**第二章**

**2.1整数表示**

**2.1.1十六进制表示法**

二进制与十六进制的转换、十进制与十六进制的转换

T2.2

n 十进制 十六进制

9 512 0x200

19 524288 0x80000

14 16384 0x4000

16 66536 0x10000

17 131072 0x20000

5 32 0x20

8 512 0x80

T2.4

0x5044

0x4FFc

0x507c

0xae

**2.1.2字数据大小**

x86-64下各数据类型的大小，注意long为8字节，指针（char\*）为8字节。

**2.1.3寻址和字节顺序**

大端法小端法，通常情况下考虑linux64系统，为小端法。

如0x01234567

在大端法下，按字节存储为01 23 45 67，在小端法下，为67 45 23 01

T2.5

A.21 87

B.21 43 87 65

C.21 43 65 87 65 43

**2.1.4表示字符串**

字符串12345的存储为31 32 33 34 35 00（数字x的ASCII码为0x3x，字符串结束符null为0x00），与大小端规则无关。

**2.1.5表示代码 2.1.6布尔代数简介**

略

**2.1.7c语言位级运算**

将数转为二进制进行位级运算，再转回原进制。

**2.1.8c语言逻辑运算**

略

T2.15

！（x^y）

**2.1.9c语言移位运算**

左移位只有一种，右边补0。

右移位有两种，逻辑右移在左端补0，算数右移在左端补最高有效位的值。

当移动位数k大于数据类型位数w，则实际位移量为k mod w。

如对int lval，有lva<<1== lva<<33。

**2.2整数表示**

一些数学术语的含义：如B为2进制，T为补码，U为无符号数。

B2为将w为2进制数转补码，为w位无符号下的加法。

**2.2.1整形数据类型**

在64位机上的数据范围：



若某数据类型的字节数为k，则补码范围是-2^(8k-1)~2^(8k-1)-1，无符号范围是0~2^8k-1。

若没有给出机器位数，则int最小要2字节，long最小要4字节。

**2.2.2无符号数编码**

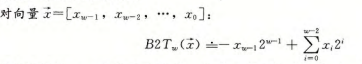
w位二进制数的编码方式：



对于w位能表示的无符号数，其最大值为[11...11]=2^w-1，其最小值为[00...00]=0。

**2.2.3补码编码**

与无符号编码相似，只不过最高位为负权：



对于w位补码，其最大值位[011...11]=2^(w-1)-1，其最小值为[100...00]=-2^(w-1)，而[111...11]=-1。



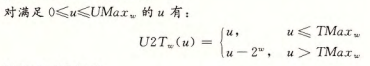
**2.2.4有符号数与无符号数的转换**

有符号数与无符号数的转换时，数值可能会改变，但是位模式不变。

T2U(x)=B2U(T2B(x))。

也可以使用公式：





即如果x在0~Tmax范围内，则有符号数与无符号数相等，否则无符号数比有符号数大2^w。

**2.2.5c语言中的有符号数和无符号数**

如果一个表达式的一个运算数是有符号的而另一个是无符号的，那么C语言会隐式地将有符号参数强制类型转换为无符号数，这一性质在标准算数中影响不大，但是在关系运算中值得注意。

