

datalab 报告

姓名：李纪仪

学号：2022201543

总分	bitXor	samesign	logtwo	byteSwap	reverse	logicalShift	leftBitCount	float_i2f	floatScale2	float64_f2i	floatPower2
34	1	2	4	2	3	3	4	4	4	3	4

test 截图：

▼

✔

datalab test

1

▶ Run panjd123/autograding-command-grader@v1

8

rm -f *.o btest fshow ishow *~

9

gcc -O -Wall -m32 -lm -o btest bits.c btest.c decl.c tests.c

10

gcc -O -Wall -m32 -o fshow fshow.c

11

gcc -O -Wall -m32 -o ishow ishow.c

12

Pass, great!!! name: bitXor, point: 1

13

Pass, great!!! name: sameSign, point: 2

14

Pass, great!!! name: logTwo, point: 4

15

Pass, great!!! name: byteSwap, point: 2

16

Pass, great!!! name: reverse, point: 3

17

Pass, great!!! name: logicalShift, point: 3

18

Pass, great!!! name: leftBitCount, point: 4

19

Pass, great!!! name: float_i2f, point: 4

20

Pass, great!!! name: floatScale2, point: 4

21

Pass, great!!! name: float64_f2i, point: 3

22

Pass, great!!! name: floatPower2, point: 4

23

Total Points:34

解题报告

亮点

samesign

- 注意到0既不是正数、也不是负数
- 调整条件判断顺序，减少条件判断和运算符使用

```
// 当x、y都为0时，输出1
if (!x && !y)
    return 1;
// 当x、y仅有一个为0时，输出0
if (!(x && y))
    return 0;
```

logtwo

本质是在寻找最高位的1（不是符号位）的位置

使用二分的思想实现：

先判断是否大于16位，

大于，就检查更高8位

否则，检查更低8位

下一轮同样判断是否大于8位

大于，就检查更高4位

否则，检查更低4位

以此类推

如此，就不用进行31次的判断来寻找最高位的1，每次只判断是否大于这一轮的中间值，用递归的思想处理前半或者后半

```
int result = 0;           // 检查高位并累加到结果中
result = (v > 0xFFFF) << 4; // 如果 v 大于 16 位
v >>= result;             // 相应右移

int tmp = (v > 0xFF) << 3;
result |= tmp;             // 如果 v 大于 8 位
v >>= tmp;                 // 相应右移

tmp = (v > 0xF) << 2;
result |= tmp;             // 如果 v 大于 4 位
v >>= tmp;                 // 相应右移

tmp = (v > 0x3) << 1;
result |= tmp;             // 如果 v 大于 2 位
v >>= tmp;                 // 相应右移

return result | (v >> 1); // 最后检查最高位，加入到结果中
```

reverse

对于32位数，先分成16位一组交换，这样只用递归的交换这两组16位

对于16位数，同样可以分成8位一组交换，再递归处理每组的二进制码

以此类推

```
unsigned reverse(unsigned v) {
    v = (v >> 16) | (v << 16);           // 每16位一组交换
    v = ((v >> 8) & 0x00FF00FF) | ((v & 0x00FF00FF) << 8); // 每8位一组交换
    v = ((v >> 4) & 0x0F0F0F0F) | ((v & 0x0F0F0F0F) << 4); // 每4位一组交换
    v = ((v >> 2) & 0x33333333) | ((v & 0x33333333) << 2); // 每2位一组交换
    v = ((v >> 1) & 0x55555555) | ((v & 0x55555555) << 1); // 奇偶位交换
    return v;
}
```

思路来源CSAPP-实验1 Datalab 学习记录 - 简书(jianshu.com), 如下图:

