习 题

3. 参考答案：

（1）后缀：w，　源：基址+比例变址+偏移，　目：寄存器

（2）后缀：b，　源：寄存器，　目：基址+偏移

（3）后缀：l，　源：比例变址，　目：寄存器

（4）后缀：b，　源：基址，　目：寄存器

（5）后缀：l，　源：立即数，　目：栈

（6）后缀：l，　源：立即数，　目：寄存器

（7）后缀：w，　源：寄存器，　目：寄存器

（8）后缀：l，　源：基址+变址+偏移，　目：寄存器

4. 参考答案：

（1）源操作数是立即数 0xFF，需在前面加‘$’

（2）源操作数是 16 位，而长度后缀是字节‘b’，不一致

（3）目的操作数不能是立即数寻址

（4）操作数位数超过 16 位，而长度后缀为 16 位的‘w’

（5）不能用 8 位寄存器作为目的操作数地址所在寄存器

（6）源操作数寄存器与目操作数寄存器长度不一致

（7）不存在 ESX 寄存器

（8）源操作数地址中缺少变址寄存器

5. 参考答案：

表 **3.12** 题 **5** 用表

src_type	dst_type	机器级表示
char	int	movsbl %al, (%edx)
int	char	movb %al, (%edx)
int	unsigned	movl %eax, (%edx)
short	int	movswl %ax, (%edx)
unsigned char	unsigned	movzbl %al, (%edx)
char	unsigned	movsbl %al, (%edx)
int	int	movl %eax, (%edx)

6. 参考答案：

（1）xptr、yptr 和 zptr 对应实参所存放的存储单元地址分别为：R[ebp]+8、R[ebp]+12、R[ebp]+16。

（2）函数 func 的 C 语言代码如下：

```
void func(int *xptr, int *yptr, int *zptr)
{
    int tempx=*xptr;
    int tempy=*yptr;
    int tempz=*zptr;

    *yptr=tempx;
    *zptr = tempy;
    *xptr = tempz;
}
```

7．参考答案：

（1）R[edx]=x

（2）R[edx]=x+y+4

（3）R[edx]=x+8*y

（4）R[edx]=y+2*x+12

（5）R[edx]=4*y

（6）R[edx]=x+y

8．参考答案：

（1）指令功能为：R[edx]←R[edx]+M[R[eax]]=0x00000080+M[0x8049300]，寄存器 EDX 中内容改变。改变后的内容为以下运算的结果：00000080H+FFFFFFF0H

```
  0000 0000 0000 0000 0000 0000 1000 0000
+ 1111 1111 1111 1111 1111 1111 1111 0000
-----------------------------------------
1 0000 0000 0000 0000 0000 0000 0111 0000
```

因此，EDX 中的内容改变为 0x00000070。根据表 3.5 可知，加法指令会影响 OF、SF、ZF 和 CF 标志。OF=0，ZF=0，SF=0，CF=1。

（2）指令功能为：R[ecx]←R[ecx]−M[R[eax]+R[ebx]]=0x00000010+M[0x8049400]，寄存器 ECX 中内容改变。改变后的内容为以下运算的结果：00000010H−80000008H

```
  0000 0000 0000 0000 0000 0000 0001 0000
+ 0111 1111 1111 1111 1111 1111 1111 1000
-----------------------------------------
0 1000 0000 0000 0000 0000 0000 0000 1000
```

因此，ECX 中的内容改为 0x80000008。根据表 3.5 可知，减法指令会影响 OF、SF、ZF 和 CF 标志。OF=1，ZF=0，SF=1，CF=1⊕0=1。

（3）指令功能为：R[bx]←R[bx] or M[R[eax]+R[ecx]*8+4]，寄存器 BX 中内容改变。改变后的内容为以下运算的结果：0x0100 or M[0x8049384]=0100H or FF00H

```
  0000 0001 0000 0000
or 1111 1111 0000 0000
----------------------
1111 1111 0000 0000
```

因此，BX 中的内容改为 0xFF00。由 3.3.3 节可知，OR 指令执行后 OF=CF=0；因为结果不为 0，故 ZF=0；因为最高位为 1，故 SF=1。

（4）test 指令不改变任何通用寄存器，但根据以下“与”操作改变标志：R[dl] and 0x80

```
  1000 0000
and 1000 0000
-------------
1000 0000
```