T2.21

类型 求值

无 1

有 1

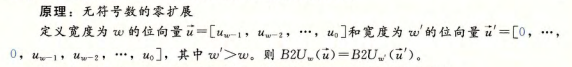
无 0

有 1

无 1

**2.2.6拓展一个数字的位表示**

在不同字长的整型间转换，而保持数值不变：无符号数——零拓展，有符号数——符号拓展。





T2.23

fun1 fun2

0x00000076 0x00000076

0x00000021 0x00000021

0x000000c9 0xffffffc9

0x00000087 0xffffff87

**2.2.7截断数字**

将w位的数截断为k位数字时，会丢弃其高w-k位，此操作可能会改变其数值（PS：拓展位数不会改变数值）。

截断无符号数时，x'=x mod 2^k，截断补码时x'=U2T(x mod 2^k)。

T2.24

无符号 补码

0 0

2 2

1 1

3 3

7 -1

**2.2.8关于有符号数与无符号数的建议**

略

**2.3整数计算**

**2.3.1无符号加法**

字长膨胀。

以参数x和y定义，表示将x+y截断w位的结果，即对2^w取模的结果。

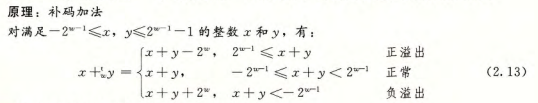
检测无符号溢出：s=x+y，当且仅当s<x时发生溢出。

无符号加法逆元：x=0时，加法逆元位0，x>0时，加法逆元为2^w-x。

**2.3.2补码加法**

由于补码加法与无符号加法有相同的位级表示，故将补码转化为无符号数相加再转化为补码。

也可以用公式（正溢出则减2^w，负溢出则加2^w）：



检测正溢出：当且仅当x>0且y>0但x+y<=0。

检测负溢出：当且仅当x<0且y<0但x+y>=0。

**2.3.3补码的非**

补码加法逆元：Tmin的逆元是它本身Tmin，其他数x的逆元是-x。

补码非的位级表示：对每一位求补再加1，即-x=~x+1。

T2.33

x x4位补码的加法逆元

十进制 十进制 十六进制

0 0 0

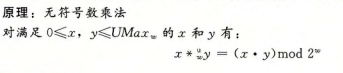
5 -5 B

-8 -8 8

-3 3 3

-1 1 1

**2.3.4无符号乘法**



**2.3.5补码乘法**

无符号乘法和补码乘法位级等价。



**2.3.6乘以常数**

再无符号和补码下，左移k位相当于乘以2^k。

如x\*14=x\*(2^3+2^2+2^1)=(x<<3)+(x<<2)+(x<<1)。

T2.40

x\*6=(x<<3)-(x<<1)

x\*31=(x<<5)-x

x\*(-6)=(x<<1)-(x<<3)