CSAPP-实验1 Datalab 学习记录

15.bitReverse

```
1  /*
2   * bitReverse - Reverse bits in a 32-bit word
3   *   Examples: bitReverse(0x80000002) = 0x40000001
4   *               bitReverse(0x89ABCDEF) = 0xF7D3D591
5   *   Legal ops: ! ~ & ^ | + << >>
6   *   Max ops: 45
7   *   Rating: 4
8   */
9  int bitReverse(int x) {
10     /*
11      *   Warning: 50 operators exceeds max of 45
12      *
13      *   int mask0 = 0xFF | 0xFF << 8;           //0x0000FFFF
14      *   int mask1 = 0xFF | 0xFF << 16;          //0x00FF00FF
15      *   int mask2 = 0x0F | 0x0F << 8 | 0x0F << 16 | 0x0F << 24; //0x0F0F0F0F
16      *   int mask3 = 0x33 | 0x33 << 8 | 0x33 << 16 | 0x33 << 24; //0x33333333
17      *   int mask4 = 0x55 | 0x55 << 8 | 0x55 << 16 | 0x55 << 24; //0x55555555
18      */
19
20     int mask0 = (0xFF) | (0xFF << 8);
21     int mask1 = (0xFF) | (0xFF << 16);
22     int a = (0x0F) | (0x0F << 8);
23     int mask2 = a | (a << 16);
24     int b = (0x33) | (0x33 << 8);
25     int mask3 = b | (b << 16);
26     int c = (0x55) | (0x55 << 8);
27     int mask4 = c | (c << 16);
28
29     int n = (x >> 16 & mask0) | (x << 16);
30     n = (n >> 8 & mask1) | (n << 8 & ~mask1);
31     n = (n >> 4 & mask2) | (n << 4 & ~mask2);
32     n = (n >> 2 & mask3) | (n << 2 & ~mask3);
33     n = (n >> 1 & mask4) | (n << 1 & ~mask4);
34     return n;
35 }
```

做出的优化, 直接使用mask的16进制形式, 减少运算符使用

logicalShift

注意到 `1 << 31 >> n << 1` 与 `0x80000000 >> n << 1` 运算结果不同

leftBitCount

与 `reverse` 思路类似, 不是遍历所有的二进制码得到左边连续的1, 而是先判度高16位是否全是1,

是, 则判断更低8位 (9~16位) 是否全是1

否, 则判断更高8位 (25~32位) 是否全是1

对于这8位, 再判断左侧4位是否全是1

以此类推

因为最后一步 `cnt += (!!(~(x >> (cnt + 1))))`; , 至少会右移一次, 所以 `0xffffffff` 和 `0xffffffe` 需要分开判断 (两者的 `cnt` 返回的都是 1) 故需要 `int off = 1 & (!(~x));` 来判断 `0xffffffff`

思路来源于《深入理解计算机系统/CSAPP》Data Lab - 知乎 (zhihu.com), (实在是无法再优化 🤖) 如下图:



The screenshot shows a Zhihu article page. At the top, the browser address bar shows the URL `https://zhuanlan.zhihu.com/p/57770700`. The article title is "谜题58 - leftBitCount". Below the title, there is a list of bullet points: "返回二进制数从高位到低位, 连续为 1 的个数", "示例: `leftBitCount(0xFFF0F0F0) = 12`", "限制操作: `! ~ & ^ | + << >>`", "操作数量: 50", and "难度: 4". The main text explains the problem: given a binary number `x >> n`, where `n` is the number of leading zeros, find the number of leading ones. The solution involves finding the minimum number of leading ones, which is `31 - n = 31 + ~n + 1`. The code for `leftBitCount` is provided in a code block.

谜题58 - leftBitCount

- 返回二进制数从高位到低位, 连续为 1 的个数
- 示例: `leftBitCount(0xFFF0F0F0) = 12`
- 限制操作: `! ~ & ^ | + << >>`
- 操作数量: 50
- 难度: 4