由 3.3.3 节可知，TEST 指令执行后 OF=CF=0；因为结果不为 0，故 ZF=0；因为最高位为 1，故 SF=1。

（5）指令功能为：M[R[eax]+R[edx]]←M[R[eax]+R[edx]]*32，即存储单元 0x8049380 中的内容改变为以下运算的结果：M[0x8049380]*32=0x908f12a8*32，也即只要将 0x908f12a8 左移 5 位即可得到结果。

1001 0000 1000 1111 0001 0010 1010 1000<<5

=0001 0001 1110 0010 0101 0101 0000 0000

因此，指令执行后，单元 0x8049380 中的内容改变为 0x11e25500。显然，这个结果是溢出的。但是，根据表 3.5 可知，乘法指令不影响标志位，也即并不会使 OF=1。

（6）指令功能为：R[cx] ←R[cx]−1，即 CX 寄存器的内容减一。

$$\begin{array}{r} 0000\ 0000\ 0001\ 0000 \\ +\quad 1111\ 1111\ 1111\ 1111 \\ \hline 1\ 0000\ 0000\ 0000\ 1111 \end{array}$$

因此，指令执行后 CX 中的内容从 0x0010 变为 0x000F。由表 3.5 可知，DEC 指令会影响 OF、ZF、SF，根据上述运算结果，得到 OF=0，ZF=0，SF=0。

9．参考答案：

```
movl  12(%ebp), %ecx     //R[ecx]←M[R[ebp]+12]，将 y 送 ECX
sall    $8, %ecx         //R[ecx]←R[ecx]<<8，将 y*256 送 ECX
movl  8(%ebp), %eax      //R[eax]←M[R[ebp]+8]，将 x 送 EAX
movl  20(%ebp), %edx     //R[edx]←M[R[ebp]+20]，将 k 送 EDX
imull  %edx, %eax        //R[eax]←R[eax]*R[edx]，将 x*k 送 EAX
movl  16(%ebp), %edx     //R[edx]←M[R[ebp]+16]，将 z 送 EDX
andl   $65520, %edx      //R[edx]←R[edx] and 65520，将 z&0xFFF0 送 EDX
addl   %ecx, %edx        //R[edx]←R[edx] + R[ecx]，将 z&0xFFF0+y*256 送 EDX
subl   %edx, %eax        //R[eax]←R[eax]-R[edx]，将 x*k-(z&0xFFF0+y*256)送 EAX
```

根据以上分析可知，第 3 行缺失部分为：

3　　int　v = x*k-(z&0xFFF0+y*256) ;

10．参考答案：

从汇编代码的第 2 行和第 4 行看，y 应该是占 8 个字节，R[ebp]+20 开始的 4 个字节为高 32 位字节，记为 y_h；R[ebp]+16 开始的 4 个字节为低 32 位字节，记为 y_l。根据第 4 行为无符号数乘法指令，得知 y 的数据类型 num_type 为 unsigned long long。

```
movl      12(%ebp), %eax     //R[eax]←M[R[ebp]+12]，将 x 送 EAX
movl      20(%ebp), %ecx     //R[ecx]←M[R[ebp]+20]，将 yh 送 ECX
imull     %eax, %ecx         //R[ecx]←R[ecx]*R[eax]，将 yh*x 的低 32 位送 ECX
mull      16(%ebp)           //R[edx]R[eax]←M[R[ebp]+16]*R[eax]，将 yl*x 送 EDX-EAX
leal      (%ecx, %edx), %edx
          // R[edx]←R[ecx]+R[edx]，将 yl*x 的高 32 位与 yh*x 的低 32 位相加后送 EDX
movl      8(%ebp), %ecx      //R[ecx]←M[R[ebp]+8]，将 d 送 ECX
movl      %eax, (%ecx)       //M[R[ecx]]←R[eax]，将 x*y 低 32 位送 d 指向的低 32 位
movl      %edx, 4(%ecx)      //M[R[ecx]+4]←R[edx]，将 x*y 高 32 位送 d 指向的高 32 位
```