x\*55=(x<<6)-(x<<3)-x

**2.3.7除以2的幂**

无符号：右移k位相当于除以2^k再向下取整。

补码：算数右移k位相当于除以2^k再向下取整；而(x+(1<<k)-1)>>k产生数值x/(2^k)向上取整。

T2.44

32位下，Tmin=0x80000000=-2^31  
A. x=Tmin

B.真

C.x=2^30+1

D.真

E.x=Tmin

F.真

G.真

**2.4浮点数**

**2.4.1二进制小数**

与十进制小数类似，但是位权变为2的幂，小数点左边是2的正幂（0~m），小数点右边是2的负幂（-1~-n）。

需要注意，正如十进制无法精确表示1/3这样的数，很多十进制中能精确表示的数在二进制下也无法精确表示，如1/5。

T2.45

小数 二进制 十进制

3/4 0.11 0.75

25/16 1.1001 1.5625

43/16 10.1011 2.6875

9/8 1.001 1.125

47/8 101.111 5.875

51/16 11.0011 3.1875

**2.4.2IEEE浮点表示**

浮点数

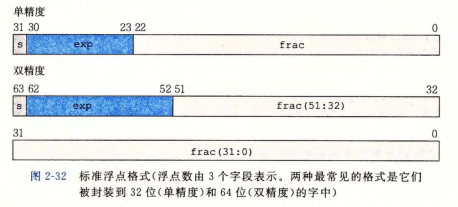
s符号位，1负0正。

M尾数，二进制小数。

E阶码。

单精度（32位）下，符号位1位，阶码8位，尾数23位。

双精度（64位）下，符号位1位，阶码11位，位数52位。



（1）阶码位模式不全为0且不全为1（0<exp<255或0<exp<2047）时为规格化值

此时阶码E=exp-bias，bias=2^(k-1)-1=127或1023。

小数字段frac=0.fn-1fn-2...f0，尾数M=1+frac

（2）阶码域全0（exp=0），非规格化

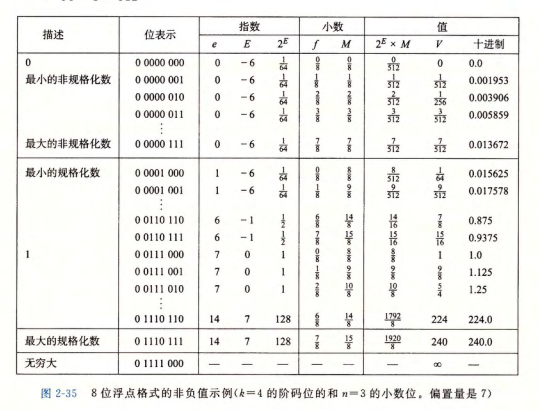
此时E=1-bias，M=f。

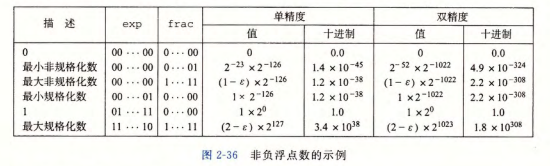
注意+0.0和-0.0除符号位不同外，其他位全为0。

（3）阶码域全1，特殊值

小数域全0时表示无穷，小数域非0时表示NaN

**2.4.3数字示例**





T2.47

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 位 | e | E | 2^E | f | M | 2^E\*M | V |
| 0 00 00 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 00 01 | 0 | 0 | 1 | 1/4 | 1/4 | 1/4 | 1/4 |
| 0 00 10 | 0 | 0 | 1 | 1/2 | 1/2 | 1/2 | 1/2 |
| 0 00 11 | 0 | 0 | 1 | 3/4 | 3/4 | 3/4 | 3/4 |
| 0 01 00 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| 0 01 01 | 1 | 0 | 1 | 1/4 | 5/4 | 5/4 | 5/4 |
| 0 01 10 | 1 | 0 | 1 | 1/2 | 3/2 | 3/2 | 3/2 |
| 0 01 11 | 1 | 0 | 1 | 3/4 | 7/4 | 7/4 | 7/4 |
| 0 10 00 | 2 | 1 | 2 | 0 | 1 | 2 | 2 |
| 0 10 01 | 2 | 1 | 2 | 1/4 | 5/4 | 5/2 | 5/2 |
| 0 10 10 | 2 | 1 | 2 | 1/2 | 3/2 | 3 | 3 |
| 0 10 11 | 2 | 1 | 2 | 3/4 | 7/4 | 7/2 | 7/2 |
| 0 11 00 | - | - | - | - | - | - | 无穷 |
| 0 11 01 | - | - | - | - | - | - | NaN |
| 0 11 10 | - | - | - | - | - | - | NaN |
| 0 11 11 | - | - | - | - | - | - | NaN |

**2.4.4舍入**

向偶数舍入、向零舍入、向下舍入、向上舍入。

**2.4.5浮点运算**

把浮点值x,y看成实数，而某个运算定义在实数上，计算将产生Round(xy)。

注意，浮点运算是可交换的，但是是不可结合的。

**2.4.6c语言中的浮点数**

从int转化为float,不会溢出，可能舍入。

int或float转化为double，能保留精确值。

double转化为float，可能溢出或舍入。

float或double转化为int，会向零舍入，可能溢出。

T2.54

A.真

B.假，int有32位精度，float只有24位精度，如x为Tmax时

C.假

D.真

E.真

F.真

G.真

H.假，f=1.0e20，d=1.0时，f+d会舍入到1e20

T2.60

unsigned replace\_byte(unsigned x,int i,unsigned char b){

return x-((char)(x>>8\*i)<<8\*i)+(b<<8\*i);

}

T2.65



T2.67

A.左移32位超过了int的范围，实际执行时相当于左移0位。

B.将beyond\_msb修改为1<<31<<1。

C.将set\_msb改为1<<15<<15<<1，beyond\_msb改为1<<15<<15<<2。

T2.68

#define w 8\*sizeof(int)

int lower\_one\_mask(int n){

return -1-(1<<(w-1)>>(w-n)<<1);

//return (unsigned)-1>>(w-n);

