datalab

前言

将datalab-handout.tar移动到WSL环境中，并解压，该实验的目的便是在bits.c文件中按照特定的规则，解决一系列puzzles

其中，使用btest检测结果的正确性，使用dlc检测是否符合规则（比如是否使用了禁止使用的operator，是否使用了过多的operator，在解决整数puzzles的时候，是否使用了non-straightline code的代码），有关整数的题目，只需要考虑32位整数，有关浮点数的题目，只需要考虑单精度浮点数

我运行的时候出现了如下的问题：

```
gcc -O -Wall -m32 -lm -o btest bits.c btest.c decl.c tests.c
In file included from btest.c:16:
/usr/include/stdio.h:27:10: fatal error: bits/libc-header-start.h: No such file or directory
   27 | #include <bits/libc-header-start.h>
      |          ^~~~~~~~~~~~~~~~~~~~~~~~~~
compilation terminated.
In file included from decl.c:1:
/usr/include/stdio.h:27:10: fatal error: bits/libc-header-start.h: No such file or directory
   27 | #include <bits/libc-header-start.h>
      |          ^~~~~~~~~~~~~~~~~~~~~~~~~~
compilation terminated.
In file included from /usr/lib/gcc/x86_64-linux-gnu/11/include/limits.h:203,
                 from /usr/lib/gcc/x86_64-linux-gnu/11/include/syslimits.h:7,
                 from /usr/lib/gcc/x86_64-linux-gnu/11/include/limits.h:34,
                 from tests.c:3:
/usr/include/limits.h:26:10: fatal error: bits/libc-header-start.h: No such file or directory
   26 | #include <bits/libc-header-start.h>
      |          ^~~~~~~~~~~~~~~~~~~~~~~~~~
compilation terminated.
make: *** [Makefile:11: btest] Error 1
```

这是由于环境没有完善造成的，通过执行以下命令来完善编译环境：

```
sudo apt-get update
sudo apt-get install gcc-multilib libc6-dev-i386
```

puzzles求解

bitXor

根据德摩根定律可知：

$$
\begin{aligned}
x \wedge y &= (\neg x \ \& \ y) \mid (x \ \& \ \neg y) \\
&= \neg\big(\neg\big((\neg x \ \& \ y) \mid (x \ \& \ \neg y)\big)\big) \\
&= \neg\big(\neg(\neg x \ \& \ y) \ \& \ \neg(x \ \& \ \neg y)\big)
\end{aligned}
$$

按照要求，编写代码：

```c
int bitXor(int x, int y)
{
  int left = ~(~x & y);
  int right = ~(x & ~y);
  int res = ~(left & right);
  return res;
}
```

```
dmz@LAPTOP-LLTKTC0C:~/cmu-15213-lab/datalab-handout$ ./btest -f bitXor
Score    Rating  Errors  Function
 1       1       0       bitXor
Total points: 1/1
```

tmin

根据补码公式:

> **补码编码的定义**
> $B2T_w(\vec{x}) = -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$

我们知道，32位补码能表示的最小数是 `1000···(共31个0)000` ，思路很简单，首先对0进行取反得到 `all_one = 1111···(共32个1)111` 然后让这个数左移31位即可

```c
int tmin(void)

{
  int all_one = ~0;
  int res = all_one << 31;
  return res;
}
```

```
dmz@LAPTOP-LLTKTC0C:~/cmu-15213-lab/datalab-handout$ ./btest -f tmin
Score    Rating  Errors  Function
 1       1       0       tmin
Total points: 1/1
```

isTmax

我们知道，32位补码能表示的最大数是 `01111···(共31个1)111` ，然后我们发现：Tmax的补码的位表示与Tmax+1相同，也就是：

$$Tmax + 1 == \neg Tmax$$

在位运算中如何实现当a和b相同时返回一呢？如下所示

```
return !(a ^ b);
```

接下来思考有没有边界情况？发现当x是`111···(共32个1)111`时，同样满足上述性质，因此我们需要把这个情况排除，综上，我们编写如下代码：

```
int isTmax(int x)

{
  int x_o = ~x;
  int x_and_one = x + 1;
  int is_not_all_one = !!(~x);
  return !(x_o ^ x_and_one) & is_not_all_one;
}
```

```
dmz@LAPTOP-LLTKTC0C:~/cmu-15213-lab/datalab-handout$ ./btest -f isTmax
Score   Rating  Errors  Function
 1       1       0       isTmax
Total points: 1/1
```

