a3, short *a3p,
          char a4, char *a4p)
{
    *a1p += a1;
    *a2p += a2;
    *a3p += a3;
    *a4p += a4;
}
```

GCC产生的汇编代码如下：

```
# a1 in %rdi, a1p in %rsi, a2 in %edx, a2p in %rcx, a3 in %r8w, a3p in %r9
# a4(8 bits) in %rsp+8, a4p(64 bits) in %rsp + 16
proc:
    movq    16(%rsp), %rax
    addq    %rdi, (%rsi)
    addl    %edx, (%rcx)
    addw    %r8w, (%r9)
    movl    8(%rsp), %edx
    addb    %dl, (%rax)
    ret
```

可以看到，其中a4是通过`8(%rsp)`的形式读出，a4p则是通过`16(%rsp)`的形式读出。

3.5.4 栈上的局部存储

某些情况下，一些局部数据不可以存放在寄存器中，而必须存放在内存中，一般有以下几种情况：

- 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“&”**，因此必须能够为它产生一个地址。
- 某些局部变量是**数组或结构**，因此必须能够通过数组或结构引用被访问到。

若对某个局部变量使用地址运算符，并将其存入栈上的空间来进行传递，则这种情况下通常需要用到**leaq指令**。例如，指令`movq $1, 8(%rsp); leaq 8(%rsp), %rdi`就将一个值为1的局部变量存在栈上距离栈顶8字节的地方，并将其**地址**传给寄存器`%rdi`，这样函数在调用`%rdi`的值时便相当于调用了一个指向这一局部变量的指针。在执行这一类操作的前后，程序需要分别**分配和释放**栈空间。

特性：可以直接访问不超过当前栈指针（`%rsp`）128字节的栈上空间。

3.5.5 寄存器中的局部存储

寄存器组是唯一被所有过程共享的资源。虽然在任何给定时刻只有一个过程是活动的，我们仍然必须确保当一个过程（**调用者**）调用另一个过程（**被调用者**）时，被调用者不会覆盖调用者稍后会使用的寄存器值。为此，x86-64采用了一组统一的寄存器使用惯例。

寄存器`%rbx`、`%rbp`和`%r12 ~ %r15`被划分为**被调用者保存寄存器**。所有其他的寄存器，除了栈指针`%rsp`外，都分类为**调用者保存寄存器**。对于**被调用者保存寄存器**，当过程P调用过程Q时，Q作为**被调用者**必须保存这些寄存器的值，要么就是**不去改变它**，要么就是**把原始值压入栈中，改变寄存器的值，然后在返回前从栈中弹出旧值**。压入栈中的值会在栈帧中创建标号为“**被保存的寄存器**”的一部分。有了这条惯例，P的代码就可以安心地把值保存在**被调用者保存寄存器**中。而对于**调用者保存寄存器**，意味着任何函数都能修改它们。

这里应该这样理解**调用者保存**：意指**调用者**有责任保存好这一部分数据，无论其他函数怎样修改这些寄存器，如果需要，调用者都应当可以自行恢复。**被调用者保存**同理。

3.5.6 递归过程

前文中叙述的寄存器和栈的惯例使得x86-64过程能够**递归**地调用它们自身。一个递归调用自身的例子如下：

```
long rfact(long n){
    long result;
    if (n <= 1)
        result = 1;
    else
        result = n * rfact(n - 1);
    return result;
}
```

GCC产生的汇编代码如下：