}

**第三章**

**3.2**

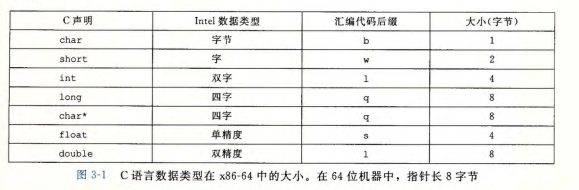
**3.2.1机器级代码**

程序计数器（PC），在x86-64中为%rip，给出要执行的下一条指令在内存中的地址。

整数寄存器、条件码寄存器。

**3.3**

Intel用“字”表示16位数据类型。



**3.4**

x86-64的CPU中包含16个64位的通用目的寄存器：



对于生成小于8字节结果的指令，生成1字节和2字节数字的指令会保持剩下的字节不变，生成4字节指令会将高位4字节置零。

**3.4.1操作数指示符**

操作数可分为3种：立即数、寄存器、内存引用。

寻址模式：

Imm(rb,ri,s)是最常用的形式，其中s必须是1，2，4或8。



T3.1

值：

0x100

0xAB

0x108

0xFF

0xAB

0x11

0x13

0xFF

0x11

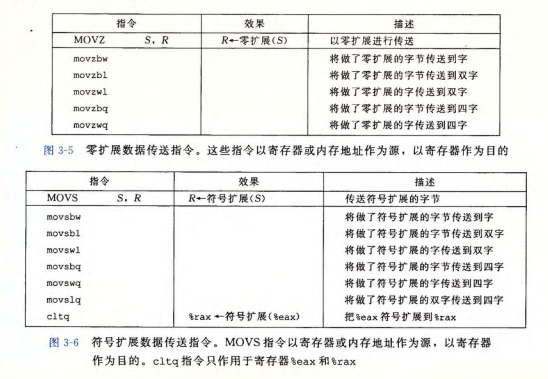
**3.4.2数据传送指令**

MOV类，将数据从原位置复制到目的位置，不做任何变化。



第一个操作数是原操作数，第二个是目的操作数。x86-64中，两个操作数不能都指向内存位置。寄存器操作数的寄存器部分大小必须与指令最后一个字符指定大小相匹配。

将较小的原的值复制到较大的目的，MOVZ类进行零扩展，MOVS进行符号扩展。



（PS：之所以没有movzlq，是因为movl即可完成相应功能。）

cltq指令没有操作数。

T3.2

movl

movw

movb

movb

movq

movw

T3.3

寻址时应为（%rbx）

应为movq

x86-64下两个操作数不能都指向内存单元

%sl不是整数寄存器

立即数不能作为目的操作数

应使用movl %eax %edx（零拓展）或movslq %eax %rdx （符号拓展）

应为%sil

**3.4.3数据传输示例**

间接引用指针就是将指针储存在寄存器中，然后在内存引用中使用这个寄存器。

T3.4

char -> int movsbl (%rdi) %eax

movl %eax (%rsi)

char -> unsigned movsbl (%rdi) %eax

movl %eax (%rsi)

unsigned char -> long movzbq (%rdi) %rax

movq %rax (%rsi)

//先从内存中读取一个int（4字节），再转化为char

int -> char movl (%rdi) %eax

movb %al (%rsi)

unsigned -> unsigned char movl (%rdi) %eax

movb %al (%rsi)

char -> short movsbw (%rdi) %ax

movw %ax (%rsi)

T3.5

void decode1(long \*xp, long \*yp, long \*zp){

long a,b,c;

a=\*xp;

b=\*yp

c=\*zp

\*yp=a

\*zp=b

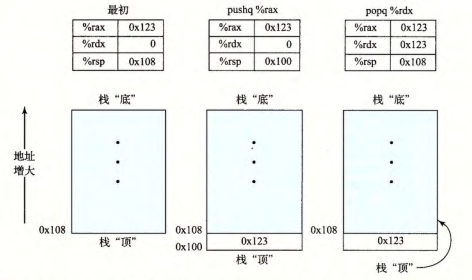
\*xp=c

}

**3.4.4压入和弹出栈数据**



x86-64中，栈向低地址方向生长，故压栈是减小栈指针，出栈是增加栈指针。



**3.5算数和逻辑操作**



**3.5.1加载有效地址**

leaq将有效地址写入到目的操作数。

T3.6

6+x%(2^16)

x+y

x+4y

7+9x

0xA+4y

9+x+2y

T3.7

t=5\*x+2\*y+8\*z

**3.5.2一元和二元操作**

略

**3.5.3移位操作**

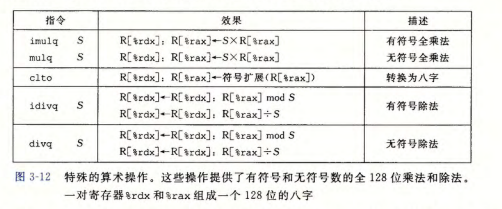
位移量k为立即数或存放在单字节寄存器%cl中，对w位数据进行位移的位移量由寄存器的低m位决定，其中2^m=w。

