3.指令的机器级表示

3.2 程序编码

3.2.1 机器级代码

程序计数器（PC，通常存放在%rip中）给出要执行的下一条指令在内存中的地址。
整数寄存器文件包含16个命名的位置，分别存储64位的值。
条件寄存器保存着最近执行的算术或逻辑指令的状态信息。

3.3 数据格式

C声明	Intel数据类型	汇编代码后缀	大小(字节)
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

图 3-1　C语言数据类型在 x86-64 中的大小。在 64 位机器中，指针长 8 字节

3.4 访问信息

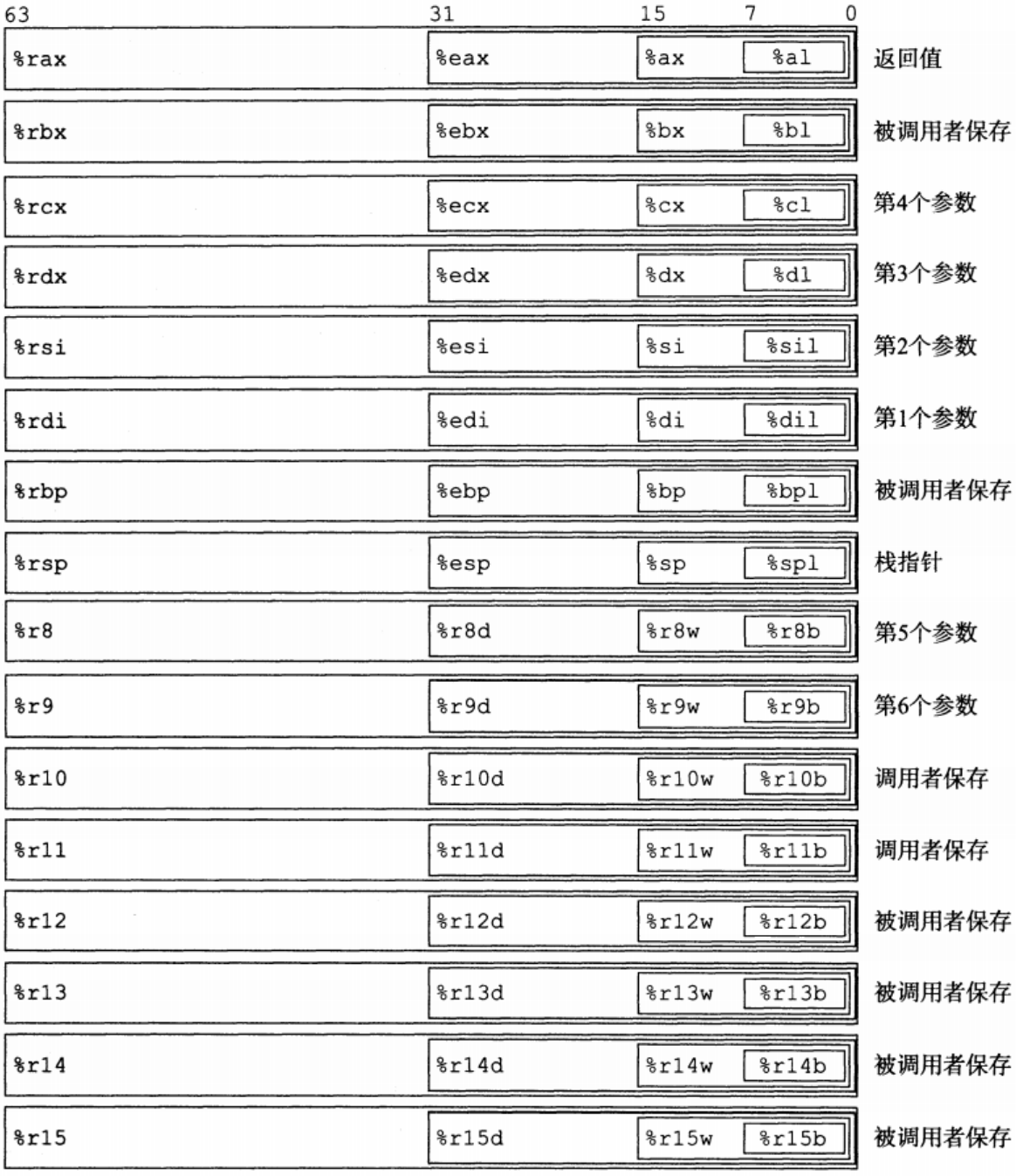


图 3-2　整数寄存器。所有 16 个寄存器的低位部分都可以作为字节、字(16 位)、双字(32 位)和四字(64 位)数字来访问

(传递数据时，生成1字节和两字节数字的指令会保持剩下的字节不变；生成4字节数字的指令会把高4位个字节置成零)

3.4.1 操作数指示符

操作数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。
有三种方式：立即数寻址、寄存器寻址、内存寻址（间接寻址和绝对寻址）

类型	格式	操作数值	名称
立即数	$\$Imm$	Imm	立即数寻址
寄存器	r_a	$R[r_a]$	寄存器寻址
存储器	Imm	$M[Imm]$	绝对寻址
存储器	(r_a)	$M[R[r_a]]$	间接寻址
存储器	$Imm(r_b)$	$M[Imm+R[r_b]]$	（基址 + 偏移量）寻址
存储器	(r_b, r_i)	$M[R[r_b]+R[r_i]]$	变址寻址
存储器	$Imm(r_b, r_i)$	$M[Imm+R[r_b]+R[r_i]]$	变址寻址
存储器	$(,r_i, s)$	$M[R[r_i] \cdot s]$	比例变址寻址
存储器	$Imm(,r_i,s)$	$M[Imm+R[r_i] \cdot s]$	比例变址寻址
存储器	(r_b, r_i, s)	$M[R[r_b]+R[r_i] \cdot s]$	比例变址寻址
存储器	$Imm(r_b, r_i, s)$	$M[Imm+R[r_b]+R[r_i] \cdot s]$	比例变址寻址

图 3-3　操作数格式。操作数可以表示立即数(常数)值、寄存器值或是来自内存的值。比例因子 s 必须是 1、2、4 或者 8