```
# n in %rdi
rfact:
    pushq    %rbx                # save %rbx(上一次调用过程中的值)
    movq     %rdi, %rbx          # store n in callee-saved register %rbx
    movl     $1, %eax            # set return value = 1
    cmpq     $1, %rdi            # compare n : 1
    jle      .L35                # if <=, goto done
    leaq     -1(%rdi), %rdi       # compute n - 1
    call     rfact               # call rfact(n-1)
    imulq    %rbx, %rax           # multiply n by rfact(n-1)
.L35:                          # done:
    popq     %rbx                # restore %rbx
    ret                                # return
```

事实上，递归调用一个函数本身和调用其他函数是一样的。栈规则提供了一种机制，每次函数调用都有它自己私有的状态信息（**保存的返回值位置和被调用者寄存器保存的值**）存储空间。**栈分配和释放**的规则很自然地就与函数**调用-返回**的顺序匹配。

3.6数组分配和访问

C语言中的数组是一种将标量数据聚集成更大数据类型的方式。C语言实现数组的方式的方式非常简单，使得其很容易被翻译成机器代码。并且，优化编译器非常善于**简化数组索引所使用的地址计算**。

3.6.1基本原则与指针的运算

C语言允许对指针进行运算，并且，计算出来的值会根据该指针引用的数据类型的大小进行**伸缩**。

对于数据类型T和整形常数N，以及数组的声明“ $T \ A[N];$ ”，表示首先在内存中分配一个 $L \cdot N$ 字节的**连续区域**，这里L即为数据类型T的大小（单位为字节）；其次，它引入了标识符A，可以用A来作为**指向数组开头的指针**，这个指针的值设为 x_A 。

对于C语言中的指针运算，假设int型数组E的**起始地址**和**整数索引**分别存放在寄存器%rsx和%rcx中。下面是一些与E有关的表达式以及对应汇编代码的示例：

表达式	类型	值	汇编代码
E	int^*	x_E	<code>movq %rdx, %rax</code>
$E[0]$	int	$M[x_E]$	<code>movl (%rdx), %eax</code>
$E[i]$	int	$M[x_E + 4i]$	<code>movl (%rdx, %rcx, 4), %eax</code>
$\&E[2]$	int^*	$x_E + 8$	<code>leaq 8(%rdx), %rax</code>
$E + i - 1$	int^*	$x_E + 4i - 4$	<code>leaq -4(%rdx, %rcx, 4), %rax</code>
$*(E + i - 3)$	int	$M[x_E + 4i - 12]$	<code>movl -12(%rdx, %rcx, 4), %eax</code>
$\&E[i] - E$	$long$	i	<code>movq %rcx, %rax</code>

值得注意的是，计算同一数据结构中的两个指针之差，结果的数据类型为long。

3.6.2数组

3.6.2.1嵌套的数组

对于如下声明的数组 `T D[R][C]`，编译器会在内存中按照**“行优先”**为其分配空间，把它看成R个含有C个T型数据的数组，它的数组元素 $D[i][j]$ 的内存地址为 $\&D[i][j] = x_D + L * (i * C + j)$ （其中L是数据类型T以字节为单位的大小）。访问时，编译器会以**数组起始位置为基地址**，**偏移量为索引**，**计算期望元素的偏移量**。例如，对于一个二维数组 `int A[5][3]`，当我们访问 `A[i][j]` 时，假设 x_A 、 i 、 j 分别存放在寄存器%rdi、%rsi、%rdx中，汇编代码如下：

```
leaq    (%rsi,%rsi,2),%rax    # compute 3i
leaq    (%rdi,%rax,4),%rax    # compute xA+12i
movl    (%rax,%rdx,4),%rax    # read from M[xA+12i+4j]
```

3.6.2.2定长数组和变长数组

C语言编译器能够**优化**定长多维数组上的操作代码。历史上C语言只支持大小**在编译时就能确定的多维数组**，程序员在需要变长数组时必须得使用`malloc`或`calloc`这样的函数为这些数组分配存储空间，而且不得不显示地编码，用行优先索引将多维数组映射到一维数组。ISO C99中引入了一种功能，允许数组的维度是表达式，在数组被**分配**的时候才计算出来。注意，动态的版本中必须用**乘法指令**进行**跨维度**时下标的伸缩，而不能采用一系列的移位和加法。

如果允许使用优化，一般情况下，GCC可能会将所有的数组引用都转换成**指针间接引用**，并且其常常可以利用访问模式的**规律性**来优化索引的计算，识别出程序访问多维数组元素的**步长**。这样，就能避免一些移位时的乘法，不论是生成基于指针的代码还是基于数组的代码，这样的优化能够显著提高程序的性能。

一个示例如下：