T3.9

salq $4 %rax

sarq %cl %rax

**3.5.5特殊算数操作**



**3.6控制**

**3.6.1条件码**

CF:进位标志，最高位进位，无符号溢出

ZF:零标志，最近操作结果为0

SF:符号标志，最近操作结果为负

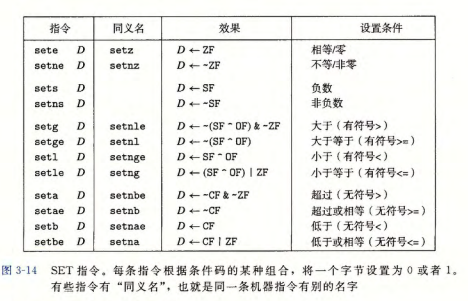
OF:溢出标志，最近操作导致补码溢出（正或负）

除leaq，图3.10中操作都会改变条件码。

cmp与sub，test与and功能相同，但不改变寄存器的值，只改变条件码。

**3.6.2访问条件码**

set指令，操作数是8位寄存器。



**3.6.3跳转指令**

直接跳转，跳转到某跳转标号；间接跳转，写法是“\*”后跟一个操作数指示符。



**3.6.4跳转指令的编码**

最常见的是PC-relative，即将目标指令地址与跳转指令的后一条地址的差值作为编码。

也可直接给出绝对地址。

与第七章“链接”相似。

T3.15

4003fe

400425

400543 400545

400560

**3.6.5用条件控制实现条件分支**

使用“go to”代码。

T3.18

x+y+z

x<-3

y>=z

y\*z

x\*y

x>2

x\*z

**3.6.6用条件传送实现条件分支**



T3.20

x OP y (x>=0?x:(7+x))/8

arith:

leaq 7(%rdi), %rax res=7+x

testq %rdi, %rdi if(rdi>=0)

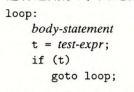
cmovns %rdi, %rax res=x

sarq $3, %rax res>>3

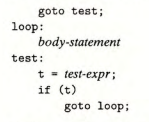
ret

**3.6.7循环**

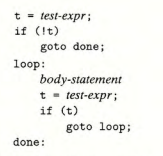
do-while循环转化为：



while循环转化为（jump to middle版本）：



或guarded-do版本：



for循环可转化为初始化语句+while循环。

T3.26

A.采用jump to middle方法

B.while(x!=0){  
 val^=x;

x/=2;

}

return val&1;

C.得到x各位相异或的结果，可以判断x的位模式中1的奇偶

T3.27

long fact\_for\_while(long n){  
 long i;

long result=1;

i=2;

while(i<=n){

result\*=i;

i++;

}

return result;

}

goto代码：

long fact\_for\_while(long n){  
 long i;

long result=1;

i=2;

if(i>n)

goto done;

loop:

result\*=i;

i++;

if(i<=n)

goto loop;

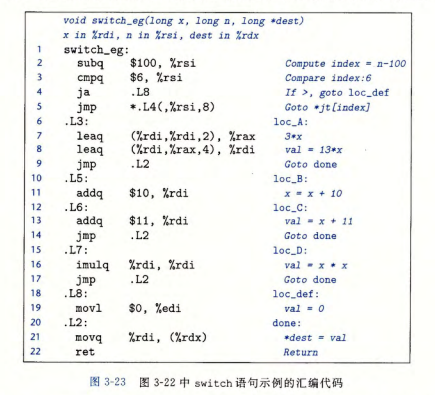
done:

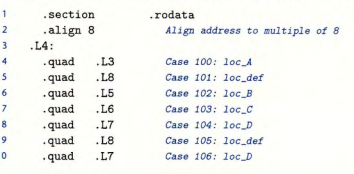
return result;

}

**3.6.8switch语句**

使用跳转表实现多重分支。





T3.31

case 5:  
 c=b^15;

case 0:

val=112+c;

break;

case 2:

case 7:

val=(b+c)\*4;

break;

case 4:

val=a;

break;

devault:

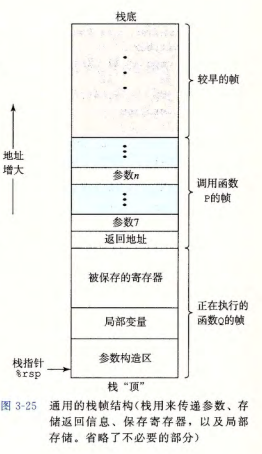
val=b;

**3.7过程**

**3.7.1运行时栈**

P调用Q时，P以及所有向上追溯到P的调用链都暂时被挂起，Q运行时只需要为局部变量分配新的存储空间。另一方面，Q返回时，任何它所分配的存储空间都被释放。

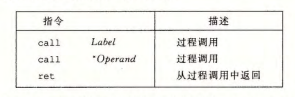
栈帧：通过寄存器，过程P最多传递6个整数值，但如果Q需要更多参数，P可以调用Q之前在自己的栈帧里存储好的参数。