allOddBits

很容易想到的一个思路是让x与0xAAAAAAAA做与运算，如果结果等于0xAAAAAAAA就返回一，因此这个问题就转换为了如何构造0xAAAAAAAA，由于datalab最大能够使用的整数常量为0xff，因此我们用0xAA去构造0xAAAAAAAA，综上，我们编写如下代码：

```
int allOddBits(int x)
{
  int two_a = 0xAA;
  int eight_a = (two_a << 24) + (two_a << 16) + (two_a << 8) + two_a; //
0xAAAAAAAA
  return !(eight_a ^ (eight_a & x));
}
```

```
dmz@LAPTOP-LLTKTC0C:~/cmu-15213-lab/datalab-handout$ ./btest -f allOddBits
Score   Rating  Errors  Function
 2       2       0       allOddBits
Total points: 2/2
```

negate

刚开始我以为让这个题变一个符号位就行，于是是这样写的：

```
int negate(int x) {
    int tmin = 1 << 31;
```

```
    return x ^ tmin;
}
```

但才想起来tmin和0这两个边界没办法处理，想了半天不知道这个边界如何处理？又突然想起来，其实求负数直接用补码定义就行了......取反加一

```
int negate(int x) {
    return ~x + 1;
}
```

```
(base) dmz@LAPTOP-LLTKTC0C:~/cmu-15213-lab/datalab-handout$ ./btest -f negate
Score   Rating  Errors  Function
 2       2       0       negate
Total points: 2/2
```

isAsciiDigit

这道题是让我们判断x是否在一个区间范围`[0x30, 0x39]`之内，因此我们可以让x与0x30作差，让0x39与x作差，然后当两个差的符号均为正，则结果返回1

然而，这道题是不允许使用减号的，所以可以改成加上对应的负数，如何求负数呢？我们只需要按照上道题的取反加一的方法即可！

```
int isAsciiDigit(int x)
{
  int low = 0x30;
  int high = 0x39;
  int diff1 = x + (~low + 1);
  int diff2 = high + (~x + 1);
  int sign1 = diff1 >> 31 & 1;
  int sign2 = diff2 >> 31 & 1;
  return !(sign1 | sign2);
}
```

```
(base) dmz@LAPTOP-LLTKTC0C:~/cmu-15213-lab/datalab-handout$ ./btest bits.c -f isAsciiDigit
Score   Rating  Errors  Function
 3       3       0       isAsciiDigit
Total points: 3/3
```

conditional

使用位运算模拟条件语句，我们可以使用**掩码mask**
例如，对于`if (condition) return a; else return b;`
如果条件为true，我们就令mask为全1的值，如果条件为false，令其为全0的值
这样的话，最终就返回：`return (mask & a) | (~mask & b)`
因为当条件为true时，全1的mask与a进行与运算还是a，而~mask为0，与b进行与运算结果为0，因此最终返回a；同理，条件为false时，最终返回false

但是，我们在这个题中如何构造mask呢？可以使用： `~(!!x)+1` ，当x不等于0时，也就是条件为true时， `!!x` 为1，取反加一就是-1，位表示刚好是全1，当x等于0时，也就是条件为false时， `!!x` 为0，取反加一还是0，位表示刚好是全0。综上：

```
int conditional(int x, int y, int z)
{
  int mask = ~(!!x) + 1; // 如果x不为0，就转换为全1，否则转换为全0
  int true_val = y & mask;
  int false_val = z & ~mask;
  return true_val | false_val; // true_val和false_val必有一个是0
}
```

```
(base) dmz@LAPTOP-LLTKTC0C:~/cmu-15213-lab/datalab-handout$ ./btest bits.c -f conditional
Score   Rating  Errors  Function
 3       3       0       conditional
Total points: 3/3
```

isLessOrEqual

这道题目可以直接想到的一种思路是让y减去x(y加上x的取反加一)，如果差值的符号位是0，则返回1，否则返回0，但是直接这样写会出现溢出的情况，什么时候发生溢出？就是当x和y符号位不相同时，因此我们需要分类讨论两种情况，即符号位相同和不同，两种情况的条件转移采用上一题 `conditional` 的思路