3.4.2 数据传输指令

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

图 3-4　简单的数据传送指令

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

图 3-5 零扩展数据传送指令。这些指令以寄存器或内存地址作为源，以寄存器作为目的

指令	效果	描述
MOVS S，R	R←符号扩展(S)	传送符号扩展的字节
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

图 3-6 符号扩展数据传送指令。MOVS 指令以寄存器或内存地址作为源，以寄存器作为目的。cltq 指令只作用于寄存器%eax 和%rax

其中，传送l大小的数据会自动将高位置为0，所以不需要movzlq这条指令。
传送指令的源操作数和目的操作数必须大小匹配。

3.4.4 压入和弹出栈数据

注意，是从高地址向低地址压栈！栈底在高地址，栈顶在低地址，且地址从右往左增大。在将字符压栈的时候不要写反了！同时注意字符串是大端法噢
栈顶元素保存在%rsp中。

指令	效果	描述
pushq S	R[%rsp]←R[%rsp]－8； M[R[%rsp]]←S	将四字压入栈
popq D	D←M[R[%rsp]]； R[%rsp]←R[%rsp]＋8	将四字弹出栈

图 3-8 入栈和出栈指令

3.5 算数和逻辑操作

注意，都是第二个操作数为目的操作数，同时运算时是第二个操作数在前（作为被减数、被除数）。

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

图 3-10 整数算术操作。加载有效地址(leaq)指令通常用来执行简单的算术操作。其余的指令是更加标准的一元或二元操作。我们用$>>_A$和$>>_L$来分别表示算术右移和逻辑右移。注意，这里的操作顺序与 ATT 格式的汇编代码中的相反

3.5.1 加载有效地址lea (load effective address)

目的操作数必须是一个寄存器

```
leaq    (%rdi,%rdi,4), %rax
leaq    (%rax,%rsi,2), %rax
leaq    (%rax,%rdx,8), %rax
```

这里的括号并没有从内存中取数据的作用!