**3.7.2转移控制**

call Q会将地址A压入栈中，并将PC设置为Q的起始地址，A被称为返回地址。

ret会从栈中弹出A，并把PC设置为A。

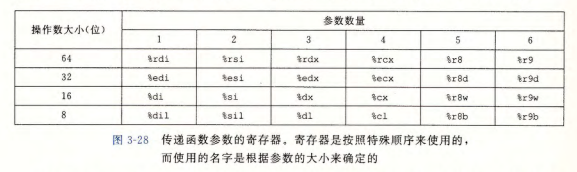


T3.32

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令 | | | 状态值 | | | | | |
| 标号 | PC | 指令 | %rdi | %rsi | %rax | %rsp | \*%rsp | 描述 |
| F1 | 400548 | lea | 10 | - | - | 0x7fffffffe818 | 400565 | X+1 |
| F2 | 40054c | sub | 10 | 11 | - | 0x7fffffffe818 | 400565 | X-1 |
| F3 | 400550 | callq | 9 | 11 | - | 0x7fffffffe818 | 400565 | 调用last(9,11) |
| L1 | 400540 | mov | 9 | 11 | - | 0x7fffffffe810 | 400555 | u |
| L2 | 400543 | imul | 9 | 11 | 9 | 0x7fffffffe810 | 400555 | u\*v |
| L3 | 400547 | retq | 9 | 11 | 99 | 0x7fffffffe810 | 400555 | 返回9 |
| F4 | 400555 | repz retq | 9 | 11 | 99 | 0x7fffffffe818 | 400565 | 返回9 |
| M2 | 400565 | mov | 9 | 11 | 99 | 0x7fffffffe820 | - | - |

**3.7.3数据传送**

x86-64中，可以通过寄存器最多传递6个整形参数。



若有大于6个参数，则超出6的部分用栈传递，为P代码分配的栈帧须要能容纳参数7~n，且参数7位于栈顶，参数7~n分别存储在%rsp+8~%rsp+8\*(n-6)。

通过栈调用参数时，所有数据大小都向8的倍数对齐。

注意从内存中读入小于4字节的变量时，一般从内存中读入4字节，再取其低位。

T3.33

(int a,short b,long \*u,char \*v)

或(int b,short a,long \*v,char \*u)

**3.7.4栈上的局部存储**

寄存器不足以存放局部变量，或对局部变量使用&生成地址时，需用栈上局部存储。

生成地址时，原变量存储在栈中，用leaq将地址存储在寄存器或先存储在寄存器再移动到内存中。

**3.7.5寄存器中的局部存储空间。**

寄存器组是唯一被所有过程共享的资源。

%rbx，%rbp和%r12~%r15是被调用者保存寄存器，即在返回时这些寄存器的值需保持与调用时相同。

其他寄存器是调用者保存寄存器，在调用的过程中可任意修改。

**3.7.6递归过程**

T3.35

A.x

B.if(x==0)

return 0;

unsigned long nx=x>>2;

long rv=rfun(nx);

return x+rv;