11．参考答案：

根据第 3.3.4 节得知，条件转移指令都采用相对转移方式在段内直接转移，即条件转移指令的转移目标地址为：（PC）+偏移量。

（1）因为 je 指令的操作码为 01110100，所以机器代码 7408H 中的 08H 是偏移量，故转移目标地址为：0x804838c+2+0x8=0x8048396。

call 指令中的转移目标地址 0x80483b1=0x804838e+5+0x1e，由此，可以看出，call 指令机器代码中后面的 4 个字节是偏移量，因 IA-32 采用小端方式，故偏移量为 0000001EH。call 指令机器代码共占 5 个字节，因此，下条指令的地址为当前指令地址 0x804838e 加 5。

（2）jb 指令中 F6H 是偏移量，故其转移目标地址为：0x8048390+2+0xf6=0x8048488。

movl 指令的机器代码有 10 个字节，前两个字节是操作码等，后面 8 个字节为两个立即数，因为是小端方式，所以，第一个立即数为 0804A800H，即汇编指令中的目的地址 0x804a800，最后 4 个字节为立即数 00000001H，即汇编指令中的常数 0x1。

（3）jle 指令中的 7EH 为操作码，16H 为偏移量，其汇编形式中的 0x80492e0 是转移目的地址，因此，假定后面的 mov 指令的地址为 *x*，则 *x* 满足以下公式：0x80492e0=*x*+0x16，故 *x*=0x80492e0−0x16=0x80492ca。

（4）jmp 指令中的 E9H 为操作码，后面 4 个字节为偏移量，因为是小端方式，故偏移量为 FFFFFF00H，即−100H=−256。后面的 sub 指令的地址为 0x804829b，故 jmp 指令的转移目标地址为 0x804829b+0xffffff00=0x804829b−0x100=0x804819b。

12．参考答案：

（1）汇编指令的注解说明如下：

```
1     movb    8(%ebp), %dl        //R[dl]←M[R[ebp]+8]，将 x 送 DL
2     movl    12(%ebp), %eax      //R[eax]←M[R[ebp]+12]，将 p 送 EAX
3     testl   %eax, %eax          //R[eax] and R[eax]，判断 p 是否为 0
4     je      .L1                 //若 p 为 0，则转.L1 执行
5     testb   $0x80, %dl          //R[dl] and 80H，判断 x 的第一位是否为 0
6     je      .L1                 //若 x>=0，则转.L1 执行
7     addb    %dl, (%eax)         //M[R[eax]]←M[R[eax]]+R[dl]，即*p+=x
8  .L1:
```

因为 C 语言 if 语句中的条件表达式可以对多个条件进行逻辑运算，而汇编代码中一条指令只能进行一种逻辑运算，并且在每条逻辑运算指令生成的标志都是存放在同一个 EFLAGS 寄存器中，所以，最好在一条逻辑指令后跟一条条件转移指令，把 EFLAGS 中标志用完，然后再执行另一次逻辑判断并根据条件进行转移的操作。

（2）按照书中图 3.22 给出的“if () goto …”语句形式写出汇编代码对应的 C 语言代码如下：

```
1     void comp(char x, int *p)
2     {
3         if (p!=0)
4             if (x<0)
5                 *p += x;
6     }
```

13．参考答案：

```
1     int func(int x, int y)
2     {
3         int z =  x*y  ;
4         if (  x<=-100  ) {
5             if  ( y>x )
6                 z = x+y  ;
7         else
8             z = x-y  ;
9         } else if  ( x>=16 )
10            z = x &y  ;
11        return z;
12    }
```

14．参考答案：

（1）每个入口参数都要按 4 字节边界对齐，因此，参数 *x*、*y* 和 *k* 入栈时都占 4 个字节。

```
1     movw    8(%ebp), %bx        //R[bx]←M[R[ebp]+8]，将 x 送 BX
2     movw    12(%ebp), %si       //R[si]←M[R[ebp]+12]，将 y 送 SI
3     movw    16(%ebp), %cx       //R[cx]←M[R[ebp]+16]，将 k 送 CX
4  .L1:
```

```
5     movw    %si, %dx        //R[dx]←R[si]，将 y 送 DX
6     movw    %dx, %ax        //R[ax]←R[dx]，将 y 送 AX
7     sarw    $15, %dx        //R[dx]←R[dx]>>15，将 y 的符号扩展 16 位送 DX
8     idiv    %cx             //R[dx]←R[dx-ax]÷R[cx]的余数，将 y%k 送 DX
                              //R[ax]←R[dx-ax]÷R[cx]的商，将 y/k 送 AX
9     imulw   %dx, %bx        //R[bx]←R[bx]*R[dx]，将 x*(y%k) 送 BX
10    decw    %cx             //R[cx]←R[cx]-1，将 k-1 送 CX
11    testw   %cx, %cx        //R[cx] and R[cx]，得 OF=CF=0，负数则 SF=1,零则 ZF=1
12    jle     .L2             //若 k 小于等于 0，则转.L2
13    cmpw    %cx, %si        //R[si] - R[cx]，将 y 与 k 相减得到各标志
14    jg      .L1             //若 y 大于 k，则转.L1
15  .L2:
16    movswl  %bx, %eax       // R[eax]←R[bx]，将 x*(y%k) 送 AX
```