若二进制数 `x >> n`, 为 `0xffffffff`, 那么可以说明其 `[n, 31]` 位上的数为全 1, 只要找出这个最小的数 `n`, 本题答案即为 `31 - n = 31 + ~n + 1`, 运用类似于题目 41 的技巧来找出这位, 若 `n = 0` 时有两种情况, `0xffffffff` 和 `0xffffffe`, 需要区别。

```
int leftBitCount(int x) {
    int cnt = 0;
    int off = 1 & (!(~x));
    cnt += (!!(~(x >> 16))) << 4;
    cnt += (!!(~(x >> (cnt + 8)))) << 3;
    cnt += (!!(~(x >> (cnt + 4)))) << 2;
    cnt += (!!(~(x >> (cnt + 2)))) << 1;
    cnt += (!!(~(x >> (cnt + 1))));
    return 32 + ~cnt + off;
}
```

floatScale2

对于非规格化数处理

- `frac = f & 7FFFFFFF;`
- if (`frac >> 22 == 0`), `f*2` 还是非规格化数, `frac` 直接左移一位
- if (`frac >> 22 == 1`), `f*2` 是规格化数, `frac` 左移一位后, `frac` 最高位的 1 变为 `exp` 的一位, `exp` 由 0 变 1 (结果乘 2), 规格化数 `frac` 最高位隐藏

所以处理非规格化数只需返回 `(uf << 1) | sign`

反馈/收获/感悟/总结

参考的重要资料

[CSAPP-实验1 Datalab 学习记录 - 简书 \(jianshu.com\)](https://jianshu.com/p/57770700)

[《深入理解计算机系统/CSAPP》Data Lab - 知乎 \(zhihu.com\)](https://zhihu.com/p/57770700)

[ICS Datalab - 知乎 \(zhihu.com\)](https://zhihu.com/p/57770700)

附录（完整代码）

1. bitXor

```
int bitXor(int x, int y) {  
    return ~(x & y) & ~(~x & ~y);  
}
```

思路：

- 观察真值表：

x	y	Xor
0	0	0
0	1	1
1	0	1
1	1	0

- 异或可以表达为：

$$\begin{aligned}x \wedge y &= \sim ((x \& y) | (\sim x \& \sim y)) \\ &= \sim (x \& y) \& \sim (\sim x \& \sim y)\end{aligned}$$

2. sameSign

```
int sameSign(int x, int y) {  
    if (!x && !y) // 当x、y都为0时，输出1  
        return 1;  
    if (!(x && y)) // 当x、y仅有一个为0时，输出0  
        return 0;  
    return !((x ^ y) >> 31); // 当x、y都不为0时，异或后，最高位为1则符号不同，最高位为0则符号相同  
}
```

思路：

- c语言整数大小为4byte，符号位为最高位
- 使用异或，通过最高位判断符号是否相同： `!((x^y)>>31)`
- 考虑异常情况：
 - 0既不是正数也不是负数
 - 当x, y都为0时，输出1
 - 当x, y仅有一个为0时，输出0

3. logtwo

```
int logtwo(int v) {
    int result = 0;           // 检查高位并累加到结果中
    result = (v > 0xFFFF) << 4; // 如果 v 大于 16 位
    v >>= result;             // 相应右移

    int tmp = (v > 0xFF) << 3;
    result |= tmp;            // 如果 v 大于 8 位
    v >>= tmp;                // 相应右移

    tmp = (v > 0xF) << 2;
    result |= tmp;           // 如果 v 大于 4 位
    v >>= tmp;               // 相应右移

    tmp = (v > 0x3) << 1;
    result |= tmp;           // 如果 v 大于 2 位
    v >>= tmp;               // 相应右移

    return result | (v >> 1); // 最后检查最高位，加入到结果中
}
```

思路：

本质是在寻找最高位的1（不是符号位）的位置

使用二分的思想实现：

先判断是否大于16位，

 大于，就检查更高8位

 否则，检查更低8位

下一轮同样判断是否大于8位

 大于，就检查更高4位

 否则，检查更低4位

以此类推

如此，就不用进行31次的判断来寻找最高位的1，每次只判断是否大于这一轮的中间值，用递归的思想处理前半或者后半

思路来源于博客（找不到了🤔），做出的优化：使用tmp减少运算次数

4. byteSwap

```
int byteSwap(int x, int n, int m) {
    int n_shift = n << 3; // n * 8 得到第 n 个字节的位位置
    int m_shift = m << 3; // m * 8 得到第 m 个字节的位位置

    int mask = (0xFF << n_shift) | (0xFF << m_shift);
    int bytes = ((x >> n_shift) & 0xFF) << m_shift | ((x >> m_shift) & 0xFF) <<
n_shift;

    return (x & ~mask) | bytes;
}
```