```
int fix_prod_ele(fix_matrix A, fix_matrix B, long i, long k)
{
    long j;
    int result = 0;
    for (j = 0; j < N; j++)
        result += A[i][j] * B[j][k];
    return result;
}
```

编译器优化后的C代码如下：

```
int fix_prod_ele(fix_matrix A, fix_matrix B, long i, long k)
{
    int *Aptr = &A[i][0];
    int *Bptr = &B[0][k];
    int *Bend = &B[N][k];
    int result = 0;
    do                                     //No Need for initial test
    {
        result += *Aptr * *Bptr;
        Aptr++;                           //Move Aptr to next column
        Bptr += N;                         //Move Bptr to next row
    } while (Bptr != Bend);               //Test for stopping point
    return result;
}
```

3.7 异质的数据结构

C语言提供了两种将不同类型的对象组合到一起创建数据类型的机制：**结构**，用关键字`struct`来声明，将多个对象集合到一个单位中；**联合**，用关键字`union`来声明，允许用几种不同的类型来引用一个对象。

3.7.1 结构

C语言`struct`声明创建的数据类型将可能不同类型的对象聚合到一个对象中，用名字来引用结构的各个组成部分。结构的所有组成部分都存放在内存中一段**连续**的区域内，而**结构的指针就是结构中第一个字节的地址**。编译器会维护关于每个结构类型的信息，**指示每个字段的字节偏移**。它以这些**偏移**作为内存引用指令中的**位移**，从而产生对结构元素的引用。

要产生一个指向结构内部对象的指针，我们只需要将**结构的地址加上该字段的偏移量**。例如，对于结构：

```
struct demo{
    int i;
    int j;
    int a[10];
}
```

我们只需要一行汇编代码 `leaq 8(%rdi,%rsi,4)` 就可以产生C语句 `&(demo->a[i])` 的效果，访问到结构成员 `a[i]`（假设 `struct demo` 类型的变量存放在寄存器 `%rdi` 中，数组下标 `i` 存放在寄存器 `%rsi` 中）。并且，需要了解的是，结构中各个字段的选取完全是在**编译时**处理的。机器代码**不包含**关于字段声明或字段名字的信息。

3.7.2联合

联合提供了一种规避C语言类型系统的方式，允许以**多种类型来引用同一个对象**，用关键字 `union` 来声明。一个联合中的所有字段引用的都是数据结构的**起始位置**。

一种应用情况是，我们事先知道对一个数据结构中的两个不同字段的使用是**互斥的**，那么将这两个字段声明为联合的一部分，而不是结构的一部分，会**减少分配空间的总量**。这样，**一个联合总的大小等于它最大字段的大小**。假设我们想实现一个二叉树的数据结构，每个叶字结点都有两个 `double` 类型的数据值，而每个内部节点都有指向两个孩子节点的指针，但是没有数据，声明如下：