（2）被调用者保存寄存器有 BX、SI，调用者保存寄存器有 AX、CX 和 DX。

在该函数过程体前面的准备阶段，被调用者保存的寄存器 EBX 和 ESI 必须保存到栈中。

（3）因为执行第 8 行除法指令前必须先将被除数扩展为 32 位，而这里是带符号数除法，因此，采用算术右移以扩展 16 位符号，放在高 16 位的 DX 中，低 16 位在 AX 中。

15．参考答案：

```
1    int f1(unsigned x)
2    {
3        int y = 0 ;
4        while (   x!=0   ) {
5            y ^=x    ;
6            x>>=1   ;
7        }
8        return   y&0x1      ;
9    }
```

函数 f1 的功能返回：(x ^ x>>1 ^ x>>2 ^ ….) & 0x1，因此 f1 用于检测 x 的奇偶性，当 x 中有奇数个 1，则返回为 1，否则返回 0。

16．参考答案：

函数 sw 只有一个入口参数 *x*，根据汇编代码的第 2~5 行指令知，当 *x*+3>7 时转标号.L7 处执行，否则，按照跳转表中的地址转移执行，*x* 与跳转目标处标号的关系如下：

x+3=0：.L7
x+3=1：.L2
x+3=2：.L2
x+3=3：.L3
x+3=4：.L4
x+3=5：.L5
x+3=6：.L7
x+3=7：.L6

由此可知，switch (x) 中省略的处理部分结构如下：

```
case -2:
case -1:
    ……      // .L2 标号处指令序列对应的语句
    break;
case 0:
```

```
        ……      // .L3 标号处指令序列对应的语句
        break;
case 1:
        …….     // .L4 标号处指令序列对应的语句
        break;
case 2:
        ……      // .L5 标号处指令序列对应的语句
        break;
case 4:
        ……      // .L6 标号处指令序列对应的语句
        break;
default:
        ……      // .L7 标号处指令序列对应的语句
```

17．参考答案：

根据第 2、3 行指令可知，参数 a 是 char 型，参数 p 是指向 short 型变量的指针；根据第 4、5 行指令可知，参数 b 和 c 都是 unsigned short 型，根据第 6 行指令可知，test 的返回参数类型为 unsigned int。因此，test 的原型为：unsigned int test(char a, unsigned short b, unsigned short c, short *p);

18．参考答案：

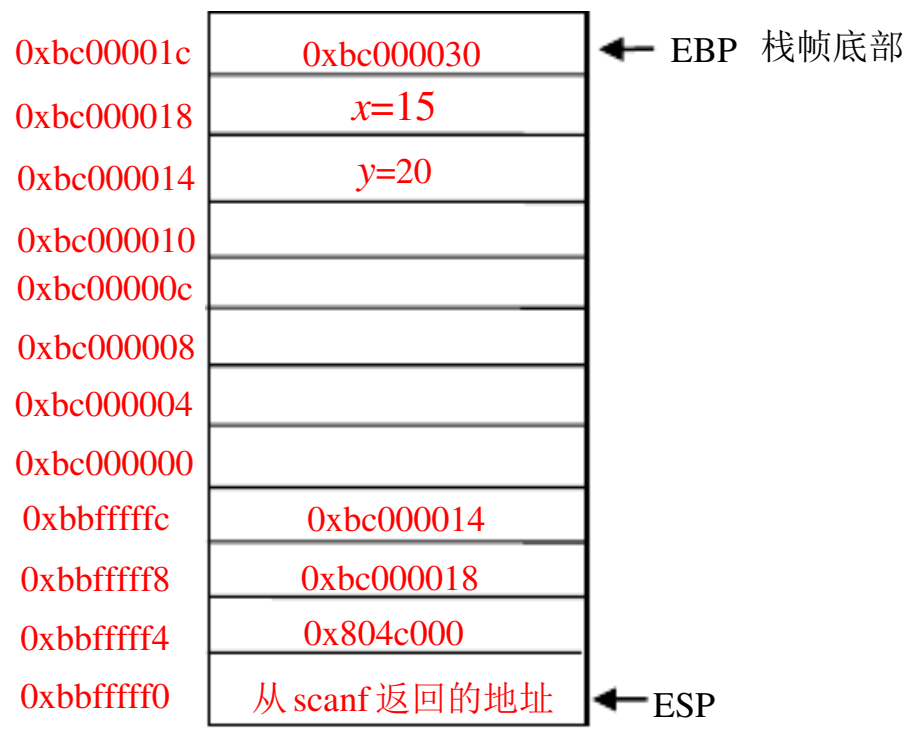
每次执行 pushl 指令后，R[esp]=R[esp]-4，因此，第 2 行指令执行后 R[esp]=0xbc00001c。