思路：

- 提取第n、m字节
- 将原始数字x的n、m位置字节置为0
- 用或运算将字节插入交换后的位置

5. reverse

```
unsigned reverse(unsigned v) {
    v = (v >> 16) | (v << 16); // 每16位一组交换
    v = ((v >> 8) & 0x00FF00FF) | ((v & 0x00FF00FF) << 8); // 每8位一组交换
    v = ((v >> 4) & 0x0F0F0F0F) | ((v & 0x0F0F0F0F) << 4); // 每4位一组交换
    v = ((v >> 2) & 0x33333333) | ((v & 0x33333333) << 2); // 每2位一组交换
    v = ((v >> 1) & 0x55555555) | ((v & 0x55555555) << 1); // 奇偶位交换
    return v;
}
```

思路：

对于32位数，先分成16位一组交换，这样只用递归的交换这两组16位

对于16位数，同样可以分成8位一组交换，再递归处理每组的二进制码

以此类推

6. logicalShift

```
int logicalShift(int x, int n) {
    return (x >> n) & ~(1 << 31 >> n << 1);
}
```

思路：

x 右移 n 位后，用 1 00...00 1111 (注：有 n 个 0) 去把算术右移的符号位扩展置为 0

注：

- `return (x >> n) & ~(0x80000000 >> n << 1);` 中 `0x80000000 >> n << 1` 是算术右移，掩码变为 1 00...00 1111 错误
- `return (x >> n) & ~(1 << 31 >> n << 1);` 中 `1 << 31 >> n << 1` 是逻辑右移

7. leftBitCount

```
int leftBitCount(int x) {
    int cnt = 0; // 初始化计数器 cnt，用于记录左边连续 1 的数量。

    // off 用于处理全 1 的特殊情况。如果 x 全是 1 (~x 全是 0)，off = 1，否则 off = 0。
    // (~x) 是 x 按位取反，!(~x) 是取反后的值再进行逻辑非运算，结果为 1 表示 x 全是 1，否则为 0。
    int off = 1 & !(~x);

    // 检查 x 最高 16 位是否存在 0，如果存在 0，说明前 16 位不是全 1，则向右移 16 位并更新计数器。
    // (!!(~(x >> 16))) 用于判断 x 右移 16 位后的补码是否存在 0。若存在 0，结果为 1；否则为 0。
    cnt += (!!(~(x >> 16))) << 4;

    // 接下来根据 cnt + 8 检查接下来的 8 位是否存在 0。
    // (!!(~(x >> (cnt + 8)))) 判断在 cnt 位置右移 8 位后的 x 是否存在 0。
    cnt += (!!(~(x >> (cnt + 8)))) << 3;

    // 再根据 cnt + 4 检查接下来的 4 位是否存在 0。
    cnt += (!!(~(x >> (cnt + 4)))) << 2;

    // 根据 cnt + 2 检查接下来的 2 位是否存在 0。
    cnt += (!!(~(x >> (cnt + 2)))) << 1;

    // 最后根据 cnt + 1 检查接下来的一位是否存在 0。
    cnt += (!!(~(x >> (cnt + 1))));

    // 返回结果是 32 减去 cnt 加上 off。off 的作用是处理全 1 的情况，如果 x 全是 1，结果会多加 1。
    return 32 + ~cnt + off;
}
```

思路：

与 `reverse` 思路类似，不是遍历所有的二进制码得到左边连续的1，而是先判度高16位是否全是1，

是，则判断更低8位（9~16位）是否全是1

否，则判断更高8位（25~32位）是否全是1

对于这8位，再判断左侧4位是否全是1

以此类推

因为最后一步 `cnt += (!!(~(x >> (cnt + 1))))`，至少会右移一次，所以 `0xffffffff` 和 `0xffffffffe` 需要分开判断（两者的 `cnt` 返回的都是1）故需要 `int off = 1 & !(~x)`；来判断 `0xffffffff`