3.5.3 移位操作

移位量可以是一个立即数，或者存放在单字节寄存器%cl中（只允许使用存在这个寄存器中）

3.5.5 特殊的算术操作

指令		效果	描述
imulq	S	R[%rdx]:R[%rax]←S×R[%rax]	有符号全乘法
mulq	S	R[%rdx]:R[%rax]←S×R[%rax]	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

看似是单操作数，实际上另一个操作数存放在%rax中，且%rax作为被除数/乘数。

乘法的结果，高32位存放在%rdx中，低32位存放在%rax中。
除法的结果，余数存放在%rdx中，结果存放在%rax中。

3.6 控制

3.6.1 条件码

CF:进位标志。最近的操作使最高位产生了进位。可以用来检查无符号操作的溢出。
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-正溢出或者负溢出。
leaq不会改变任何条件码，因为他仅仅是用来进行地址运算的。
其他指令，例如XOR会将进位标志和溢出标志设置成0；移位操作会将进位标志设置为最后一个被移出的位，而溢出标志设置为0。
有两类指令只设置条件码而不改变任何其他寄存器。

指令		基于	描述
CMP	S_1，S_2	S_2-S_1	比较
cmpb			比较字节
cmpw			比较字
cmpl			比较双字
cmpq			比较四字
TEST	S_1，S_2	$S_1 \& S_2$	测试
testb			测试字节
testw			测试字
testl			测试双字
testq			测试四字

图 3-13 比较和测试指令。这些指令不修改任何寄存器的值，只设置条件码

testq %rax，%rax **是用来测试**%rax**的符号。**

3.6.2 访问条件码

通常有三种方式:
1）可以根据条件码的某种组合，将一个字节设置为0或者1；
2）可以条件跳转到程序的某个其他部分；
3）可以有条件地传送数据

指令	同义名	效果	设置条件
sete D	setz	D ← ZF	相等/零
setne D	setnz	D ← ~ZF	不等/非零
sets D		D ← SF	负数
setns D		D ← ~SF	非负数
setg D	setnle	D ← ~(SF ^ OF) & ~ZF	大于（有符号>）
setge D	setnl	D ← ~(SF ^ OF)	大于等于（有符号>=）
setl D	setnge	D ← SF ^ OF	小于（有符号<）
setle D	setng	D ← (SF ^ OF) | ZF	小于等于（有符号<=）
seta D	setnbe	D ← ~CF & ~ZF	超过（无符号>）
setae D	setnb	D ← ~CF	超过或相等（无符号>=）
setb D	setnae	D ← CF	低于（无符号<）
setbe D	setna	D ← CF | ZF	低于或相等（无符号<=）

图 3-14 SET 指令。每条指令根据条件码的某种组合，将一个字节设置为 0 或者 1。有些指令有“同义名”，也就是同一条机器指令有别的名字

3.6.3 跳转指令

指令	同义名	跳转条件	描述
jmp *Label*		1	直接跳转
jmp **Operand*		1	间接跳转
je *Label*	jz	ZF	相等/零
jne *Label*	jnz	~ZF	不相等/非零
js *Label*		SF	负数
jns *Label*		~SF	非负数
jg *Label*	jnle	~(SF ^ OF) & ~ZF	大于（有符号>）
jge *Label*	jnl	~(SF ^ OF)	大于或等于（有符号>=）
jl *Label*	jnge	SF ^ OF	小于（有符号<）
jle *Label*	jng	(SF ^ OF) | ZF	小于或等于（有符号<=）
ja *Label*	jnbe	~CF & ~ZF	超过（无符号>）
jae *Label*	jnb	~CF	超过或相等（无符号>=）
jb *Label*	jnae	CF	低于（无符号<）
jbe *Label*	jna	CF | ZF	低于或相等（无符号<=）