**3.8数组的分配与访问**

**3.8.1基本原则**

对于T A[N]，xA为起始地址，则第i个元素的位置为xA+L\*i。

T3.36

元素大小 整个数组大小 元素i

2 14 xs+2\*i

8 24 xt+8\*i

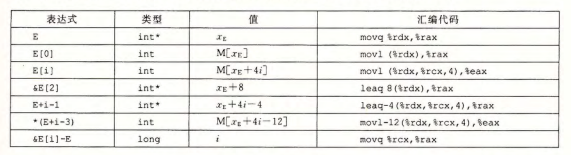
8 48 xu+8\*i

4 32 xv+4\*i

8 32 xw+8\*i

**3.8.2指针运算**

c语言中，指针p指向数据类型为T的数据的指针，则表达式p+i的值为xp+i\*L。



T3.37

类型 值 汇编代码

short\* xs+2 leaq 2(%rdx) %rax

short M[xs+6] movw 6(%rdx) %ax

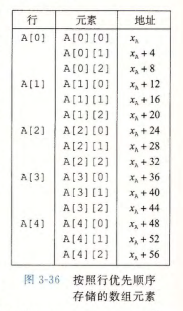
short\* xs+2\*i leaq (%rdx,%rcx,2) %rax

short M[xs+8\*i+2] movw 2(%rdx,%rcx,8) %ax

short\* xs+2\*i-10 leaq -10(%rdx,%rcx,2) %rax

**3.8.3嵌套的数组**

嵌套的数组按照行优先的顺序存储，&D[i][j]=x0+L(C\*i+j)。



T3.38

M=5，N=7

**3.9.3数据对齐**

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。

T3.44

0 1 2 4 8 12 16 24 32 40

A.i c j d

B.i c d j

C.w0 w1 w2 c0 c1 c2 （结构体作为元素时，对齐要求是其中最长元素的对齐要求，可能需要在结构的末尾进行填充）

D.w0 w1 w2 w3 w4 \*c0 \*c1 \*c2

E.a0 a1 t

（结构体作为元素时，对齐要求是其中最长元素的对齐要求）

T3.45

A.

char \*a 0

short b 8

double c 16

char d 24

float e 28

char f 32

long g 40

int h 48

B.

56字节，结尾须填充4字节满足8字节的对齐要求

C.

重排后：a,c,g,e,h,b,d,f

偏移量：0,8,16,24,28,32,34,35

总大小：40字节

**第十章**

**10.1unix I/O**

Linux中，所有io设备都被模型化为文件，输入输出相当于文件读写。

应用程序通过内核打开文件时，内核返回一个描述符。

Linux shell创建的每个进程开始时都有三个打开的文件：标准输入（描述符0），标准输出（描述符1），标准错误（描述符2）。

seek操作显示地设置文件当前位置k。

一个读操作就是从文件复制n>0个字节到内存，从当前文件位置k始，然后将k增加到k+n。若文件大小为m，当k>=m时会触发EOF。

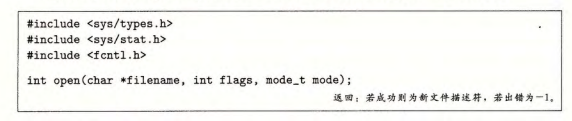
**10.2文件**

每个目录至少包括两个条目：“.”是到当前目录自身的链接，“..”是到父目录的链接。

可以用cd命令修改当前工作目录。

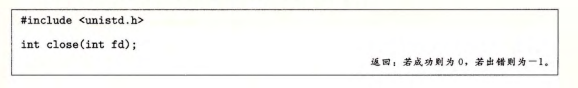
**10.3打开和关闭文件**

进程是用open函数来打开一个已存在的文件或者创建一个新文件的：



如：fd=open("foo.txt",O\_RDONLY,0);