```
struct node_t{
    nodetype_t type;           // type of node
    union{
        struct{
            struct node_t *left;
            struct node_t *right;
        } internal;
        double data[2];
    }
} info;
```

这个结构总共需要24个字节：`type`是4个字节，`info.internal.left`和`info.internal.right`各要8个字节，或是`info.data`需要16个字节，且在字段`type`和联合的元素之间需要4个字节的填充。

联合还可以用来访问**不同数据类型的位模式**。当用联合来将各种不同大小的数据类型结合到一起时，**字节顺序**问题就变得非常重要了（注意利用联合进行强制类型转换时，小端法机器和大端法机器上的字节顺序不同）。

3.7.3数据对齐

许多计算机系统对基本数据类型的**合法地址**做出了一些限制，要求某种类型对象的地址必须是某个值 K （通常是2、4或8）的倍数。这种**对齐限制**简化了形成处理器和内存之间接口的**硬件设计**。当然，无论内存是否对齐，x86-64硬件都能正确工作，但是，对齐数据可以**提高内存系统的性能**。对齐原则是**任何 K 字节的基本对象的地址必须是 K 的倍数**。编译器也可以通过指令 `.align 8` 来指明**全局数据**所需的对齐。

对于包含结构的代码，编译器可能需要在字段的分配中插入**间隙**，以保证每个结构元素都满足它的对齐要求；并且，结构本身对它的起始地址也有一些对齐要求，因此，结构的末尾可能需要一些填充。

3.8机器级程序中控制与数据的结合

3.8.1GDB调试器

GNU的调试器GDB提供了许多有用的特性，支持机器级程序的运行时评估和分析，可以观察正在运行的程序，同时又对程序的执行有相当的控制。

3.8.2 内存越界引用和缓冲区溢出

我们知道，C语言中对于数组引用不进行**边界检查**，并且**局部变量和状态信息**（例如保存的寄存器值和返回地址）都存放在栈中。这两种情况结合到一起就能导致严重的程序错误，**对越界的数组元素的写操作会破坏存储在栈中的状态信息**。

一种特别常见的状态破坏称为**缓冲区溢出**。例如，在栈中分配某个字符数组来保存一个字符串，但是字符串的长度超出了为数组分配的空间。如下所示就说明了这个问题：

```
void echo()
{
    char buf[8];
    gets(buf);
    puts(buf);
    return 0;
}
```

其中，`gets()` 和 `puts()` 是库函数，`gets()` 没有办法确定它是否为保存整个字符串分配了足够的空间。很多常用的库函数，都有这样一个属性——不需要告诉它们缓冲区的大小，就产生一个字节序列[97]。这样就会导致**缓冲区溢出漏洞**。上面C代码的GCC编译产生的机器代码如下：

```
echo:
    subq    $24, %rsp
    movq    %rsp, %rdi
    call    gets
    movq    %rsp, %rdi
    call    puts
    addq    $24, %rsp
    ret
```

