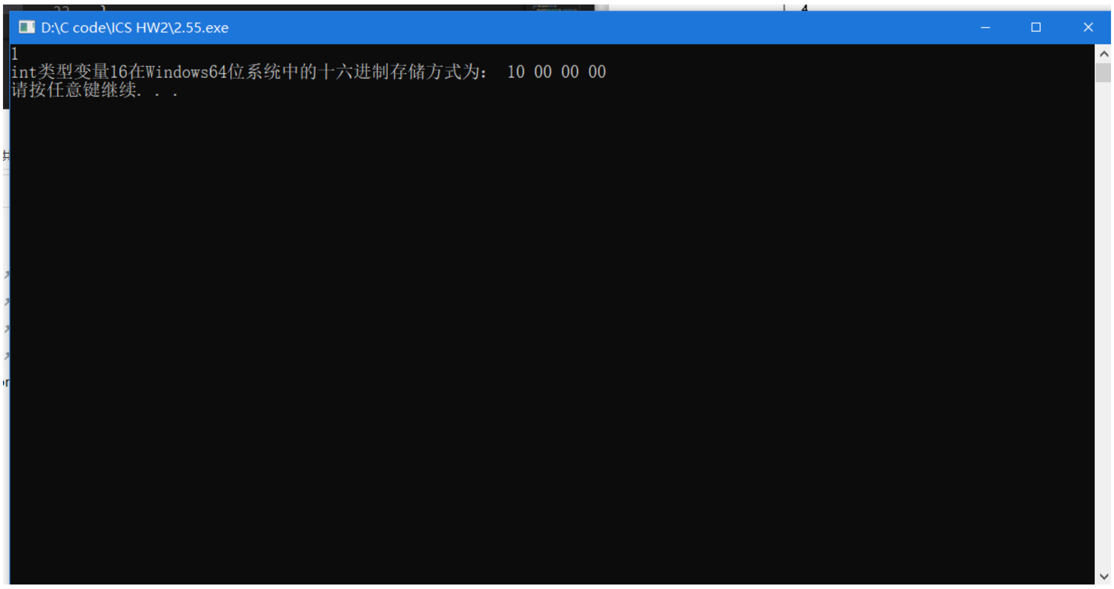
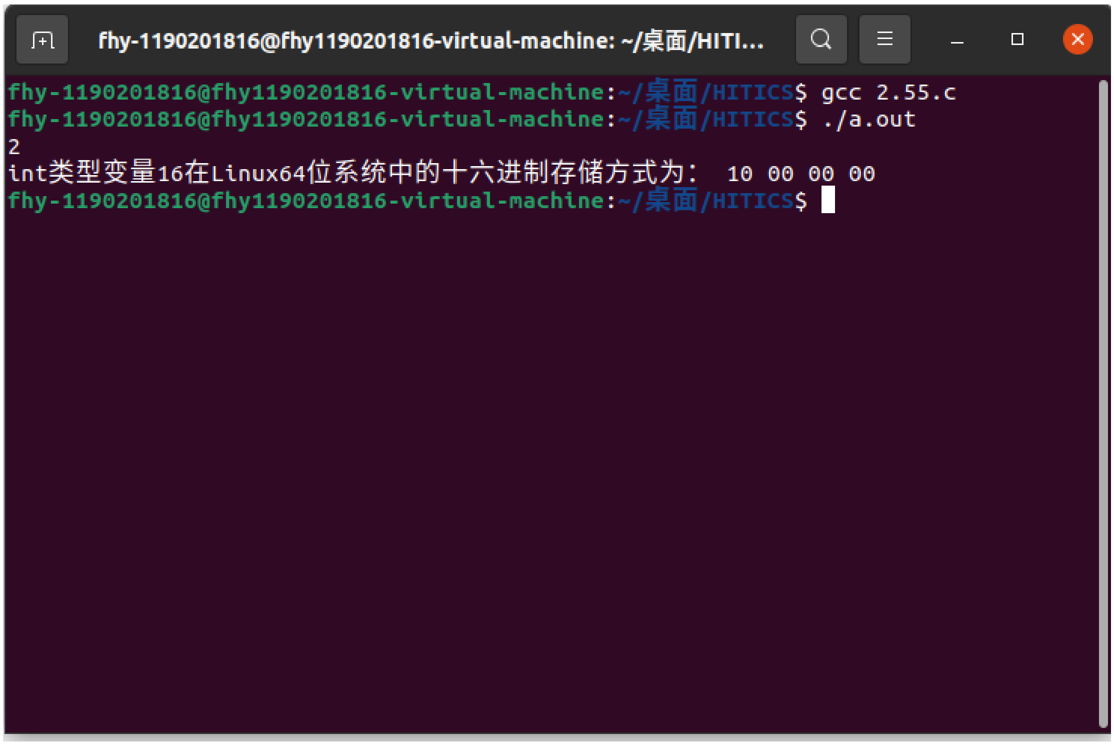
2.55