（1）执行第 3 行指令后，R[ebp]=R[esp]=0xbc00001c。到第 12 条指令执行结束都没有改变 EBP 的内容，因而执行第 10 行指令后，EBP 的内容还是为 0xbc00001c。执行第 13 行指令后，EBP 的内容恢复为进入函数 funct 时的值 0xbc000030。

（2）执行第 3 行指令后，R[esp]=0xbc00001c。执行第 4 行指令后 R[esp]= R[esp]-40=0xbc00001c-0x28=0xbbffffff4。因而执行第 10 行指令后，未跳转到 scanf 函数执行时，ESP 中的内容为 0xbbffffff4-4=0xbbffffff0；在从 scanf 函数返回后 ESP 中的内容为 0xbbffffff4。执行第 13 行指令后，ESP 的内容恢复为进入函数 funct 时的旧值，即 R[esp]=0xbc000020。

（3）第 5、6 两行指令将 scanf 的第三个参数&y 入栈，入栈的内容为 R[ebp]-8=0xbc000014；第 7、8 两行指令将 scanf 的第二个参数&x 入栈，入栈的内容为 R[ebp]-4=0xbc000018。故 x 和 y 所在的地址分别为 0xbc000018 和 0xbc000014。

（4）执行第 10 行指令后，funct 栈帧的地址范围及其内容如下：



19．参考答案：

第 1 行汇编指令说明参数 *x* 存放在 EBX 中，根据第 4 行判断 *x*=0 则转.L2，否则继续执行第 5~10 行指令。根据第 5、6、7 行指令可知，入栈参数 *nx* 的计算公式为 *x*>>1；根据第 9、10、11 行指令可知，返回值为(*x*&1)+*rv*。由此

推断出 C 缺失部分如下：

```
1    int refunc(unsigned x) {
2        if (   x==0   )
3            return    0    ;
4        unsigned nx =  x>>1   ;
5        int rv = refunc(nx) ;
6        return   (x & 0x1) + rv    ;
7    }
```

该函数的功能为计算 *x* 的各个数位中 1 的个数。

20．参考答案：

在 IA-32 中，GCC 为数据类型 long double 型变量分配 12 字节空间，实际上只占用 10 个字节。

数组	元素大小（B）	数组大小（B）	起始地址	元素 *i* 的地址
char A[10]	1	10	&A[0]	&A[0]+*i*
int B[100]	4	400	&B[0]	&B[0]+4*i*
short *C[5]	4	20	&C[0]	&C[0]+4*i*
short **D[6]	4	24	&D[0]	&D[0]+4*i*
long double E[10]	12	120	&E[0]	&E[0]+12*i*
long double *F[10]	4	40	&F[0]	&F[0]+4*i*

21．参考答案：

表达式	类型	值	汇编代码
S	short *	A_S	leal (%edx), %eax
S+i	short *	A_S+2*i	leal (%edx, %ecx, 2), %eax
S[i]	short	M[A_S+2*i]	movw (%edx, %ecx, 2), %ax
&S[10]	short *	A_S+20	leal 20(%edx), %eax
&S[i+2]	short *	A_S+2*i+4	leal 4(%edx, %ecx, 2), %eax
&S[i]-S	int	(A_S+2*i-A_S)/2=i	movl %ecx, %eax
S[4*i+4]	short	M[A_S+2*(4*i+4)]	movw 8(%edx, %ecx, 8), %ax
*(S+i-2)	short	M[A_S+2*(i-2)]	movw -4(%edx, %ecx, 2), %ax