我们设xy符号位异号为真，同号为假，那么条件变成了： `sign_x ^ sign_y`
当异号时，我们直接返回x的符号位即可，因此x符号位为1时，x为负，y为正，最后必然返回1，反之同理
当同号时，直接利用最开始说的方法就可以了

综上，编写如下代码

```
int isLessOrEqual(int x, int y)
{
  // 假设符号位异号为真，同号为假
  int sign_x = x >> 31 & 1;
  int sign_y = y >> 31 & 1;
  int condition = sign_x ^ sign_y;
  int mask = ~(!!condition) + 1;
  int true_val = sign_x & mask;
  int diff = y + (~x + 1);
  int sign_diff = diff >> 31 & 1;
  int false_val = !sign_diff & (~mask);
  return true_val | false_val;
}
```

```
(base) dmz@LAPTOP-LLTKTC0C:~/cmu-15213-lab/datalab-handout$ ./btest -f isLessOrEqual
Score    Rating    Errors    Function
 3        3         0        isLessOrEqual
Total points: 3/3
```

logicalNeg

本题可以利用”0的负数还是0“的性质，也就是说0进行取反加一之后，其最高符号位依然为0，而其它任意的非0数x的最高符号位，和-x的最高符号位必定是异号，利用此性质：

```
int logicalNeg(int x)
{
    int sign_or = x | (~x + 1);
    return (sign_or >> 31) + 1;
}
```

```
(base) dmz@LAPTOP-LLTKTC0C:~/cmu-15213-lab/datalab-handout$ ./btest bits.c -f logicalNeg
Score    Rating    Errors    Function
 4        4         0        logicalNeg
Total points: 4/4
```

howManyBits

本puzzle的目的是求一个数如果用补码表示，最少需要多少位，我们分别举一个正数和负数的例子：
对于12，二进制是1100，表示的话最少需要4+1=5位（还有一个是符号位），也就是说01100和001100表示的都是12
对于-5，二进制是1011，表示的话最少需要4位，（最高位1为符号位），1011和11011表示的都是-5

根据上面这个例子，我们找到了一个规律，对于正数，我们需要找到最高位的1的位置，然后再加上一个符号位，就是最少需要的位数，比如1100，最高位1的位置是在从低到高第4个位置，然后再加一就是5；对于负数，我们需要找到最高位的0的位置，然后再加上一个符号位，比如1011，最高位0的位置在从低到高第3个位置，然后再加上就是4

为了统一，我们使用以下代码先对x处理，当x为正数时，保持x不变，当x为负数时，将x取反，这样做的好处是，不管是正数还是负数，我们都去找最高位1的位置即可：

```
int sign = x >> 31;
x = (sign & ~x) | (~sign & x); // 如果x为正数的话，不变，如果x为负数的话，x取反
```

接下来就可以按位依次寻找最高位的1，一种直接的想法是依次判断当前的数是不是全0，如果不是就让sum加一反之sum不变，并让x右移，共循环31次，如下所示：

```
int howManyBits(int x)
```

```
{
  int sign = x >> 31;
  x = (sign & ~x) | (~sign & x); // 如果x为正数的话，不变，如果x为负数的话，x取反
  int sum = 0;
  sum = sum + !!x;
  x = x >> 1;
  sum = sum + !!x;
  x = x >> 1;
  //......共重复31次
  sum = sum + !!x;
  x = x >> 1;
  return sum + 1;//加上一个符号位
}
```