图 3-15 jump 指令。当跳转条件满足时，这些指令会跳转到一条带标号的目的地。有些指令有“同义名”，也就是同一条机器指令的别名

3.6.4 跳转指令的编码

直接寻址和PC相对寻址。
直接寻址就是直接给绝对地址；
PC相对寻址是用目标指令的地址与紧跟在跳转指令之后的地址之差作为编码。（记得负数的表示）

3.6.5 用条件控制来实现条件分支

if-else结构

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
```

汇编语言会使用如下的形式：

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

注意第一个判断是！t，对立条件成立则跳转的else语句部分。

3.6.6 用条件传送来实现条件分支

先将分支成功和失败的结果都算出来，再根据分支结果选择使用哪一个。但如果两个表达式中任意一个可能产生错误条件或者副作用，就会导致非法行为（例如产生空指针）。
一般来说分支预测错误的开销会超过更复杂的计算，GCC还是会使用条件转移控制。总的来说，条件数据传送提供了一种用条件控制转移来实现条件操作的替代策略。它们只能适用于非常受限的情况。

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

指令		同义名	传送条件	描述
cmove	S, R	cmovz	ZF	相等/零
cmovne	S, R	cmovnz	~ZF	不相等/非零
cmovs	S, R		SF	负数
cmovns	S, R		~SF	非负数
cmovg	S, R	cmovnle	~(SF ^ OF) & ~ZF	大于（有符号>）
cmovge	S, R	cmovnl	~(SF ^ OF)	大于或等于（有符号>=）
cmovl	S, R	cmovnge	SF ^ OF	小于（有符号<）
cmovle	S, R	cmovng	(SF ^ OF) | ZF	小于或等于（有符号<=）
cmova	S, R	cmovnbe	~CF & ~ZF	超过（无符号>）
cmovae	S, R	cmovnb	~CF	超过或相等（无符号>=）
cmovb	S, R	cmovnae	CF	低于（无符号<）
cmovbe	S, R	cmovna	CF | ZF	低于或相等（无符号<=）

图 3-18　条件传送指令。当传送条件满足时，指令把源值 S 复制到目的 R。有些指令是“同义名”，即同一条机器指令的不同名字

3.6.7 循环

循环里面的判断是正向判断噢，和if语句不同。

1.do-while循环

do-while语句的通用形式如下：

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

汇编语言下：

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

2.while循环

while语句的通用形式如下：

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

因为在执行循环体之前要先进行一次判断，所以有两种翻译方式。

第一种：跳转到中间。即开头一个跳转指令，先跳转到判断代码部分。

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

第二种：guarded-do。在开头进行一次对立条件的判断，成立则跳转到while结尾，即不执行循环体。不成立后部分则和do-while循环一样。

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

相应地，还可以把它翻译成 goto 代码如下：

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

（当判断出第一次判断一定会成立时，可以不写第一个判断，直接写成while-do的形式。）

3.6.8 switch语句

跳转表、内部标号、对应的情况。