22．参考答案：

根据汇编指令功能可以推断最终在 EAX 中返回的值为：

M[a+28*i+4*j]+M[b+20*j+4*i]，因为数组 a 和 b 都是 int 型，每个数组元素占 4B，因此，M=5, N=7。

23．参考答案：

执行第 11 行指令后，a[i][j][k]的地址为 a+4*(63*i+9*j+k)，所以，可以推断出 M=9，N=63/9=7。根据第 12 行指令，可知数组 a 的大小为 4536 字节，故 L=4536/(4*L*M)=18。

24．参考答案：

（1）常数 M=76/4=19，存放在 EDI 中，变量 *j* 存放在 ECX 中。

（2）上述优化汇编代码对应的函数 trans_matrix 的 C 代码如下：

```
1    void trans_matrix(int a[M][M]) {
2        int i, j, t, *p;
3        int c=(M<<2);
3        for (i = 0; i < M; i++) {
4            p=&a[0][i];
5            for (j = 0; j < M; j++) {
```

```
6                       t=*p;
7                       *p = a[i][j];
8                       a[i][j] = t;
9                       p += c;
10                  }
11          }
12  }
```

25．参考答案：

（1）node 所需存储空间需要 4+(4+4)+4=16 字节。成员 p、s.x、s.y 和 next 的偏移地址分别为 0、4、8 和 12。

（2）np_init 中缺失的表达式如下：

```
void np_init(struct node *np)
{
    np->s.x = __np->s.y____ ;
    np->p = __&(np->s.x)___ ;
    np->next= __np___ ;
}
```

26．参考答案：

表达式 EXPR	TYPE 类型	汇编指令序列
uptr->s1.x	int	movl (%eax), %eax movl %eax, (%edx)
uptr->s1.y	short	movw 4(%eax), %ax movw %ax, (%edx)
&uptr->s1.z	short *	leal 6(%eax), %eax movw %eax, (%edx)
uptr->s2.a	short *	movl %eax, (%edx)

uptr->s2.a[uptr->s2.b]	short	movl 4(%eax), %ecx
		movl (%eax, %ecx, 2), %eax
		movl %eax, (%edx)

*uptr->s2.p	char	movl 8(%eax), %eax
		movb (%eax), %al
		movb %al, (%edx)

27．参考答案：

（1）S1:　s　c　i　d

0　2　4　8　总共 12 字节，按 4 字节边界对齐

（2）S2:　i　s　c　d

0　4　6　7　总共 8 字节，按 4 字节边界对齐

（3）S3:　c　s　i　d

0　2　4　8　总共 12 字节，按 4 字节边界对齐

（4）S4:　s　c

0　6　总共 8 字节，按 2 字节边界对齐

（5）S5:　c　s　i　d　e

0　4　8　12　16　总共 24 字节，按 4 字节边界对齐（Linux 下 double 型按 4 字节对齐）

（6）S6:　c　s　d

0　36　40　总共 44 字节，按 4 字节边界对齐

28．参考答案：

Windows 平台要求不同的基本类型按照其数据长度进行对齐。每个成员的偏移量如下：

c	d	i	s	p	l	g	v
0	8	16	20	24	28	32	40

结构总大小为 48 字节，因为其中的 d 和 g 必须是按 8 字节边界对齐，所以，必须在末尾再加上 4 个字节，即 44+4=48 字节。变量长度按照从大到小顺序排列，可以使得结构所占空间最小，因此调整顺序后的结构定义如下：