这样做答案是没问题的，可以通过btest的正确性测试，但是不能通过dlc的测试，因为operator共使用了131个，超过了上限90个

因此，在这里参考网上的做法，使用二分法，依次缩小最高位1的所在范围，从而减少operator的数量，如下所示：

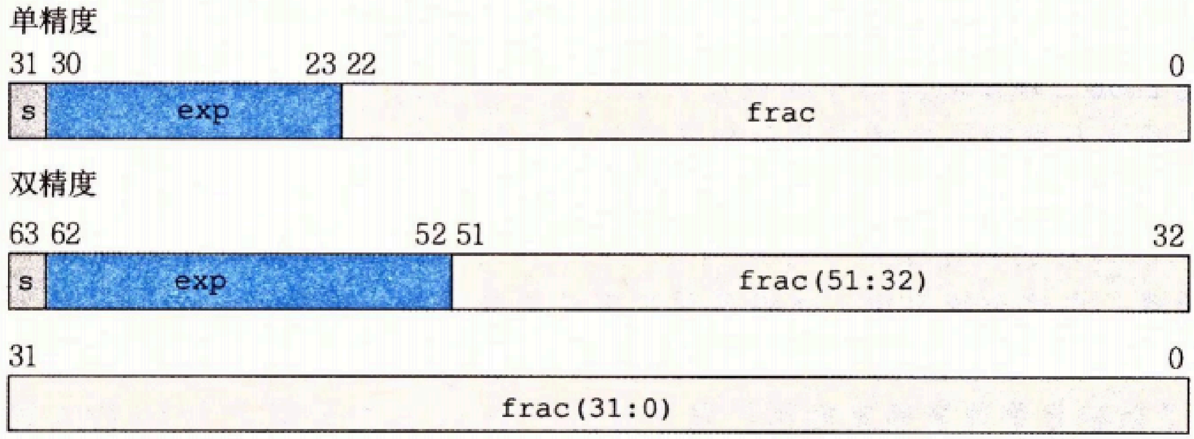
```
int howManyBits(int x)
{
  int sign = x >> 31;
  int b16, b8, b4, b2, b1, b0;
  x = (sign & ~x) | (~sign & x); // 如果x为正数的话，不变，如果x为负数的话，x取反
  // 判断最高16位是否有1，如果有，则让x右移，以判断最高16位具体最高1在哪里
  // 如果没有，x不右移，以判断8-16位中是否有1，以此类推.....
  b16 = !!(x >> 16) << 4;
  x = x >> b16;
  b8 = !!(x >> 8) << 3;
  x = x >> b8;
  b4 = !!(x >> 4) << 2;
  x = x >> b4;
  b2 = !!(x >> 2) << 1;
  x = x >> b2;
  b1 = !!(x >> 1);
  x >> b1;
  b0 = !!x;
  return b16 + b8 + b4 + b2 + b1 + b0 + 1;
}
```

这种情况下，共使用了38个operators

```
^[[Admz@LAPTOP-LLTKTC0C:~/cmu-15213-lab/datalab-handout$ ./btest bits.c -f howManyBits
Score    Rating  Errors  Function
 4       4       0       howManyBits
Total points: 4/4
```

floatScale2

本puzzle的目的是给定一个无符号数uf，把它看成是一个浮点数的位模式，计算其乘以二后的结果并返回，我们先复习一下IEEE浮点数的表示：

> 
>
>
> 图 2-32　标准浮点格式(浮点数由 3 个字段表示。两种最常见的格式是它们被封装到 32 位(单精度)和 64 位(双精度)的字中)

我们首先根据单精度IEEE浮点数的格式分别拆分出uf的sign、exp和frac：

```
int sign = uf & 0x80000000;
int exp = uf & 0x7f800000;
int frac = uf & 0x007fffff;
```

然后我们分三种情况进行讨论：

情况一：无穷大或者NaN
当exp全1时，uf为无穷大或者NaN，此时直接返回uf即可：

```
if (exp == 0x7f800000) return uf;
```

情况二：非规格化数
当exp全0，uf为非规格化数，此时如果frac全0，那么uf为0，乘以2还是0，因此直接返回uf；如果frac不为0，只需要将frac左移1即可，这里，我们举两个例子说明：
假设exp的位数是4，frac的位数是3：
例子一：0 0000 001，E是1/64，M是1/8，最终的值是：1/512
将这个非规格化数的尾数部分左移1位，得到0 0000 010，E不变，M变为2/8，最终变为2/512