```
1   void switch_eg_impl(long x, long n,
2                       long *dest)
3   {
4       /* Table of code pointers */
5       static void *jt[7] = {
6           &&loc_A, &&loc_def, &&loc_B,
7           &&loc_C, &&loc_D, &&loc_def,
8           &&loc_D
9       };
10      unsigned long index = n - 100;
11      long val;
12
13      if (index > 6)
14          goto loc_def;
15      /* Multiway branch */
16      goto *jt[index];
17
18   loc_A:    /* Case 100 */
19      val = x * 13;
20      goto done;
21   loc_B:    /* Case 102 */
22      x = x + 10;
23      /* Fall through */
24   loc_C:    /* Case 103 */
25      val = x + 11;
26      goto done;
27   loc_D:    /* Cases 104, 106 */
28      val = x * x;
29      goto done;
30   loc_def:  /* Default case */
31      val = 0;
32   done:
33      *dest = val;
34  }
```

b）翻译到扩展的C语言

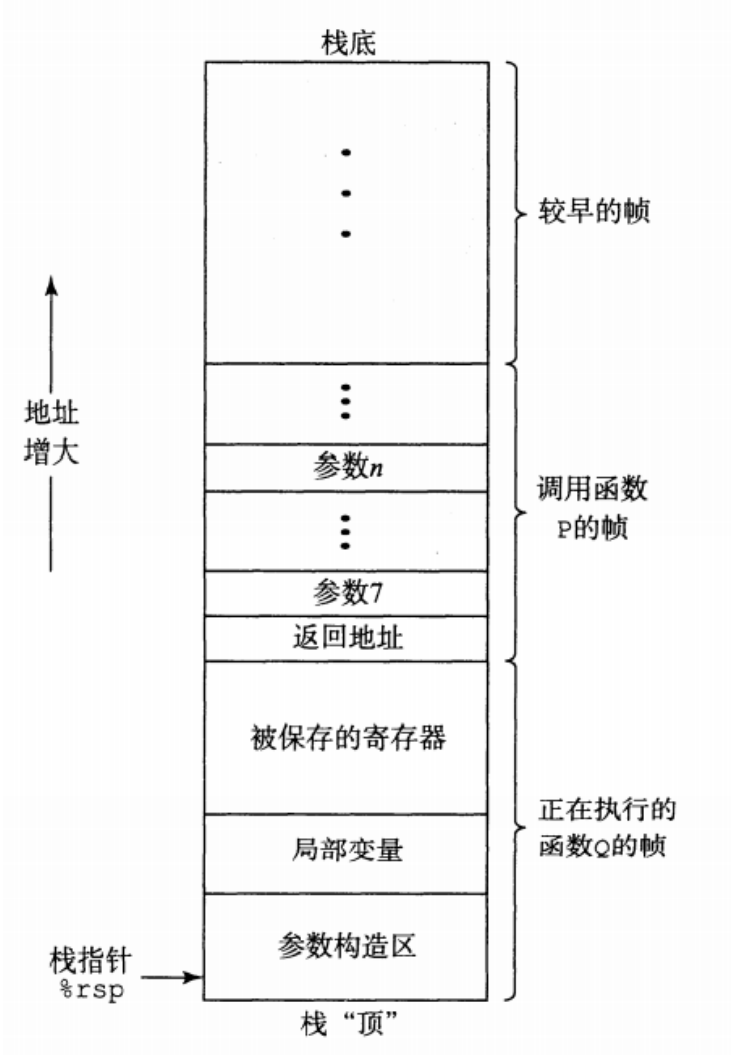
3.7 过程

假设过程P调用过程Q，Q执行后返回到P。这些动作包括下面一个或多个机制：
传递控制：程序计数器被设置为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后面的那条指令的地址。
传递数据：P必须能够向Q提供一个或多个参数，Q必须能够向P返回一个值。
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。

3.7.1 运行时栈

程序可以用栈来管理它的过程所需要的存储空间，栈和程序寄存器存放着传递控制和数据、分配内存所需要的信息。当前

执行的过程的帧总是在栈顶。大多数过程的栈帧都是定长的，在过程的开始就分配好了。许多函数甚至不需要栈帧。当所有局部变量都可以保存在寄存器中，而且该函数根本不会调用其他任何函数（又是称之为叶子过程）



3.7.2 转移控制

将控制从P转移到Q只需要简单地把程序计数器（PC）设置成Q的代码的起始位置。
