DataLab 实验报告

2020201018 涂奕腾

1. bitXor

利用异或运算的等价表达：

$$A \wedge B = (\neg A \cap B) \cup (A \cap \neg B) = \neg(\neg(A \cap \neg B) \cap \neg(\neg A \cap B))$$
$$= \neg((A \cap B) \cup (\neg A \cap \neg B)) = \neg(A \cap B) \cap \neg(\neg A \cap \neg B)$$

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

2. evenBits

通过左移直接构造即可。

```
int evenBits(void) {
    int x = 0x55;
    x |=  x << 8;
    x |=  x << 16;
    return x;
}
```

3. fitsShort

满足条件的数二进制表示中高 17 位上应当全为 0 或全为 1。可以先将 x 右移 15 位，此时满足条件的数各位上应当全为 0 或全为 1，与原数右移 31 的值相等。

```
int fitsShort(int x) {
    return !((x>>31)^(x>>15));
}
```

4. isTmax

首先考虑 Tmax 的性质：当$x = Tmax$时，我们记$y = x + 1$则有：$y = \sim x$

满足这一式子的除了 $x = Tmax$ 外还有 $x = -1$，不难想到我们可以用 $!y$ 快速判断 -1：

```
int isTmax(int x) {
    int y=x+1;
    return !~(x^y) ^ !y;
}
```

考虑进一步优化，在上面的想法中我们在 $!\sim(x\hat{}y)$ 中用 4 个运算符构造出了“0”，接下来我们尝试仅用 3 个运算符表示“0”。观察 $x = Tmax$，可以发现此时有如下等式：

$$x + y = 0xFFFFFFFF = -1$$

故有：

$$0 = x + y + 1 = y + y$$

如此，我们仅用三个运算符就构造出了“0”：

```
int isTmax(int x) {
    int y = x + 1;
    return (!(y + y)) ^ (!y);
}
```

这份代码在本体和 Ubuntu 上均可通过，但在平台上无法通过，于是将其改为另一形式上等价的表达后方可通过：

```
int isTmax(int x) {
    int y = x + 1;
    return !((y + y) | !y);
}
```

5. fitsBits

类比 3 中 fitsShort 函数，我们只需将 x 右移 n - 1 位与 x 右移 31 位进行比较即可：

```
int fitsBits(int x, int n) {
    return !((x >> 31) ^ (x >> n + (~0)));
}
```

进一步优化，利用右移时超越位数自动取模的性质，右移 n-1 位即为右移 31+n 位，可优化至 5 个运算符：

```
int fitsBits(int x, int n) {
    return !((x >> 31) ^ (x >> 31 + n));
}
```

6. upperBits

构造高 n 位为 1 的数，可以考虑将 1<<31 右移 n-1 位，特判 n=0 的情况即可。

```
int upperBits(int n) {
    return (((!!n) << 31) >> n + (~0));
}
```

同 5 中的优化方法，可以通过超越位数取模优化至 5 个运算符。

```
int fitsBits(int x, int n) {
    return (!!n) << 31 >> 31 + n;
}
```

7. addOddBits

构造出奇数位均为 1 的掩码，用 & 操作提取奇数位上的 1，与掩码比较即可。

```
int allOddBits(int x) {
    int y = 0xAA;
    y |= y << 8;
    y |= y << 16;
    return !((x & y) ^ y);
}
```

8. byteSwap

利用异或运算的性质：$x \wedge y \wedge y = x$，我们考虑通过右移操作和掩码提取出这两个字节上的数进行异或操作得到它们的异或和，再通过左移至相应字节再次进行异或完成字节上数值的交换。

```
int byteSwap(int x, int n, int m) {
    n <<= 3;
    m <<= 3;
    int tmp = ((x>>n) ^ (x>>m)) & (0xFF);
    return x ^ (tmp<<n) ^ (tmp<<m);
}
```

9. absVal

当 x 为正时无需操作，当 x 为负时需将 x 取负。

由 $-x = \sim x + 1$ 可得：$-x = \sim(x - 1)$，同时注意到 $\sim x = x \wedge (-1)$，故当 x 为负数时只需：$(x - 1) \wedge (-1)$ 即可获得其绝对值。由于当 x 为正时右移 31 为 0；当 x 为负时右移 31 位为 -1，而$x \wedge 0 = x$，我们便得到了通式。

```
int absVal(int x) {
    int flag = x >> 31;
    return (x + flag) ^ flag;
}
```