思路来源于《深入理解计算机系统/CSAPP》Data Lab - 知乎(zhihu.com)，（实在是无法再优化🤖，只能疯狂写注释）

8. float_i2f

```
unsigned float_i2f(int x) {
    if (x == 0)
        return 0;

    unsigned sign = x & (1 << 31);          // 获取符号位, x 的最高位用于表示正负
    unsigned exp = 0;                        // 指数部分, 后续通过移位确定
    unsigned frac = 0;                       // 尾数部分, 将绝对值表示为尾数
    unsigned round = 0;                     // 处理舍入用的中间变量
    unsigned absX = sign ? (~x + 1) : x;     // 取 x 的绝对值, 如果 x 是负数则取反加 1
    unsigned tmp = absX;                    // 临时变量用于计算指数

    // 计算指数: 每次右移 absX, 直到 tmp 变为 0
    // 统计移位次数, 即求出 absX 是几位数, 存储到 exp
    while ((tmp = tmp >> 1))
        ++exp;

    // 用cnt实现exp计算, 会超运算符使用
    // int cnt = ((tmp >> 16)) << 4;
    // exp += cnt;
    // cnt = ((tmp >> (8 + cnt)) != 0) << 3;
    // exp += cnt;
    // cnt = ((tmp >> (4 + cnt)) != 0) << 2;
    // exp += cnt;
    // cnt = ((tmp >> (2 + cnt)) != 0) << 1;
    // exp += cnt;
    // cnt = ((tmp >> (1 + cnt)) != 0);
    // exp += cnt;

    // 计算尾数部分: 将 absX 左移以得到尾数
    // 31 - exp 是为了移到最高位, 并且再左移 1 位用于精度保留
    frac = absX << (31 - exp) << 1;

    // 提取需要用来舍入的部分: 尾数的低位部分, 用于决定是否舍入
    round = frac & 0x1FF;

    // 将尾数右移 9 位, 保留高 23 位的有效部分
    frac = frac >> 9;

    // 舍入规则: 如果舍入部分大于 0xFF + 1 = 0x100, 则进 1, 否则根据舍入部分大小决定是否进位
    // 当舍入部分恰好等于 0xFF + 1 时, 采用尾数的最低位决定是否进位
    round = (round > 0x100) + ((round == 0x100) & (frac & 1));

    // 返回浮点格式: 符号位 | 指数部分 | 尾数部分, 最后加上舍入值
    return (sign | ((exp + 0x7F) << 23) | frac) + round;
}
```

思路:

按int转float的规则实现

思路参考于[ICS Datalab - 知乎 \(zhihu.com\)](https://www.zhihu.com/question/20066106)

• float_i2f(x)

```

/*
 * float_i2f - Return bit-level equivalent of expression (float) x
 * Result is returned as unsigned int, but
 * it is to be interpreted as the bit-level representation of a
 * single-precision floating point values.
 * Legal ops: Any integer/unsigned operations incl. ||, &&, also if, while
 * Max ops: 30
 * Rating: 4
 */
unsigned float_i2f(int x) {
    unsigned sign = x & (1 << 31);
    unsigned exp = 0;
    unsigned frac = 0;
    unsigned round = 0;
    unsigned absX = sign ? (~x + 1) : x;
    unsigned tmp = absX;
    while ((tmp = tmp >> 1))
        ++exp;
    frac = absX << (31 - exp) << 1;
    round = frac << 23 >> 23;
    frac = frac >> 9;
    if (round > 0xFF + 1) round = 1;
    else if (round < 0xFF + 1) round = 0;
    else round = frac & 1;
    return x ? (sign | ((exp + 0x7F) << 23) | frac) + round : 0;
}