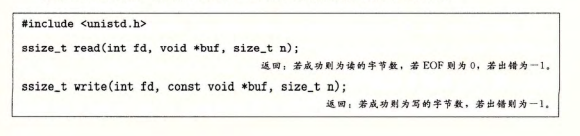
最后进程调用close函数关闭文件。



T10.1

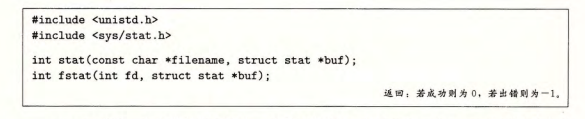
fd2=3

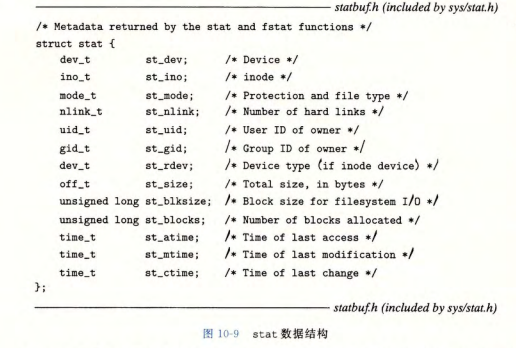
**10.4读和写文件**



**10.6读取文件元数据**

应用程序能调用stat和fstat来读取元数据，并填写表格：





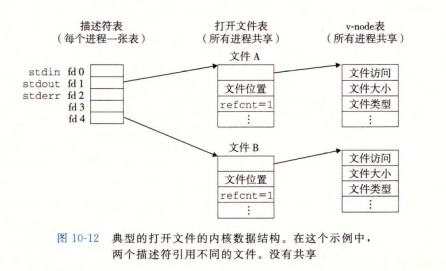
**10.8共享文件**

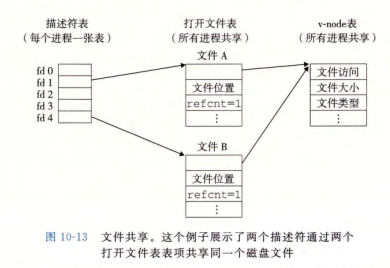
描述符表、文件表、v-node表，其中文件表与v-node表所有进程共享。

多个描述符可以通过不同的文件表来引用不同的文件或同一个文件，每个描述符有不同的文件位置，对不同描述符的读操作可从不同位置获取数据。

fork创建一个子进程，子进程继承父进程的描述符表。

（PS：fork调用一次返回两次，对子进程返回0，父进程返回子进程的PID）





T10.2

c=f

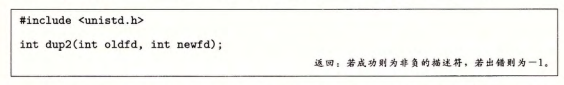
T10.3

c=o（父子进程共享一个文件表，子进程读取文件第一个字节时，文件位置加一，父进程读取文件第二个位置）

**10.9I/O重定向**

命令行输入输出重定向：输入“<”，输出">"。

dup2函数：



dup2函数复制描述符表表项oldfd到描述符表表项newfd，覆盖描述符表表项newfd以前的内容。若newfd已被打开，则关闭之，并删除对应的文件表和v-node表。

T10.4

dup2(5,0);

T10.5

c=o

**10.10标准I/O**

语言定义了一组高级输入输出函数，称为标准I/O库，为程序员提供了UnixI/O的较高级别的替代。

标准I/O将打开的文件模型化为一个流。

注意，不能在二进制文件中使用fgets，scanf等io函数，以及string.h中的字符处理函数。二进制文件中的0在前者中会被当做EOL，在后者中被作为end of string。

**第七章**

**7.1编译器驱动程序**

main.c-->c预处理器（cpp）得ASCII中间文件main.i-->c编译器（ccl）得ASCII汇编语言文件main.s-->汇编器（as）得可重定位的目标文件main.o-->链接器（ld）可执行目标文件prog

**7.3目标文件**

可重定位、可执行、共享目标文件

**7.4可重定位目标文件**

elf文件的基本格式：



.text已编译程序的机器代码。

.data已初始化的全局和静态变量。局部变量在运行时被保存在栈中，既不出现在.data节中，也不出现在.bss节中。

.bss未初始化的全局和静态变量，以及所有被初始化为0的全局或静态变量。不占据实际空间。

.symtab一个符号表，它存放在程序中定义和引用的函数和全局变量的信息，每个可重定位目标文件在.symtab中都有一张符号表。

**7.5符号和符号表**

全局符号、外部符号、局部符号。

在同一模块的两个函数各自定义了一个静态局部变量x时，编译器向汇编器输出两个不同名的局部连接器符号，如x.1,x.2。

.symtab中包含ELF符号表：



section中有三个特殊伪节，ABS、UNDEF和COMMON。

COMMON：未初始化的全局变量。

.bss现在修改为：未初始化的静态变量，以及初始化为0的全局和静态变量。

**7.6符号解析**

**7.6.1解析多重定义的全局符号**

函数和已初始化的全局变量是强符号，未初始化的全局变量是弱符号。

规则1：不允许有多个同名的强符号。

规则2: 如果有一个强符号和多个弱符号同名，那么选择强符号。

规则3: 如果有多个弱符号同名，那么从这些弱符号中任意选择一个。

**7.6.2与静态库链接**

在链接时，链接器将只复制被程序引用的目标模块，这就减少了可执行文件在磁盘和内存中的大小。

**7.6.3链接器如何使用静态库来解析引用**

维护集合E（所有参与形成可执行文件的文件），U（被引用但尚未定义的符号），D（已定义的符号）。

对输入文件f：

f是目标文件：f添加到E，修改U、D。

f是存档文件：对f的每个成员m，若m含有U中符号的定义，则将m加到E，修改U、D，否则丢弃m。

完成所有扫描后，若U非空，则报错。

库一般放在命令行末尾，若库之间相互调用，则被调用的库置于调用库之后。

eg：gcc foo.c libx.a liby.a libx.a

**7.7重定位**

重定位节和符号的定义、重定位节中符号的引用。

**7.7.1重定位条目**

代码的重定位条目在.rel.text中，已初始化数据的重定位条目在.rel.data中。

**7.7.2重定位符号引用**

PC相对引用、绝对引用。

7.10动态链接共享库