10. divpwr2

当输入的 x 为正数和 0 时直接右移 n 位即可，当 x 位负数时应当加上 $2^n - 1$ 的偏置量再进行右移操作。偏置量的二进制表示中低 n 位为 1，高位为 0. 记 $y = x >> 31$,则易构造出偏置量为 $y \wedge (y << n)$.

```
int divpwr2(int x, int n) {
    int y = x >> 31;
    return (x + (y ^ (y << n))) >> n;
}
```

11. leastBitPos

即为树状数组中常用的 lowbit 函数，不作赘述。

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

12. logicalNeg

考虑相反数的性质，只有 0 与其相反数的最高位都为 0，取 x 与其相反数进行或运算的结果右移 31 位，只有 0 能得 0，其余数都得 -1.

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

13. bitMask

考虑构造两个数取与：一个数 x 的第 0 位到第 highbit 位为 1，高位为 0；另一个数第 31 位到第 lowbit 位为 1，低位为 0。对于 y，我们可以通过 -1 左移 lowbit 位得到，而对于 x，可以通过 $-1 + 2^{highbit+1}$ 得到。为了防止溢出可以通过1 << height << 1实现，另一种更优秀的做法是 2 << height。

```
int bitMask(int highbit, int lowbit) {
    int t = ~0;
    return (t + (2 << highbit)) & (t << lowbit);
}
```

14. isLess

$x < y$ 可以分为以下两种情况判断：

1)当 $x < 0$ 且 $y > 0$ 时直接判断

2)当 x, y 同号时应当满足 $x - y < 0$ ，此时因为同号可以保证减法不会溢出

我们记 $z = \sim y$,则：

情况 1)下满足条件时 x 和 z 的符号位应均为 1

情况 2)下满足条件时 x 和 z 的符号位相反且 $x - y$ (即 $x + z + 1$) 的符号位为 1

将两种情况取或即可。

```
int isLess(int x, int y) {
    y = ~y;
    int tmp1 = (x + y + 1) & (x ^ y);
    int tmp2 = x & y;
    return ((tmp1 | tmp2) >> 31) & 1;
}
```

15. logicalShift

考虑构造高 n 位为 0，低位为 1 的掩码，与右移后结果取 & 即可。构造掩码我们可以从取反入手，构造高 n 位为 1，低位为 0 的数，即将 TMin 右移 n – 1 位，由于低位一定是 0，可以先右移 n 位再左移 1 位实现。

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

16. satMul2

首先计算出 $y = x * 2$, 由 x 和 y 的最高位是否相同判断是否溢出，并由 y 的最高位判断是正溢出还是负溢出，并通过与 Tmin 进行异或操作得到溢出时目标的输出。由此我们可以得到一个初步的代码（详细见注释）：

```
int satMul2(int x) {
    int y = x<<1;
    int tmp1 = x>>31;
    int tmp2 = y>>31;
    int overflow = tmp1 ^ tmp2;//不溢出：0   溢出：-1
    int mask = ((1<<31) ^ tmp2) & overflow;//不溢出：0，正溢出：0x7FFFFFFF，负溢出：0x80000000
    y = ~overflow & y;//不溢出：y  溢出：0
    return y | mask;
}
```

接着进行优化。考虑构造异或的掩码 mask，当未溢出时 mask=0，溢出时 mask 符号位为 1，正溢出时低位为 y 取反，负溢出时低位与 y 相同，如此通过一个异或操作就能得到目标的输出。构造过程如下：

$$(1 << 31)\ \hat{}\ \text{tmp2} = \begin{cases} 0\text{x7FFFFFFF}, & \text{y 为负数} \\ 0\text{x80000000}, & \text{y 为正数} \end{cases}$$

记 $z = (1 << 31)\ \hat{}\ tmp2\ \hat{}\ y$，则 y 为负时，z 的符号位为 1,其余位为 y 对应位取反；y 为正时，z 符号位为 1，其余位与 y 的对应位相同。

排除不溢出的情况，z & overflow 即为所求的异或掩码。

```
int satMul2(int x) {
    int y = x<<1;
    int tmp1 = x>>31;
    int tmp2 = y>>31;
    int overflow = tmp1 ^ tmp2;
    int mask = ((1<<31) ^ tmp2 ^ y) & overflow;
    return y ^ mask;
}
```