call 指令会把地址A压入栈中，并将PC设置为Q的起始地址。压入的地址A被称为返回地址，是紧跟在call指令后面的那条指令的地址。ret指令会从栈中弹出地址A，并把PC设置为A。

3.7.3 数据传送

（这是调用函数P存的，要对齐8位）
大部分过程间的数据传送是通过寄存器实现的。但是通过寄存器最多传递六个整型（整数和指针）（六个寄存器位%rdi,%rsi,%rdx,%rcx,%r8,%r9）
如果参数大于6个，就要通过栈来传递。通过栈传递参数时，所有的数据大小都向8的倍数对齐。

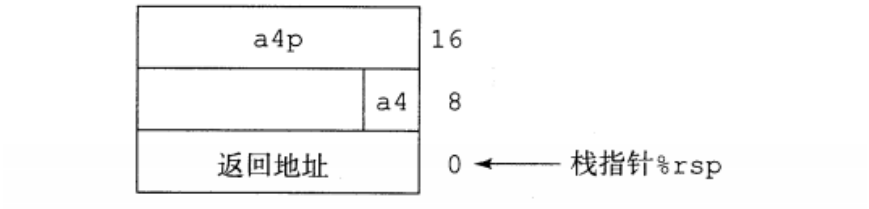
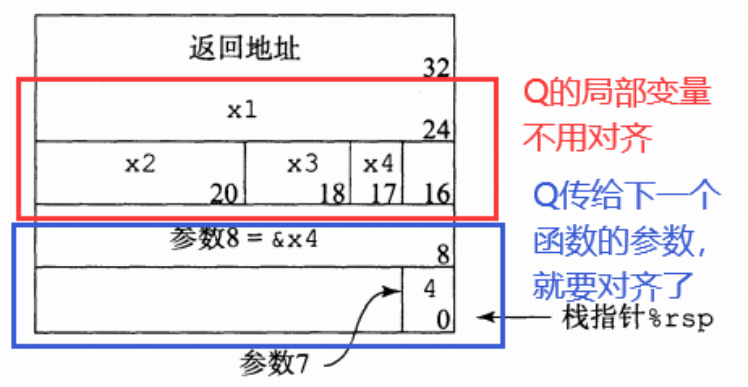


图 3-30　函数 proc 的栈帧结构。参数 a4 和 a4p 通过栈传递

(其中a4为字符类型)

3.7.4 栈上的局部存储

（这个是被调用函数Q的，不用对齐）
Q有些时候需要将局部变量存放在内存中：
①寄存器不足够存放所有的本地数据；
②对一个局部变量使用地址运算符“&”，必须先将局部变量存到内存中，才可以为它产生一个地址。
③某些局部变量是数组或者结构，因此必须通过数组或结构引用被访问到。



3.7.5 寄存器中的局部存储空间

%rbx，%rbi，%r12-%r15是被调用者保护寄存器，当过程Q要使用这些寄存器时，必须先将P存在这些寄存器中的值压栈。

3.7.6 递归调用

关键是代码之间的转换，一定要掌握好

3.8 数组分配和访问

3.8.1 基本原则

对于数据类型 T 和整型常数 N，声明如下：

T `A[`N`];`

3.8.2 指针运算

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

3.8.2 嵌套的数组

数组元素在内存中按照“行优先”的顺序排列。

行	元素	地址
A[0]	A[0][0]	x_A
	A[0][1]	x_A+4
	A[0][2]	x_A+8
A[1]	A[1][0]	x_A+12
	A[1][1]	x_A+16
	A[1][2]	x_A+20
A[2]	A[2][0]	x_A+24
	A[2][1]	x_A+28
	A[2][2]	x_A+32
A[3]	A[3][0]	x_A+36
	A[3][1]	x_A+40
	A[3][2]	x_A+44
A[4]	A[4][0]	x_A+48
	A[4][1]	x_A+52
	A[4][2]	x_A+56

图 3-36 按照行优先顺序存储的数组元素

计算某一数组元素的地址

T D[R][C];

它的数组元素 D[i][j]的内存地址为

$$\&D[i][j] = x_D + L(C \cdot i + j) \qquad (3.1)$$

3.8.5 变长数组

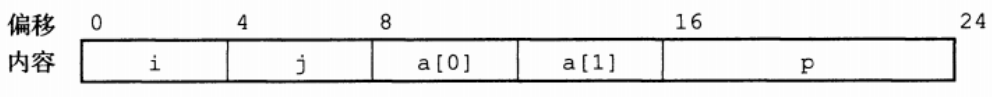
参数n必须在声明数组A[N][N]之前声明
就是用leaq计算i*C变成用imulq计算i*n。
允许优化的化可以先计算出这个值，就避免了乘法的使用。

3.9 异质的数据结构

3.9.1 结构（struct)