```

做出的优化

- 优化了round的计算 `round = (round > 0x100) + ((round == 0x100) & (frac & 1));`
- 尝试优化取消while循环，但是会超运算符使用次数

```

// 用cnt实现exp计算，会超运算符使用
// int cnt = ((tmp >> 16)) << 4;
// exp += cnt;
// cnt = ((tmp >> (8 + cnt)) != 0) << 3;
// exp += cnt;
// cnt = ((tmp >> (4 + cnt)) != 0) << 2;
// exp += cnt;
// cnt = ((tmp >> (2 + cnt)) != 0) << 1;
// exp += cnt;
// cnt = ((tmp >> (1 + cnt)) != 0);
// exp += cnt;

```

9. floatScale2

```
unsigned floatScale2(unsigned uf) {
    int exp = uf & 0x7f800000;    // 提取指数部分
    int sign = (uf & 0x80000000); // 提取符号位
    if (exp == 0)                  // 非规格化数或0
        return (uf << 1) | sign;  // 左移一位并保留符号位
    if (exp == 0x7f800000)         // 无穷大或NaN
        return uf;
    exp += 0x800000;               // 指数加1（乘以2）
    if (exp == 0x7f800000)         // 溢出到无穷大
        return 0x7f800000 | sign; // 返回无穷大并保留符号位
    return (uf & 0x807fffff) | exp; // 返回新浮点数：更新后的指数加原符号和小数部分
}
```

思路：

处理非规格化数

- $\text{frac} = f \& 7\text{FFFF}$;
- if ($\text{frac} \gg 22 == 0$), $f*2$ 还是非规格化数, frac 直接左移一位
- if ($\text{frac} \gg 22 == 1$), $f*2$ 是规格化数, frac 左移一位后, frac 最高位的1变为 exp 的一位, exp 由0变1 (结果乘2), 规格化数 frac 最高位隐藏

所以处理非规格化数只需返回 $(uf \ll 1) | \text{sign}$

10. float64_f2i

```
int float64_f2i(unsigned uf1, unsigned uf2) {
    int s = uf2 & 0x80000000;    // 符号位
    int exp = (uf2 >> 20) & 0x7FF; // 指数部分
    int frac = ((uf2 & 0xFFFFF) << 12) | (uf1 >> 20); // 合并尾数

    int E = exp - 1023; // 调整指数
    if (E < 0)
        return 0; // 指数小于0, 表示值小于1, 返回0（下溢）

    if (E >= 31)
        return 0x80000000; // 指数过大, 溢出, 返回最大值

    frac = (frac | 0x100000) >> (20 - E); // 恢复隐含的1位, 并根据E调整尾数

    return s ? -frac : frac; // 根据符号返回最终结果
}
```

思路：

对于c语言中的double:

sign	exp	frac
1位	11位	52位

提取符号: $\text{int } s = \text{uf2} \& 0x80000000;$

提取exp: `int exp = (uf2 >> 20) & 0x7FF;`

0x7FF 为11位1，可以保留11位的exp并将剩余位置为0

提取合并尾数：尾数部分由 `uf2` 的低20位和 `uf1` 的高12位拼接而成。将 `uf2` 的低20位左移12位后与 `uf1` 的高12位合并，构成完整的尾数

11. floatPower2

```
unsigned floatPower2(int x) {
    if (x < -149) // 结果太小，返回 0
        return 0;
    if (x < -126) // 非规格化数的情况
        return 1 << (149 + x); // 计算出尾数的二进制表示
    if (x < 128) // 正常情况，计算指数部分
        return (x + 127) << 23; // 指数转换为浮点格式
    return 0x7F800000; // 结果太大，返回正无穷大
}
```

思路（ics课上有讲过这题，思路与课堂讲解的代码一致）：

对于c语言中的float：

sign	exp	frac
1位	8位	23位

bias = 127（规格化数） 或 126（非规格化数）

- 当 $x < -(23 + 126) = -149$ 时，结果太小，float的精度无法表示，返回0
- 当 $-149 \leq x < -126$ 时，结果为非规格化数， $exp = 0$
- 当 $x < 128$ 时， $exp < 255$ ，结果为规格化数， $exp = x + 127$
- 当 $x > 127$ 时， $exp > 254$ ，结果为无穷