例子二：0 0000 111，E是1/64，M是7/8，最终的值是：7/512
将这个非规格化数的尾数部分左移1位，得到0 0001 110，变为了规格化数，E不变，M变为1+6/8=14/8，最终的值是：14/512
因此，这部分代码为：

```
if (exp == 0) {
    if (frac == 0) return uf;
```

```
    frac = frac << 1;
}
```

情况三：规格化数
这种情况下，我们直接让阶码exp加一即可，但是如果加一之后，exp变成全1的数，说明uf表示的浮点数乘以2之后溢出为正无穷了，所以，我们需要将frac置零

```
exp += 0x00800000;
if (exp == 0x7f800000) frac = 0;
```

最后，我们将三种情况的结果进行合并：`return sign | exp | frac`

综上：

```
unsigned floatScale2(unsigned uf)
{
  int sign = uf & 0x80000000;
  int exp = uf & 0x7f800000;
  int frac = uf & 0x007fffff;
  if (exp == 0x7f800000) // 无穷大或NaN
    return uf;
  if (exp == 0) // 非规格化数
  {
    if (frac == 0)
      return uf;
    frac = frac << 1;
  }
  else // 规格化数
  {
    exp += 0x00800000;
    if (exp == 0x7f800000)
      frac = 0;
  }
  return sign | exp | frac;
}
```

```
dmz@LAPTOP-LLTKTC0C:~/cmu-15213-lab/datalab-handout$ ./btest bits.c -f floatScale2
Score   Rating  Errors  Function
 4      4       0       floatScale2
Total points: 4/4
```

floatFloat2Int

本puzzle的目的是给定一个无符号数uf，将其转换成整数，如果uf表示的数值超过了int的范围就返回0x80000000，我们首先回顾一下如何根据IEEE754的标准进行位模式和其对应的数据值的转换，如下图所示：

> 下面展示了8位浮点格式的示例，其中k=4，n=3：
>
> | 描述 | 位表示 | 指数 | | | 小数 | | 值 | | |
> |---|---|---|---|---|---|---|---|---|---|
> | | | e | E | 2^E | f | M | $2^E \times M$ | V | 十进制 |
> | 0 | 0 0000 000 | 0 | −6 | $\frac{1}{64}$ | $\frac{0}{8}$ | $\frac{0}{8}$ | $\frac{0}{512}$ | 0 | 0.0 |
> | 最小的非规格化数 | 0 0000 001 | 0 | −6 | $\frac{1}{64}$ | $\frac{1}{8}$ | $\frac{1}{8}$ | $\frac{1}{512}$ | $\frac{1}{512}$ | 0.001953 |
> | | 0 0000 010 | 0 | −6 | $\frac{1}{64}$ | $\frac{2}{8}$ | $\frac{2}{8}$ | $\frac{2}{512}$ | $\frac{1}{256}$ | 0.003906 |
> | | 0 0000 011 | 0 | −6 | $\frac{1}{64}$ | $\frac{3}{8}$ | $\frac{3}{8}$ | $\frac{3}{512}$ | $\frac{3}{512}$ | 0.005859 |
> | | ⋮ | | | | | | | | |
> | 最大的非规格化数 | 0 0000 111 | 0 | −6 | $\frac{1}{64}$ | $\frac{7}{8}$ | $\frac{7}{8}$ | $\frac{7}{512}$ | $\frac{7}{512}$ | 0.013672 |
> | 最小的规格化数 | 0 0001 000 | 1 | −6 | $\frac{1}{64}$ | $\frac{0}{8}$ | $\frac{8}{8}$ | $\frac{8}{512}$ | $\frac{1}{64}$ | 0.015625 |
> | | 0 0001 001 | 1 | −6 | $\frac{1}{64}$ | $\frac{1}{8}$ | $\frac{9}{8}$ | $\frac{9}{512}$ | $\frac{9}{512}$ | 0.017578 |
> | | ⋮ | | | | | | | | |
> | | 0 0110 110 | 6 | −1 | $\frac{1}{2}$ | $\frac{6}{8}$ | $\frac{14}{8}$ | $\frac{14}{16}$ | $\frac{7}{8}$ | 0.875 |
> | | 0 0110 111 | 6 | −1 | $\frac{1}{2}$ | $\frac{7}{8}$ | $\frac{15}{8}$ | $\frac{15}{16}$ | $\frac{15}{16}$ | 0.9375 |
> | 1 | 0 0111 000 | 7 | 0 | 1 | $\frac{0}{8}$ | $\frac{8}{8}$ | $\frac{8}{8}$ | 1 | 1.0 |
> | | 0 0111 001 | 7 | 0 | 1 | $\frac{1}{8}$ | $\frac{9}{8}$ | $\frac{9}{8}$ | $\frac{9}{8}$ | 1.125 |
> | | 0 0111 010 | 7 | 0 | 1 | $\frac{2}{8}$ | $\frac{10}{8}$ | $\frac{10}{8}$ | $\frac{5}{4}$ | 1.25 |
> | | ⋮ | | | | | | | | |
> | | 0 1110 110 | 14 | 7 | 128 | $\frac{6}{8}$ | $\frac{14}{8}$ | $\frac{1792}{8}$ | 224 | 224.0 |
> | 最大的规格化数 | 0 1110 111 | 14 | 7 | 128 | $\frac{7}{8}$ | $\frac{15}{8}$ | $\frac{1920}{8}$ | 240 | 240.0 |
> | 无穷大 | 0 1111 000 | — | — | — | — | — | — | ∞ | — |
>
> 图 2-35　8 位浮点格式的非负值示例($k=4$ 的阶码位的和 $n=3$ 的小数位。偏置量是 7)