可以看到，程序首先把栈指针减去了24，在栈上分配了24字节，字符数组buf就位于栈顶。并且，`%rsp`被复制到`%rdi`作为调用`gets`和`puts`的参数。但是，由于分配给buf数组的空间后紧跟着的就是（`echo`函数的）返回地址，如果用户输入的字符串足够长（超过24个字符），那么，**返回指针**的值以及更多可能的保存状态就会被破坏。如果存储的返回地址的值被破坏了，`ret`指令就会导致程序跳转到一个完全意想不到的位置。

缓冲区溢出的一个更加致命的使用就是**让程序执行它本来不愿意执行的函数**，这是一种最常见的通过计算机网络攻击系统安全的方法。通常，输入给程序一个字符串，这个字符串包含一些**可执行代码的字节编码**，称为**攻击代码**。另外，还有一些字节会用一个指向攻击代码的指针**覆盖返回地址**。那么，执行`ret`指令的效果就是**跳转到攻击代码**。

在一种攻击形式中，攻击代码会使用系统调用启动一个**shell**程序，给攻击者提供一组**操作系统函数**。在另一种攻击形式中，攻击代码会执行一些**未授权的任务**，修复对栈的破坏，然后第二次执行`ret`指令，**（表面上）正常返回到调用者**。

3.8.3 对抗缓冲区溢出攻击

现代编译器和操作系统实现了很多机制以避免遭受这样的攻击，限制入侵者通过缓冲区溢出攻击获得系统控制权。

3.8.3.1 栈随机化

攻击者为了产生指向攻击代码的指针，需要知道字符串放置的栈地址。在过去，栈的位置相当固定。因此，如果攻击者可以确定一个常见的Web服务器所使用的栈空间，就可以设计一个在许多机器上都能实施的攻击。

栈随机化的思想使得栈的位置在程序每次运行时都有变化。它的实现方式是：程序开始时，在栈上分配一段0~n字节之间的随机大小的空间，例如，使用分配函数`alloca`在栈上分配指定字节数量的空间。程序并不使用这段空间，但是它会导致程序每次执行时后续的**栈位置发生变化**。分配的范围n必须**足够大**才能获得足够多的栈地址变化，又必须**足够小**不至于浪费程序太多的空间。在Linux系统中，栈随机化已经成为了一类更大的技术**地址空间布局随机化**中的一种。

但是，攻击者依然有办法利用**蛮力克服随机化**，他可以反复地用不同的地址进行攻击。一种常见的技巧就是在实际的攻击代码前插入很长的一段`nop`（读作“no op”，意即“no operation”）。执行`nop`指令除了对程序计数器加1，没有任何效果，但只要攻击者猜测的返回地址在这段序列中，程序就会“滑过”这段序列，到达攻击代码，因此这个序列的常用术语叫做“**空操作雪橇**”。对于32位操作系统，这样枚举尚且可以接受；但是对于64位操作系统，这就非常困难了。因此，**栈随机化**可以大大降低病毒或是蠕虫攻击一个系统的难度，但是也不能提供完全的安全保障。

3.8.3.2 栈破坏检测

计算机的第二道防线是能够检测到**何时栈已经被破坏**，这种破坏通常发生在当**超越局部缓冲区边界**时。最近的GCC版本在产生的代码中加入了一种**栈保护者**的机制。其思想是在局部缓冲区与栈状态之间存储一个特殊的**金丝雀值**。这个**金丝雀值**（canary），也被称为**哨兵值**，是在程序每次运行时随机产生的，因此，攻击者没有简单的办法知道它是什么。在**恢复寄存器状态**以及**从函数返回**之前，程序检查这个**金丝雀值**是否被该函数的某个操作或者该函数调用的某个函数的某个操作改变了。如果是的，那么程序**异常终止**。

下面是一段插入了**溢出检测**的汇编代码的例子：