由上图可知，我是用的 Win 与 Linux 系统下数据都是小端法存储的。

2.59

2.63

```c
unsigned srl (unsigned x, int k)
// Perform shift arithetically
{
    unsigned xsra = (int) x >> k;
    int w = sizeof(int) << 3;
    int h = (int)-1 << (w-k); //得到二进制高k位都为1的二进制数
    xsra = xsra & (~h); //~h位高k位为0，其余位为1的二进制数，逻辑右移k位的高k位都为0
    return xsra; //原二进制数和~h进行按位与得到高k位为0，其余位不变的xsra.
}

int sra(int x, int k)
// Perfprm shift logically
{
    int xsrl = (unsigned) x >> k;
    int w = sizeof(int) << 3;
    int h = (int)-1 << (w-k);//h为高k位都为1的二进制数
    int Is_neg = x & ((int)(-1) << (w-1));
    Is_neg = Is_neg && Is_neg;//若x为负数，则Is_neg为1，否则为0
    h = h & ((!Is_neg)-1);  /*
                            若Is_neg为1，则表达式(!Is_neg)-1为-1，即二进制全为1，h和-1按位与不变
                            若Is_neg为0，则表达式(!Is_neg)-1为0，h和0按位与变为0.
                            */
    xsrl = xsrl | h;        /*
                            若x为负数原二进制数与h按位或得到得到高k位都为1其余位不变的数
                            若x为正数原二进制数与h按位或不变.
                            */
    return xsrl;
}
```

测试样例:

```c
int main()
{
    unsigned testu = 0x80000000;
    int testi = 0x80000000;
    printf("%x\n",srl(testu,31));
    printf("%x\n",sra(testi,31));
    getchar();
    return 0;
}
```

```
d:\C code\ICS HW2\2.63.exe
1
ffffffff
```

2.67

```c
int my_int_size_is32()
{
    int set_msb = 1 << 31; //若1（二进制：1）左移31位set_msb为非零的数，则机器的int至少为32位。
    int set_lsb = 2 << 31; //若2（二进制：01）左移31位set_lsb为零，则机器的int最多为32位。
    return set_msb && (!set_lsb);
}

int my_int_size_is16()
{
    int set_msb = 1 << 15; //若1（二进制：1）左移15位set_msb为非零的数，则机器的int至少为16位。
    int set_lsb = 2 << 15; //若2（二进制：01）左移15位set_lsb为零，则机器的int最多为16位。
    return set_msb && (!set_lsb);
}
```

2.71

答：

A：原操作将通过将 word 右移 bytenum*8 位将需要的字节移动到最低 8 位，&0xFF 使除了最低的 8 位其他位都归为 0。这种操作不是符号位扩展，而是算数右移，因为这样无论符号位是 1 还是 0，都被扩展为 0。

B：

```c
/2 > C 2.71.c > xbyte(packed_t, int)
int xbyte(packed_t word, int bytenum)
{
    int un_byte; //无符号扩展
    int exp_byte; //有符号扩展
    int flag; //标志符号位是否为1
    int h = 0xffffffff;
    un_byte = word >> (bytenum << 3) & 0xff;
    flag = un_byte >> 7;
    exp_byte = un_byte;
    h = h << 8;
    h = h & ((!flag)-1);
    exp_byte = un_byte | h; //如果符号位为1，则将其他位变为1
    return exp_byte;
}
```

我的方法开始与原方法相同，之后的操作与题目 2.63 类似，flag 为标志位，若需操作字节的符号位为 1，则 flag 为 1，否则为 0。若 flag 为 1，则 h 被操作为 0xffffff00，无符号扩展后的数与 h 按位或后即为有符号扩展。若 flag 为 0，则 h 为 0x00000000，无符号扩展与有符号扩展相同，所以不变。