对于uf，我们依次获得sign、E和M，其中exp需要减去偏置Bias=127得到最终的幂E，而frac需要补上最前面隐含的1，以进行规格化数处理得到最终的M

```
int sign = uf >> 31 & 1;
int E = ((uf >> 23) & 0xff) - 127;
int M = (uf & 0x007fffff) | 0x00800000;
```

我们发现当E<0时，最后的十进制数都是小于1的，对于这部分的数，我们直接返回0即可：

```
if (E < 0) return 0;
```

接下来，我们举一个例子来探究如何通过E和M推出最终的结果，我们以十进制数3为例， 它的二进制表示是11，可以表示为：$1.1 * 2^{1}$，由此frac为 `1000···（共22个0）000` ，M为 `1100···(共22个0）000` ，E为 `1` ，如何从M变为我们想要的“11”？方法是将M右移22位，这个22位是怎么计算的，其实就是23-E；当E>23时怎么办呢？比如$1.1 * 2^{24}$，它其实表示的是整数 `11000···（共23个0）···` ，是M左移1位得到的，这个1恰好是通过E-23算出的；因此

```
if (E > 23) res = M << (E - 23);
else res = M >> (23 - E);
```

上面说到E是下边界的，当小于0的时候，最终得到的十进制数一定小于1，那么E大于多少的时候得到的十进制数会爆int呢？还以上述这个例子，$1.1 * 2^E$表示的十进制数是：`11000···（E-1个0）00`总共有E+1位数，int是32位的，去掉符号位还剩31位，因此需要满足：

$$E + 1 <= 31$$

因此当E大于30时，会爆int，此时返回题目中要求的`0x80000000u`即可

最后，我们判断符号位，如果符号位为1，我们将res取反加一，也就是变成负数

综上：

```
int floatFloat2Int(unsigned uf)
{
  int sign = uf >> 31 & 1;
  int E = ((uf >> 23) & 0xff) - 127;
  int M = (uf & 0x007fffff) | 0x00800000;
  int res;
  if (E < 0)
    return 0;
  if (E > 30)
    return 0x80000000u;
  if (E > 23)
    res = M << (E - 23);
  else
    res = M >> (23 - E);
  if (sign)
    return ~res + 1;
  else
    return res;
}
```

```
dmz@LAPTOP-LLTKTC0C:~/cmu-15213-lab/datalab-handout$ ./btest bits.c -f floatFloat2Int
Score    Rating   Errors   Function
 4        4        0        floatFloat2Int
Total points: 4/4
```

floatPower2

本puzzle的目的是给定一个整数x，让我们求解2^x并以无符号数形式返回IEEE754的单精度浮点数格式，当x过大的时候，返回正无穷的表示，当x过小的时候，返回0，我们分类讨论：