```
echo:
    subq    $24, %rsp
    movq    %fs:40, %rax
    movq    %rax, 8(%rsp)
    call    gets
    movq    8(%rsp), %rax
    xorq    %fs:40, %rax
    je      .L9
    call    __stack_chk_fail
.L9:
    addq    $24, %rsp
    ret
```

可以看到，函数先通过指令参数`%fs:40`从内存中读出一个值，再把它存放在栈中相对于`%rsp`偏移量为8的地方。`%fs:40`指明**金丝雀值**是用**段寻址**从内存中读入的。将**金丝雀值**放在一个特殊的段中，标志为**只读**，这样攻击者就不能覆盖存储的**金丝雀值**。在函数返回之前，程序通过`xorq`指令来将**当前存储在栈对应位置上的值与金丝雀值**进行比较，如果两个数不同，代码就会调用一个错误处理的例程。

栈保护很好地防止了缓冲区溢出攻击破坏存储在程序栈上的状态，且只会带来很小的性能损失，特别是因为GCC只有在函数中有**局部char类型缓冲区**的时候才会插入这样的代码。

3.8.3.3 限制可执行代码区域

这道防线可以消除攻击者向系统中插入可执行代码的能力。一种方法是限制哪些内存区域能够存放可执行代码。在典型的程序中，只有保存编译器产生的代码的那部分内存才是需要可执行的，其他部分可以被限制为只允许读和写。

许多系统允许控制三种访问形式：**读**（从内存读数据）、**写**（存储数据到内存）和**执行**（将内存中的内容看作机器级代码）。

3.8.4 支持变长栈帧

在此前讨论的情况中，编译器都能够预先确定需要为栈帧分配多少空间。但是有些函数，需要的局部存储是**变长的**。例如，当函数调用标准库函数`alloca()`（用于在栈上分配任意字节数量的存储），或是代码声明一个**局部变长数组**时。

为了管理**变长栈帧**，x86-64代码使用寄存器`%rbp`作为**帧指针**（frame pointer，或base pointer）。当使用帧指针时，代码必须先把`%rbp`之前的值保存到栈中，因为它是一个**被调用者保存寄存器**。然后在函数的整个执行过程中，都使得`%rbp`指向那个时刻栈的位置。对于固定长度的局部变量，都用**相对于`%rbp`的偏移量**来引用它们。在**变长栈帧**中，各个局部变量的**对齐**操作十分关键。

在较早版本的x86代码中，每个函数调用都使用了帧指针，而现在，只在**栈帧长可变**的情况下才使用。

3.9 浮点体系结构

处理器的**浮点体系结构**包含存储、访问、操作、传递浮点数值等多个方面。2013年Core i7 Haswell处理器中引入了AVX2浮点体系结构，其允许数据存储在16个YMM寄存器中，它们的名字为`%ymm0 ~ %ymm15`。每个YMM寄存器都是256位（32字节）。当对标量数据操作时，这些寄存器只保存浮点数，且只使用**低32位**（对于`float`）或**低64位**（对于`double`）。汇编代码使用寄存器的SSE XMM寄存器名字`%xmm0 ~ %xmm15`来引用它们，每个XMM寄存器都对应YMM寄存器的低128位（16字节）。其中，`%xmm0`是用于**储存返回值的寄存器**，`%xmm1 ~ %xmm7`用于保存参数，`%xmm8 ~ %xmm15`为**调用者保存寄存器**。

3.9.1 浮点传送和转换操作

在内存与寄存器间进行**浮点传送**的部分指令如下（表中的 X 表示XMM寄存器）：

指令	源	目的	描述
<code>vmovss</code>	M_{32}/X	X/M_{32}	传送单精度浮点数
<code>vmovsd</code>	M_{64}/X	X/M_{64}	传送双精度浮点数
<code>vmovaps</code>	X	X	传送对齐的封装好的单精度数
<code>vmovapd</code>	X	X	传送对其的封装好的双精度数

AVX引用内存的指令为**标量指令**，意味着它们只对**单个**而不是一组封装好的数据值进行操作。无论数据对齐与否，这些指令都能正确执行，不过代码优化规则建议**32位内存数据满足4字节对齐，64位内存数据满足8字节对齐**。

标量指令的操作对象为一个数，而向量指令的操作对象为向量，即有序排列的一组数。

在浮点数与整数间进行转换的指令如下：

指令	源	目的	描述
vcvttss2si	X/M_{32}	R_{32}	用截断的方法把单精度数转换成整数
vcvttss2si	X/M_{64}	R_{32}	用截断的方法把双精度数转换成整数
vcvttss2siq	X/M_{32}	R_{64}	用截断的方法把单精度数转换成四字整数
vcvttss2siq	X/M_{64}	R_{64}	用截断的方法把双精度数转换成四字整数

指令	源1	源2	目的	描述
vcvtsi2ss	M_{32}/R_{32}	X	X	把整数转换成单精度数
vcvtsi2sd	M_{32}/R_{32}	X	X	把整数转换成双精度数
vcvtsi2ssq	M_{64}/R_{64}	X	X	把四字整数转换成单精度数
vcvtsi2sdq	M_{64}/R_{64}	X	X	把四字整数转换成双精度数

表一中的指令是把一个从XMM寄存器或内存中读出的浮点值进行转换，并将结果写入一个**通用寄存器**。在把浮点数转换成整数的时候，指令会执行**截断**，将值向0进行舍入。表二中的指令则是把整数转换成浮点数，其使用的是三操作数模式。其中第一个操作数读自于**内存或一个通用目的寄存器**。第二个操作数的值只会影响结果的**高位字节**，因此通常情况下可以忽略。而目标必须是**XMM寄存器**。在最常见的使用场景中，第二个源和目的的操作数都是一样的。例如，指令 `vcvtsi2sdq %rax,%xmm1,%xmm1` 从寄存器%rax中读出一个长整数，把它转换成数据类型`double`，并把结果存放至XMM寄存器%xmm1的低字节中。

在两种不同的浮点格式之间的转换对应的汇编代码如下：

