ICS I datalab 报告

中国人民大学 sheriyuo

摘 要

RUC 2023 计算机理论基础 I datalab 的详细实现和一定优化方案。

不理解为什么要卷运算符个数，更不理解为什么要把去年最优解设为今年的 95 分。

独立完成一些 95 分已耗费笔者太多精力，水平有限，告辞。排名 12/195。

```
bitXor 0.95 7/7/7/7
thirdBits 0.95 4/4/4/4
fitsShort 0.95 4/4/4/4
isTmax 0.95 5/5/5/6
fitsBits 1.90 5/5/6/7
upperBits 1.00 4/5/6/9
anyOddBit 1.90 7/7/7/7
byteSwap 1.90 10/10/10/17
absVal 3.80 3/3/3/5
divpwr2 1.90 5/5/6/7
float_neg 1.90 4/4/4/6
logicalNeg 3.80 5/5/5/6
bitMask 2.85 5/5/6/8
isGreater 2.85 7/7/9/12
logicalShift 2.85 5/5/6/14
satMul2 2.85 6/6/7/10
subOK 2.85 7/7/8/12
trueThreeFourths 3.80 9/9/11/12
isPower2 3.80 6/6/7/11
float_i2f 4.00 14/15/21/32
howManyBits 3.80 22/22/28/37
float_half 3.80 12/12/16/22
---
sum = 95.4310
```

1 bitXor

用 `~` 和 `&` 实现位运算异或。

提取 0 的贡献，运用德摩根律有

$$x \oplus y = \overline{xy \cup \overline{xy}} = \overline{xy} \cap \overline{(\overline{xy})}$$

需要 7 个运算符，提取 1 的贡献则需要 8 个运算符。

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

2 thirdBits

求出从 LSB 开始，每个第 3 位为 1 其余位为 0 的数。

数字上限是 $0 \sim 255$，所以采用初始值 `0x49`，左移倍增两次即可，需要 4 个运算符。

```c
int thirdBits(void) {
    int x = 0x49;
    x = x | (x << 9);
    x = x | (x << 18);
    return x;
}
```

3 fitsShort

求出 x 是否在 `short` 范围内。

若一个 `int` 的值在 `short` 范围内，那么符号位后 16 位应该与符号位相同，再后 15 位为 `short` 有效位。

运用算术右移，异或判断是否相等即可，需要 4 个运算符。

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

4 isTmax

求出 x 是否是 `INT_MAX`，不能使用移位。

若 `x = INT_MAX`，那么 `y = x + 1`，此时 `y + y = 0`。

那么判断 `!(y + y)`，并且特判同时成立的 `x = UINT_MAX` 即可，需要 5 个运算符。

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

5 fitsBits

求出 x 是否能用 $n(1 \le n \le 32)$ 位二进制位存储。

与 3 相同，符号位后 $32 - n$ 位应该与符号位相同，再后 n 位为有效位。

运用算术右移，异或判断是否相等即可。

注意到 `x >> n - 1` 实际上需要 `x >> n + ~0`，可以利用移位位数会 mod32 的性质，写成 `x >> n + 31`，只需要 5 个运算符。

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

6 upperBits

求出前 $n(0 \le n \le 32)$ 高位二进制位为 1 其余位为 0 的数。

先求出 `1 << 31`，然后运用算数右移的性质右移 $n - 1$ 位即可。

同 5 的优化，可以用 `x >> n + 31` 省去一次取反。

特判 $n = 0$ 的情况，将 `1 << 31` 替换为 `!!n << 31` 即可，需要 5 个运算符。

`!!n` 还存在优化空间，发现 `(n + 31) & 32` 只会在 $n = 0$ 时取 0，于是将 `!!n << 31` 替换为 `(n + 31 & 32) << 26` 即可，只需要 4 个运算符。

```
int upperBits(int n) {
    int x = n + 31;
    return (x & 32) << 26 >> x;
}
```