2.75

答：由书中等式 2.18 $\begin{aligned} x' &= x + x_{31}2^{32} \\ y' &= y + y_{31}2^{32} \end{aligned}$ 可得，$x'y' = xy + 2^{32}(x_{31}y + y_{31}x) + x_{31}y_{31}2^{64}$ 。

我们要将 32 位的乘法扩展到 64 位，则最终的结果只精确 2^{63}，即最终的结果会进行对 2^{64} 的取模运算。所以我们可以将最后一项舍去，最终的公式变为：

$$x'y' = xy + 2^{32}(x_{31}y + y_{31}x)$$

由以上等式编程。

```
#include <stdio.h>
#include <inttypes.h>

int signed_high_prod(int x, int y)  //有符号数相乘的高32位
{
    int64_t mul = (int64_t) x * y;
    return mul >> 32;
}

unsigned unsigned_high_prod(unsigned x, unsigned y) //无符号数相乘的高32位
{
    int h_x = x >> 31;
    int h_y = y >> 31;
    int signed_prod = signed_high_prod(x,y);
    return signed_prod + x * h_y + y * h_x;
}
```

2.79

原理： 除以 2 的幂的补码除法，向上舍入

C 变量 x 和 k 分别有补码值 x 和无符号数值 k，且 $0 \leqslant k < w$，则当执行算术移位时，C 表达式 (x+(1<<k)−1)>>k 产生数值 $\lfloor x/2^k \rfloor$。

```
int mul3div4(int x)
{

    int w = sizeof(int) << 3;
    int Is_neg = x >> (w-1);
    Is_neg = Is_neg && Is_neg;//若x为负数，Is_neg为1
    int Is_bias = 2 & ((!Is_neg)-1);//若x为负数，则Is_bias为2，否则为0
    int bias = (1 << Is_bias) -1;//计算偏置量
    x = (x + bias) >> 2;//进行算术右移
    return x;
}
```

根据书中补码除以 2^k 得到 $\lfloor x/2^k \rfloor$ 的原理，设计该算法。

2.83

答：A：设 X 为这个无穷串的值，由 $X \cdot 2^k = X+Y$ 可得。$X = Y/2^k-1$。

B：(a) 5/7

(b) 2/5

(b) 19/63

2.87

描　述	Hex	M	E	V	D
−0	0x8000	0	−14	−0	−0.0
最小的>2 的值	0x4001	$\frac{1025}{1024}$	1	1025×2^{-9}	2.00195312
512	0x6000	1	9	512	512.0
最大的非规格化数	0x03FF	$\frac{1023}{1024}$	−14	1023×2^{-24}	6.09755516e-5
−∞	0xFC00	—	—	−∞	−∞
十六进制表示为 3BB0 的数	3BB0	$\frac{123}{64}$	−1	123×2^{-7}	0.9609375

2.91

答：A：11.0010010000111111011011

B：11.001001(001 循环)···

C：第九位，0x40490FDB 的第九位为 0，22/7 的二进制小数的第九位为 1。

2.95

```
typedef unsigned float_bits;
float_bits float_half(float_bits f)
{
    unsigned s = f >> 31;
    unsigned exp = (f >> 23) &0xff;
    if(exp == 0 || exp == 1) //NaN或者正负无穷大
    {
        return f;
    }
    unsigned frac = f & 0x7fffff;
    unsigned c = (frac & 1) && ((frac >> 1) & 1);//进位
    if (exp == 0)//非规格化数
    {
        frac >> 1;
        frac = frac + c;
    }
    else if(exp == 1)//阶码为1，需要"借位"
    {
        exp = 0;
        frac = (frac >> 1);
        frac = frac + 0x400000;
        frac = frac + c;
    }
    else
    {
        exp = exp -1;
    }
    return (s << 31) | (exp << 23) | (frac);
}
```

分类讨论：

1. 当浮点数为 NaN 或者无穷大时，直接返回其本身。

2. 当浮点数为非规格化数时，要计算是否进位。
   是否进位需要观察 frac 的最后两位 XY，若 Y 是 1，向右移位 1 位后则需要舍入。
   向偶数舍入要使最低位为 0，若 X=1，则 frac+1；若 X=0，则直接舍去 Y。
3. 当浮点数为规格化数时，直接将 exp 减去 1 即可。