情况一：返回正无穷

在规格数中，exp最大为：`11111110`，也就是254，偏置Bias=127，因此E最大为254-127=127，所以，当x>127时，返回正无穷0x7f800000

情况二：返回0

在非规格数中，能够表示的最小数为：E=1-Bias=-126，而M最小为：`000···（共26个0）1`整体表示的最小数为：2^{-149}，因此当x<-149时，返回0

情况三：非规格数

当x大于等于-149且x小于等于-127时，需要使用非规格数进行表示

对于最小的情况，也就是2^{-149}，sign和exp全为0，frac只有最低位为1，因此位模式是1<<0

对于最大的情况，也就是2^{-127}，sign和exp也为0，frac只有最高位为1，因此位模式是1<<23

对于一般情况，也就是2^{x}，位模式为：1<<(x+149)

情况四：规格数

规格数的表示为$1.0 * 2^{E}$

其中E=x=exp-Bias=exp-127，推出exp=x+127

最终的位模式是让exp左移23位，即exp<<23

综上：

```c
unsigned floatPower2(int x)
{
  if (x > 127)
    return 0x7f800000;
  else if (x < -126)
    return 0;
  else if (x <= -127)
    return 1 << (x + 149);
  else
    return (x + 127) << 23;
}
```

```
dmz@LAPTOP-LLTKTC0C:~/cmu-15213-lab/datalab-handout$ ./btest bits.c -f floatPower2
Score   Rating  Errors  Function
 4       4       0       floatPower2
Total points: 4/4
```

总结与思考

```
dmz@LAPTOP-LLTKTC0C:~/cmu-15213-lab/datalab-handout$ ./btest
Score   Rating  Errors  Function
 1      1       0       bitXor
 1      1       0       tmin
 1      1       0       isTmax
 2      2       0       allOddBits
 2      2       0       negate
 3      3       0       isAsciiDigit
 3      3       0       conditional
 3      3       0       isLessOrEqual
 4      4       0       logicalNeg
 4      4       0       howManyBits
 4      4       0       floatScale2
 4      4       0       floatFloat2Int
 4      4       0       floatPower2
Total points: 36/36
```

到此完成了datalab的全部puzzles，执行 `./dlc -e bits.c`，各puzzles使用的operator数量结果如下，每个题解并不是最优的，应存在更好的方法使用更少的operators，本题解方法仅供参考。

```
dmz@LAPTOP-LLTKTC0C:~/cmu-15213-lab/datalab-handout$ ./dlc -e bits.c
dlc:bits.c:150:bitXor: 8 operators
dlc:bits.c:162:tmin: 2 operators
dlc:bits.c:177:isTmax: 8 operators
dlc:bits.c:191:allOddBits: 9 operators
dlc:bits.c:202:negate: 2 operators
dlc:bits.c:222:isAsciiDigit: 12 operators
dlc:bits.c:236:conditional: 8 operators
dlc:bits.c:257:isLessOrEqual: 19 operators
dlc:bits.c:271:logicalNeg: 5 operators
dlc:bits.c:303:howManyBits: 38 operators
dlc:bits.c:336:floatScale2: 11 operators
dlc:bits.c:367:floatFloat2Int: 16 operators
dlc:bits.c:391:floatPower2: 9 operators
```

本次实验主要收获了有关位运算的知识以及技巧，体验到了位运算的强大性，利用位运算效率高的特点，也可以对平常写的一些代码进行优化。最后总结一下本次实验的主要收获：

- 德摩根定律进行与运算和或运算的相互转换
- 利用位运算模拟条件语句，可以使用全1（条件为真）或全0的掩码（条件为假）
- 利用位运算比较大小，分同号和异号两种情况，同号下可以直接相减再比较符号位
- 进行整数位运算时，边界情况一般是Tmin、0
- 进行浮点数位运算时，边界情况一般是向上溢出到正无穷以及向下数值太小无法表示
- 对浮点数的IEEE754的格式更加熟悉