```
# Conversion from single to double precision
vunpcklps    %xmm0,%xmm0,%xmm0    # Replicate first vector element
vcvtps2pd    %xmm0,%xmm0          # Convert two vector elements to double
```

`vunpcklps`指令通常用来**交叉放置**来自两个XMM寄存器的值，并把它们存储到第三个寄存器中。在上面的代码中，如果初始%xmm0寄存器的内容为字 $[s_3, s_2, s_1, s_0]$ ，则该指令会将寄存器的值更新为 $[s_1, s_1, s_0, s_0]$ 。然后，`vcvtps2pd`指令把源XMM寄存器中的两个低位单精度值拓展成目的XMM寄存器中的两个双精度值。因此，寄存器的值会被更新为 $[d_{x_0}, d_{x_0}]$ ，其中 d_{x_0} 是将 x 转换成双精度后的结果。

实际上，我们不太清楚为什么GCC会产生这样的代码，没有必要在XMM寄存器中将结果复制一遍。其实，可以采用单条指令 `vcvtsi2sdq %xmm0,%xmm0,%xmm0` 将保存在%xmm0低位4字节的单精度值转换成一个双精度值，并将结果存储在寄存器%xmm0的低8字节。

将双精度转换为单精度的汇编代码如下：

```
# Conversion from double to single precision
vmovddup     %xmm0,%xmm0          # Replicate first vector element
vcvtupd2psx  %xmm0,%xmm0          # Convert two vector elements to single
```

假设初始寄存器%xmm0中保存着两个双精度值 $[x_1, x_0]$ 。然后`vmovddup`指令把它设置为 $[x_0, x_0]$ 。`vcvtupd2psx`指令把这两个值转换成单精度，再存放到该寄存器的低位一半中，并将高位一半设置为0，得到结果 $[0.0, 0.0, x_0, x_0]$ 。

同样，我们也可以使用单条指令 `vcvtsd2ss %xmm0,%xmm0,%xmm0` 来完成`double`到`float`的转换，编译器使用上面的方式并没有明显直接的意义。

一个在不同浮点数格式间进行转换的C代码示例如下：

```
double fcvt(int i, float *fp, double *dp, long *lp)
{
    float f = *fp; double d = *dp; long l = *lp;
    *lp = (long) d;
    *fp = (float) i;
    *dp = (double) l;
    return (double) f;
}
```

GCC对应产生的汇编代码如下：