7 anyOddBit

判断 x 奇数位上是否有 1。

用 `0xaa` 左移倍增两次得到所有奇数位都为 1 的数，取位运算与后转 `bool` 即可。

```
int anyOddBit(int x) {
    int y = 0xaa;
    y = y | (y << 8);
    y = y | (y << 16);
    return !!(y & x);
}
```

8 byteSwap

求出 x 交换第 $n, m(0 \leq n, m \leq 3)$ 个字节后的数。

利用异或结合律实现 `swap`，将要交换的两个字节异或后放在最低有效字节上，取该字节左移分别 n, m 位与原数作异或操作，即可得到交换之后的数，需要 10 个运算符。

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

9 absVal

求出 x 的绝对值。

若 x 是负数，其绝对值即为 `-x = ~(x - 1)`。而注意到 `~` 和 `-1` 采用同样的补码进行操作，提出 `y = x >> 31` 来作为运算的同时也不影响正数的结果，答案为 `(x + y) ^ y`，需要 3 个运算符。

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

10 divpwr2

求出 x 除以 $2^n(0 \le n \le 30)$ 向下取整的结果。

正数的右移是向下取整的，而负数的右移是向上取整的，给负数加上 $2^n - 1$ 的偏移量后右移 n 位即可。

一种偏移量的求法是，获取符号位 `y = x >> 31`，用 9 中 `y` 也可以用来代替 `-1` 的技巧，有 `k = ((1 << n) + y) & y`，需要 6 个运算符。

使用 `1 << n` 是可以优化的，利用 `x >> 31` 的补码性质，有 `k = y ^ (y << n)`，只需要 5 个运算符。

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

11 float_neg

给定浮点数 f 的 `unsigned` 二进制表示，求出 $-f$ 的二进制表示。若 `f = nan`，返回 `nan`，可以使用所有运算符、`unsigned` 及其范围内常数和 `if`，`while`。

判断 `nan` 只需要判断其数位是否有 1，即 `nf & 0x7f800000 > 0x7f800000`。如果不是 `nan`，异或 `0x80000000` 修改符号位即可，需要 4 个运算符。

```
unsigned float_neg(unsigned uf) {
    if((uf & 0x7fffffff) > 0x7f800000)
    return uf;
    return uf ^ 0x80000000;
}
```

12 logicalNeg

使用去除 `!` 的位运算符实现 `!x`。

考虑只有 0 的负数为 0 本身，所以 `!x` 为假有 `x | -x` 符号位为 1，提出符号位设为 0 即可。

可以直接 `((x | ~x + 1) >> 31) + 1`，需要 4 个运算符。

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

```
}
```

13 bitMask

求出二进制位第 $lowbit \sim highbit$ 位为 1 的数，$lowbit > highbit$ 时为 0。

可以通过 `(~0 << lowbit) ^ (~0 << highbit << 1)` 的方法来得到这个数，但是该方法无法特判 0 的情况，需要再与上一个 `~0 << lowbit`，需要 6 个运算符。

考虑优化的本质是去除特判，所以不应该采用异或而是位运算与。对于 $highbit$，采用 `~0 + (1 << highbit << 1)` 即可得到第 $0 \sim highbit$ 位为 1 的数，直接取与，此时仍需要 6 个运算符。

发现可以用 `2 << highbit` 替换 `1 << highbit << 1`，也回避了 `<< 32` 的 ub，只需要 5 个运算符。

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

14 isGreater

判断是否有 $x > y$。

如果 x, y 符号位相同，`x + ~y` 的符号位为 1 时有 $x \leq y$，符号位为 0 时有 $x > y$，直接用 `(x + ~y >> 31) + 1` 提取即可。

如果 x, y 符号位不同，直接判断是否有 $x > -1$ 即可，改为 `x + ~(x ^ y >> 31 | y)`，需要 7 个运算符。

```c
int isGreater(int x, int y) {
    return ((x + ~(((x ^ y) >> 31) | y)) >> 31) + 1;
}
```