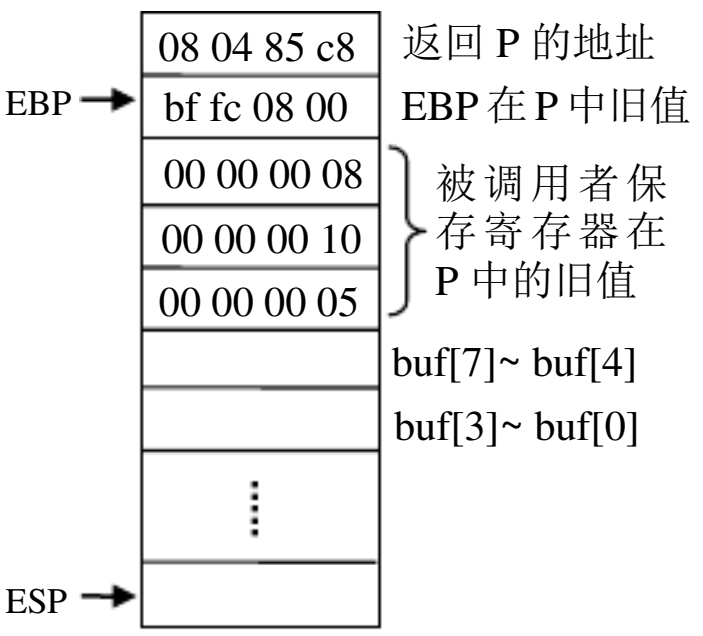
```
struct {
        double      d;
        long long   g;
        int         i;
        char        *p;
        long        l;
        void        *v;
        short       s;
        char        c;
    } test;
```

d	g	i	p	l	v	s	c
0	8	16	20	24	28	32	34

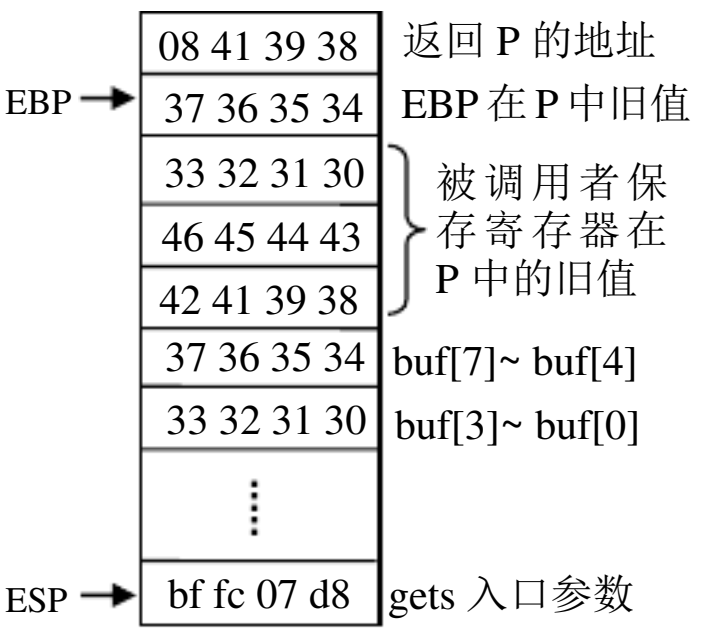
结构总大小为 34+6=40 字节。

29. 参考答案：

（1）执行第 7 行和第 10 行指令后栈中的信息存放情况如下图所示。其中 gets 函数的入口参数为 buf 数组首地址，应等于 getline 函数的栈帧底部指针 EBP 的内容减 0x14，而 getline 函数的栈帧底部指针 EBP 的内容应等于执行完 getline 中第 2 行指令（push %ebp）后 ESP 的内容，此时，R[esp]==0xbffc07f0−4=0xbffc07ec，故 buf 数组首地址为 R[ebp]−0x14= R[esp]−0x14=0xbffc07ec−0x14=0xbffc07d8。



a) 执行第 7 行后的栈



b) 执行第 10 行后的栈

（2）当执行到 getline 的 ret 指令时，假如程序不发生段错误，则正确的返回地址应该是 0x80485c8，发生段错误是因为执行 getline 的 ret 指令时得到的返回地址为 0x8413938，这个地址所在存储段可能是不可执行的数据段，因而发生了段错误（segmentation fault）。

（3）执行完第 10 行汇编指令后，被调用者保存寄存器 EBX、ESI 和 EDI 在 P 中的内容已被破坏，同时还破坏了 EBP 在 P 中的内容。

（4）getline 的 C 代码中 malloc 函数的参数应该为 strlen(buf)+1，此外，应该检查 malloc 函数的返回值是否为 NULL。

30. 参考答案：

x86-64 过程调用时参数传递是通过通用寄存器进行的，前三个参数所用寄存器顺序为 RDI、RSI、RDX。

abc 的 4 种合理的函数原型为：

① viod abc(int c, long *a, int *b);