```
_Z4fcvtiPfPdPl:
.LFB0:
    movss    (%rdx), %xmm0           # Get f = *fp
    movq     (%r9), %rax             # Get l = *lp
    vcvtsd2siq (%r8), %r10           # Get d = *dp and convert to long
    movq     %r10, (%r9)             # Store d at lp
    pxor     %xmm1, %xmm1           # Reset %xmm1
    vcvtsi2ss %ecx, %xmm1, %xmm1     # Convert i to float
    movss    %xmm1, (%rdx)           # Store i at fp
    pxor     %xmm1, %xmm1           # Reset %xmm1
    vcvtsi2sdq %rax, %xmm1, %xmm1     # Convert l to double
    movsd    %xmm1, (%r8)           # Store l at dp
    # The following two instructions convert f to double
    vunpcklps %xmm0, %xmm0, %xmm0
    vcvtps2pd %xmm0, %xmm0
    ret
```

3.9.2过程中的浮点代码

对于含有浮点的代码，有如下几条规则：

- XMM寄存器`%xmm0 ~ %xmm7`最多可以传递**8个**浮点参数。按照参数列出的顺序使用这些寄存器。同时可以通过**栈**传递额外的浮点参数。
- 函数通过寄存器`%xmm0`来返回浮点值。
- 所有的XMM寄存器都是**调用者保存**的。被调用者**可以不用保存就覆盖这些寄存器中任意一个**。

当函数包含指针、整数和浮点数混合的参数时，指针和整数通过**通用寄存器**传递，而浮点值通过**XMM寄存器**传递。也就是说，**参数到寄存器的映射取决于它们的类型和排列的顺序**。

3.9.3浮点的运算与比较操作

3.9.3.1浮点的运算操作

下表为一组执行算术运算或是位级运算的标量AVX2浮点指令：

单精度	双精度	效果	描述
vaddss	vaddsd	$D \leftarrow S_2 + S_1$	浮点数加
vsubss	vsubsd	$D \leftarrow S_2 - S_1$	浮点数减
vmulss	vmulsd	$D \leftarrow S_2 \times S_1$	浮点数乘
vdivss	vdivsd	$D \leftarrow S_2 \div S_1$	浮点数除
vmaxss	vmaxsd	$D \leftarrow \max(S_2, S_1)$	浮点数取最大值
vminss	vminsd	$D \leftarrow \min(S_2, S_1)$	浮点数取最小值
sqrtps	sqrtsd	$D \leftarrow \sqrt{S_1}$	浮点数平方根
vxorps	xorpd	$D \leftarrow S_2 \wedge S_1$	位级异或
vandps	andpd	$D \leftarrow S_2 \& S_1$	位级与

其中每条指令都有一个 (S_1) 或两个 (S_1 和 S_2) 源操作数和一个目的操作数 D 。第一个操作数 S_1 可以是一个**XMM寄存器**或一个**内存位置**。第二个操作数和目的操作数必须是**XMM寄存器**。此外，注意到在**XMM寄存器**上执行的位级操作类似于它们在**通用寄存器**上的操作，这些操作对两个源寄存器的所有位都实施指定的位级操作。需要注意的是，当整数和一个浮点数进行运算时，编译器会默认将整数类型转换为浮点数。

3.9.3.2浮点的比较操作

AVX2提供了两条用于**比较**浮点数值指令：

指令	基于	描述
<i>vucomiss</i> S_1, S_2	$S_2 - S_1$	比较单精度数值
<i>vucomisd</i> S_1, S_2	$S_2 - S_1$	比较双精度数值

这些指令在操作数顺序、条件码的设置上与CMP指令类似。并且，**参数 S_2 必须在XMM寄存器中，而参数 S_1 可以在XMM寄存器中，也可以在内存中。**

浮点比较指令会设置三个**条件码**：**零标识位ZF**、**进位标志位CF**和**奇偶标志位PF**。对于奇偶标志位，当最近的一次算术或逻辑运算产生的值的最低位字节是**偶校验**的（即这个字节中有**偶数个1**），那么就会设置这个标志位。特别地，**对于浮点比较，当两个操作数中任何一个是非NaN时，就会设置该位**。此时，我们称这种情况是**无序的**，否则为**有序的**。如果**无序**，就会认为比较失败了，这个标志位就被用来发现这样的条件（并通常通过条件跳转指令*jp*进行跳转）。在有序的情况下，如果 $S_2 < S_1$ ，进位标志位CF会被设置为1；如果 $S_2 = S_1$ ，零标志位ZF会被设置为1；如果 $S_2 > S_1$ ，则两个标志位CF和ZF都会被设置为0。无序情况下三个标志位都会被设置为1。

3.9.4定义和使用浮点数常数

和整数运算操作不同，AVX浮点操作**不能以立即数值**作为操作数。相反，编译器必须为所有的常量值**分配和初始化存储空间**，然后代码再把这些值从内存读入。一个使用浮点数常数的例子如下：

```
double cel2fahr(double temp){
    return 1.8 * temp + 32.0;
}
```

GCC产生的汇编代码如下：

```
_Z8cel2fahrd:
.LFB0:
    vmulsd    .LC0(%rip), %xmm0
    vaddsd    .LC1(%rip), %xmm0
    ret
.LC0:
    .long     3435973837
    .long     1073532108
.LC1:
    .long     0
    .long     1077936128
```

可以看到，函数分别从标号为.LC0的内存位置读出值1.8，从标号为.LC1的位置读入值32.0。这些标号对应的值都是通过**一对.long声明和十进制表示的值指定的**。如果机器采用的是**小端法**字节顺序，那么第一个值给出的就是低位4字节，第二个值给出的就是高位4字节。