15 logicalShift

对 `int` 实现逻辑右移。

算术右移 `x >> n` 后，原最高位在第 $31 - n$ 位，此时应将原最高位前的数位全部补 0。

于是让 `x >> n` 加上 `1 << (31 - n)` 后再异或上 `1 << (31 - n)` 即可。

可以用 `31 ^ n` 来实现 `31 - n`，只需要 5 个运算符。

```
int logicalShift(int x, int n) {
    int t = 1 << (31 ^ n);
    return ((x >> n) + t) ^ t;
}
```

16 satMul2

求出 $x \times 2$，如果 $x \times 2$ 溢出超过 $Tmin$ 或 $Tmax$，将其赋值为 $Tmin$ 或 $Tmax$。

溢出的条件是符号位不同，即 `x ^ (x << 1) >> 31 = -1 = y`，否则 `y = 0`。

如果出现溢出，可以发现 `y << y = -1 << -1 = 1 << 31` 本质上是在求 $Tmin$，此时 `(x << 1) >> y = (x << 1) >> 31` 就是 $x \times 2$ 的符号位，如果溢出为负数为 -1，溢出为正数为 0。

巧妙的是，$Tmin - 1 = Tmax$，于是答案即为 `(y << y) + ((x << 1) >> y)`，只需要 6 个运算符。

```
int satMul2(int x) {
    int x2 = x << 1;
    int y = (x ^ x2) >> 31;
    return (y << y) + (x2 >> y);
}
```

17 subOK

判断 $x - y$ 是否出现整型溢出。

$x - y$ 出现整型溢出当且仅当 x, y 符号位不同，且 $x, x - y$ 符号位也不同，直接用 `(x ^ y) & (x ^ (x + ~y + 1))` 判断即可，需要 8 个运算符。

考虑优化，发现 $x - y$ 是否溢出跟 $y - x - 1$ 是否溢出等价，可以优化掉取反后的 `+ 1`，只需要 7 个运算符。

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

18 trueThreeFourths

计算 $x \times \frac{3}{4}$ 向 0 取整后的结果。

`x + (~x >> 2)` 计算的是 $\lfloor \frac{3}{4}x - \frac{1}{4} \rfloor$，$x \geq 0 \land x \bmod 4 = 0$ 时需要补上 1，$x < 0$ 时需要补上 -1。

用 `(x >> 31 & 1) | !(x & 3)` 判断，需要 9 个运算符。

```c
int trueThreeFourths(int x) {
    int s = x >> 31 & 1;
    return x + (~x >> 2) + (s | !(x & 3));
}
```

19 isPower2

判断 x 是否是 2 的次幂。

由 `lowbit` 的性质可知，x 二进制位 1 只能有一个，满足 `x & (x - 1) = 0`。

发现需要特判 `0` 和 `1 << 31` 两种情况，用符号位判断或 `!x` 判断较劣，可以构造出 `!(x << 1)` 的判断方法，此时需要 7 个运算符。

考虑利用计算的 `x + ~0`，发现两个数分别变为 `0xffffffff` 和 `0x7fffffff`。由于合法的情况最大只能是 `1 << 30`，直接用 `(x + ~0) >> 30` 判断即可，只需要 6 个运算符。

```c
int isPower2(int x) {
    int y = x + ~0;
    return !((y >> 30) + (x & y));
}
```

20 float_i2f

将 x 转换为 `float`，求出转换后 `unsigned` 下的二进制位，可以使用所有运算符、`unsigned` 及其范围内常数和 `if`，`while`。

初始 `int` 的阶码为 $127 + 30$，若 $x \geq 0$，将 x 转为 `unsigned`，令 `exp = 0x4e800000`；否则，将 $-x$ 转为 `unsigned`，令 `exp = 0xce800000`。