② viod abc(unsigned c, long *a, int *b);

③ viod abc(long c, long *a, int *b);

④ viod abc(unsigned long c, long *a, int *b);

根据第 3、4 行指令可知，参数 b 肯定指向一个 32 位带符号整数类型；根据第 5 行指令可知，参数 a 指向 64 位带符号整数类型；而参数 c 可以是 32 位，也可以是 64 位，因为*b 为 32 位，所以取 RDI 中的低 32 位 R[edi]（截断为 32 位），再和*b 相加。同时，参数 c 可以是带符号整数类型，也可以是无符号整数类型，因为第 2 行加法指令 addl 的执行结果对于带符号整数和无符号整数都一样。

31．参考答案：

（1）汇编指令注释如下：

```
1    movl    8(%ebp), %edx          //R[edx]←M[R[ebp]+8]，将 x 送 EDX
2    movl    12(%ebp), %ecx         //R[ecx]←M[R[ebp]+12]，将 k 送 ECX
3    movl    $255, %esi             //R[esi]←255，将 255 送 ESI
4    movl    $-2147483648, %edi     //R[edi]←-2147483648，将 0x80000000 送 EDI
5    .L3:
6    movl    %edi, %eax             //R[eax]←R[edi]，将 i 送 EAX
7    andl    %edx, %eax             //R[eax]←R[eax] and R[edx]，将 i and x 送 EAX
8    xorl    %eax, %esi             //R[esi]←R[esi] xor R[eax]，将 val xor (i and x)送 ESI
9    movl    %ecx, %ebx             //R[ebx]←R[ecx]，将 k 送 ECX
10   shrl    %bl, %edi              //R[edi]←R[edi] >> R[bl]，将 i 逻辑右移 k 位送 EDI
11   testl   %edi, %edi
12   jne     .L3                    //若 R[edi]≠0，则转.L3
13   movl    %esi, %eax             //R[eax]←R[esi]
```

（2）*x* 和 *k* 分别存放在 EDX 和 ECX 中。局部变量 *val* 和 *i* 分别存放在 ESI 和 EDI 中。

（3）局部变量 *val* 和 *i* 的初始值分别是 255 和−2147483648。

（4）循环终止条件为 i 等于 0。循环控制变量 *i* 每次循环被逻辑右移 *k* 位。

（5）C 代码中缺失部分填空如下，注意：对无符号整数进行的是逻辑右移。

```
1   int   lproc(int x, int k)
2   {
3         int val =  255  ;
4         int i;
5         for (i=  -2147483648  ; i != 0  ; i=  (unsigned) i >> k  ) {
6               val ^=  (i & x)  ;
7         }
8         return val;
9   }
```

32．参考答案：

从第 5 行指令可知，i 在 EAX 中；从第 6 行指令可知，sptr 在 ECX 中。由第 7 行指令可知，i*28 在 EBX 中。由第 8、9 和 10 行指令可猜出，x 的每个数组元素占 28B，并且 xptr->idx 的地址为 sptr+i*28+4，故在 line_struct 中的第一个分量为 idx，因而后面的 24B 为 6 个数组元素 a[0]~a[5]，类型与 val 变量的类型相同，即 unsigned int。

line_struct 结构类型的定义如下：

```
typedef   struct {
      int          idx;
      unsigned   a[6];
} line_struct;
```

由第 11、12 行指令可知，x 数组所占空间为 0xc8−4=200−4=196B，所以 LEN=196/28=7。

33．参考答案：

（1）n1.ptr、n1.data1、n2.data2、n2.next 的偏移量分别是 0、4、0 和 4。

（2）node 类型总大小占 8 字节。

（3）chain_proc 的 C 代码中缺失的表达式如下：

uptr-> n2.next ->n1.data1 = *(uptr->n2. next->n1. ptr) – uptr->n2. data2 ;

34．参考答案：

（1）函数 trace 的入口参数 tptr 通过 RDI 寄存器传递。

（2）函数 trace 完整的 C 语言代码如下：

```
long trace( tree_ptr tptr)
{
    long    ret_val=0;
    tree_ptr    p=tptr;
    while (p!=0) {
        ret_val=p->val;
        p=p->left;
    }
    return ret_val;
}
```

（3）函数 trace 的功能是：返回二叉树中最左边叶子节点中的值 val。