```
struct rec {
    int i;
    int j;
    int a[2];
    int *p;
};
```

这个结构包括 4 个字段：两个 4 字节 int、一个由两个类型为 int 的元素组成的数组和一个 8 字节整型指针，总共是 24 个字节：



这是数组元素本身对齐的情况，如果元素不对其（例如有了char），就要遵循对齐原则。

K	类型
1	char
2	short
4	int,float
8	long,double,char*

即右列中的元素的起始地址一定要是左列元素的整数倍。当末尾不为4的倍数时，也要补齐。

3.9.2 联合（union）

联合的大小等于其最大字段的大小。
联合因为存储位数相同，当读取数据时只是将二进制表示输出为相应要求的输出格式，所以可以做到将将double转换成位级相同的long类型。如果是直接强制类型转换的话只能做到截取前面的整数部分。

3.10 在机器级程序中将控制与数据结合起来

3.10.1 理解指针

3.10.3 内存越界引用和缓冲区溢出

（因为栈的栈顶是在低地址，所以会一直往高地址存数据，哪怕数据大小已经超过了分配给它的范围）从而就会导致内存越界，新存储的数据会覆盖掉先前存储在内存中的信息。攻击者就可以根据这点，将攻击代码的地址指针覆盖原函数的返回地址，从而导致在执行完被调用函数后，进入到攻击代码中。

3.10.4 对抗缓冲区溢出攻击

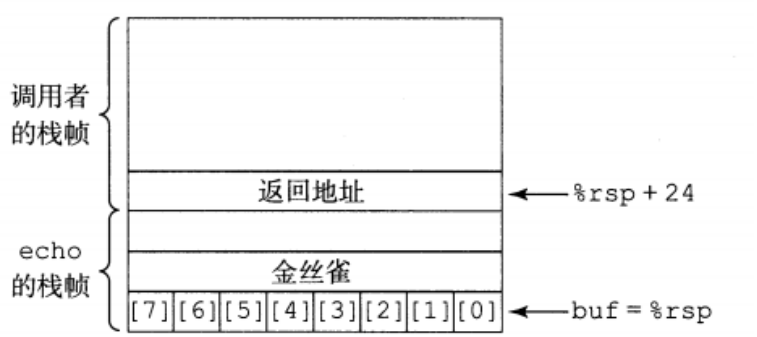
- 栈随机化
  攻击者实现攻击的两个任务：插入代码、插入指向该代码的指针。为了获取该指针，我们需要知道攻击的字符串的起始地址（栈地址）。大多数操作系统分配的栈空间的位置都是固定的，所以攻击者很容易确定这个地址。
  栈随机化：程序每次运行时分配的栈的位置都是变化的，执行相同代码可能有不同的栈地址。
  实现：使用alloca函数在程序执行之前，先在栈上分配0~n个空闲空间，程序不使用这段空间。n不能过大或过小，过大造成空间浪费，过小使得地址变化性小。
  攻击策略：获取地址变化范围，借助nop指令，暴力枚举。
  不断执行下面的程序，可以获取地址变化

  ```
  int main()
  {
      long local;
      printf(“%p\n”,&local);
      return 0;
  }
  ```

  假设在Linux-32执行以上10000次，地址变化范围为0xff7fc59c到0xffffd09c，大概范围为2^{23}。
  攻击者在自己的代码里面插入一定量的nop指令（只会使得PC不断自增），如果攻击者插入256个字节的nop，则只需要枚举2^{15}个起始地址就可以破解初始值$n = 2^{23}$的栈随机化。

- 栈破坏检查
  系统虽然无法限制越界访问，但是可以实现越界检查。
  栈破坏检查：在栈帧中的局部缓冲区和栈状态之间存放一个金丝雀值，当函数运行完毕准备返回时，程序检测这个

值是否发生变化，如果变化则说明越界写数据了，程序异常终止。



- 限制可执行代码区域
  限制内存中的一些区域能否存放可执行代码。
  典型的程序中，只有保存编译器产生的代码的那部分内存是可执行的。

3.10.5 **支持变长栈帧**

使用%rbp（base pointer）指向返回地址，以便于之后恢复。