令其为 ux，找到 ux 最高位的 1，移位过程中每一位有 `ux <<= 1`，`exp -= 0x800000`，并用 `if(ux & 0x17f) uf += 0x80;` 来判断尾数四舍六入的情况，此时 `exp + (ux >> 8)` 即为答案。

若进位导致 `ux >> 8` 为 0，需要在 `exp` 上再补 `0x01000000` 的阶码。

可以把判断符号位写在循环内，同时需要特判 0（不要 `if(!x)`），需要 14 个运算符。

```c
unsigned float_i2f(int x) {
    unsigned ux = x;
    unsigned e = 0x4e800000;
    int op = 0;
    if(x) {
        while(1) {
            if(ux & 0x80000000) {
                if(op)
                break;
                else {
                    ux = -x;
                    e = 0xce800000;
                }
            }
            else {
                ux <<= 1;
                e -= 0x800000;
            }
            op = 1;
        }
        if(ux & 0x17f)
        ux += 0x80;
        ux >>= 8;
        return e + (ux ? ux : 0x01000000);
    }
    return 0;
}
```

21 howManyBits

计算出表示 x 的最少补码位数。

对于正数，最少补码位数即最高位 1 的位数 +1；对于负数，为最高位 0 的位数 +1。可以用 `x ^= x << 1` 来将负数的 0 转为 1，此时最高位 1 的位数即为答案。

对于最高位的 1，二分 5 次查找，最后 1 次可以直接判断，重点在于优化单次查找的运算符数。

直接 `!!(x >> 16) << 4` 由于两个 `!`，含累加答案单次需要 6 个运算符。一种优化是将初始答案改为 $s = 31$，采用异或的方式累加，可以节省掉一个 `!`，单次需要 5 个运算符。

共需要 26 个运算符。

```c
int howManyBits(int x) {
    int s = 31, y, p;
    x ^= x << 1;
    y = x >> 16; p = !y << 4;
    s ^= p; x <<= p;
```

```c
    y = x >> 24; p = !y << 3;
    s ^= p; x <<= p;
    y = x >> 28; p = !y << 2;
    s ^= p; x <<= p;
    y = x >> 30; p = !y << 1;
    s ^= p; x <<= p;
    s ^= !(x >> 31);
    return s + 1;
}
```

还有优化空间。不对 x 进行移位操作，直接对 ans 进行异或计算，最后累加上构造移位的答案，单次只需要 4 个运算符，总共只需要 22 个运算符。

```c
int howManyBits(int x) {
    int ans;
    x = x ^ (x << 1);
    ans = !(x >> 16) << 4;
    ans ^= 24;
    ans ^= !(x >> ans) << 3;
    ans ^= 4;
    ans ^= !(x >> ans) << 2;
    ans += ~0x5b >> (x >> ans & 30) & 3;
    return ans + 1;
}
```

22 float_half

给出浮点数 f 的 `unsigned` 表达，计算 $0.5 \times f$ 的值，如果 f 为 `nan` 返回本身。可以使用所有运算符、`unsigned` 及其范围内常数和 `if`，`while`。

对于 `nan` 和 `inf`，直接返回本身。对于非规格化数，尾数右移 1 位并处理舍入，可同时处理 0，用 `if(exp <= 0x800000)` 判断。对于规格化数，阶码减 1 即可。

舍入需要满足 `op = (uf & 3) == 3` 的情况，结果为 `s | ((uf ^ s) + op) >> 1`。

利用 `uf << 1 >> 1` 去除符号位，可以优化为 `s | ((uf << 1) + (uf & 3)) >> 2`，只有 `uf & 3 == 3` 时才会实质上 $+1$，省去一个 `==`，需要 12 个运算符。

```c
unsigned float_half(unsigned uf) {
    unsigned s = uf & 0x80000000;
    unsigned e = uf & 0x7f800000;
    int op = uf & 3;
    if(e == 0x7f800000)
        return uf;
    else if(e <= 0x800000)
        return s | ((uf << 1) + op) >> 2;
```

```
    else
        return uf - 0x800000;
}
```