17. subOK

类似于 14 isLess 函数的实现思路，只有当 x, y 异号时才可能出现 x - y 溢出的情况，且溢出时 x – y 应与 x 异号。

```
int subOK(int x, int y) {
    return !((((x + ~y + 1) ^ x) & (x ^ y)) >> 31);
}
```

18. bitParity

由于只需要求出 x 中 0 或 1 的个数的奇偶性，我们联想到可以用异或来维护。考虑分治的思想，将 x 二进制表示的前半部分和后半部分进行异或操作，其 1 个数的奇偶性得以保留(因为在异或中 1 只能成对消去，当两个数该位置同时为 1 时该位的异或结果为 0，此时消去 2 个 1，奇偶性不变)。通过不断分治，我们可以将 1 的奇偶性信息集中到第 0 位上，判断第 0 位是 0 或 1 即可判断 x 中含 0 或 1 个数的奇偶。

```
int bitParity(int x) {
    x ^= x >> 16;
    x ^= x >> 8;
    x ^= x >> 4;
    x ^= x >> 2;
    x ^= x >> 1;
    return x & 1;
}
```

19. isPower2

根据 popcount 函数中对 x 的操作 $x\ \&\ (x-1)$，我们可以得到将 x 最低位上的 1 变成 0 后的数。如果 x 是 2 的幂次方，则进行一次此操作后应当得到 0。此外还需要特判 x = 0 或 0x80000000 的情况，注意到这两个数左移 1 位后得到的值都是 0，可以进行特判。易得到如下判断条件：

```
return (!(x & (x + ~0))) & (!!(x << 1));
```

根据逻辑运算规律，可以优化掉一个运算符:

```
int isPower2(int x) {
    return ! ((x & (x + ~0)) | (!(x << 1)));
}
```

20. float_i2f

根据课本，对于一个正整数 x，将其转化为浮点数的表示的步骤如下:

(1)找到其最高位的 1，记最高位的 1 所在位为 m

(2)小数点左移 m 位

(3)丢弃最高位的 1，并在末尾补充 23 – m 个 0 得小数字段 frac

(4)m + 127 得阶码字段 e

(5)补上符号位 s

负数转化成正数判断，而 0 是非规范化值，应当特判掉。

代码解释见注释部分。

```
unsigned float_i2f(int x) {
    if(!x) return 0; //特判 0
    unsigned y = x,sign = 0, shift = 0, flag = 0, tmp;
    if(x < 0) {//记录负数符号位
        y = -x;
        sign = 0x80000000;
    }
    while(1){//应左移 shift-1 位，实际多左移 1 位以消去最高位的 1
        tmp = y; y <<= 1; shift++;
        if(tmp & 0x80000000) break;
    }
    //y 前 23 位为尾码部分，四舍五入判断进位
    flag = (((y & 0x1ff) > 0x100)| ((y & 0x3ff) == 0x300));
    return  sign + ((159 - shift) << 23) + (y >> 9) + flag;
}
```

21. leftBitCount

将 x 取反后问题转化为求 x 前导 0 的个数，只需找到最高位 1 所在的位置即可。不难想到运用二分查找的思想，先检查高 16 位是否均为 0，若$x >> 16 != 0$，则最高位的 1 至少在 16 位及以上，反之则在 15 位及以下，我们再对最高位 1 所在的区间进行二分查找，以此类推。在函数中用 y 记录答案，用 z 记录二分查找的结果，用 n 来控制查找的区间。查找区间长度为 2 之后特判即可。

```
int leftBitCount(int x) {
    x = ~x;
    int y, z = !(x >> 16), n = z << 4;
    //当第 16-31 位均为 0 时,n=16,否则 n=0;
    y = n; x <<= n; z = !(x >> 24); n = z << 3;
    //当第 16-31 位均为 0 时查找第 8-15 位，否则查找第 24-31 位，后面
    //以此类推
    y = y + n; x <<= n; z = !(x >> 28); n = z << 2;
    y = y + n; x <<= n; z = !(x >> 30); n = z << 1;
    y = y + n; x <<= n;
    y = y + !(x >> 31) + !(x >> 30);
    return y;
}
```