ICS Datalab Report

胡译文 2021201719

Results

Quick Takeaways

- `int sign = x >> 31;` 符号位填充满整个变量，可用于有条件取反
- `(~x) == (-x-1)` 一定程度上起到负号（减法）的作用（除 `0` 和 `Tmin`）
- `!(x | y)` 连等于一个常数（如零）
- `!(x ^ y)` 判断两数是否相等
- `补码 = [](int x) { return (~x) + 1; }`
- 超出位数的位移运算是未定义行为，在某些编译器上位移运算会取模处理（不要在其他地方使用该性质）
- `NaN`: `sign`=0/1, `exponent`=all 1s, `mantisa≠0`
- `inf`: `sign`=0/1, `exponent`=all 1s, `mantisa`=0
- 熟练运用离散数学
    - `(~x) ^ y == ~(x ^ y)`
- IEEE754 标准

Solutions

bitXor

重点在于利用 `&` 和 `~` 表示 `|`。`x^y = ~(x&y) & ~(~x & ~y)`

thirdBits

一开始做的时候有一个歧义，每三位一个1的1从哪里开始。解决完歧义后剩下的就是定义变量，指数级复制。（在 logicalNeg 证明了以 `2` 为底是最优解）

fitsShort

最暴力思路：判断除符号位、低15位以外有没有1（负数是判断有没有0）（op数太多orz）

优化：判断有没有 1（或 0 ）就是其实在判断有没有非符号位。利用右移填充符号位的性质+高位应与符号位相同的性质，判断填充前和填充后是否相等即可。

```
int trunc = x >> 15;
return !((x >> 31) ^ trunc);
```

isTmax

本题关键在于探索 `Tmax` 的性质：1. `Tmax+1==Tmin` 2. `~Tmax==Tmin` 。但同时拥有这个性质的还有 `-1` （观察可以发现两者仅符号位不同）。令 `y=x+1`：

一开始利用同时利用两个性质再排除 `x+1==0` 情况：`(!( x ^ y ^ ~(!x))) & ( !! y)` 同时还实验了好几个思路：`!((y+x)^(~(!y)))`，`!(~(y+x+!y))`

优化思路时发现 `Tmax+1` 的性质：`(Tmax+1) * 2 == 0`，遂变成：`!((y+y) | (!y))`。

fitsBits

直接将 fitsShort 第一种解法中的 `15` 更换为 `~0 + n`，结果操作符超标~~尝试德摩根律优化（位运算不符合）~~。而若将第二种解法的 `16` 直接更换为 `n`，则报错:-)。

同学提示“位移会取模”，遂优化第二种解法的减法：

```
int trunc = x >> (n + 31);
return !((x >> 31) ^ trunc);
```

upperBits

构造掩码即可，一开始将 `~0` 左移 `32 - n`，其中 `n == 0 || n == 32` 时返回都是 `~0`，特判 `0` 的情况 `+1` 即可。最后运算符太多。本题灵活度很高，尝试各种姿势的构造：

```
int nn = !n;
return ret = ((~nn) << (~n + !nn));
```

但可能左移操作已经到顶，尝试上一题的优化，改用减法和右移：

```
int tmin = 1 << 31;
return (tmin >> (n + 31)) + !n;
```

anyOddBit

构造掩码即可，返回 `!!(odd & x)` 即可。

byteSwap

模拟思路：（~~一开始蠢蠢的连 `*8` 都不会写了，写了三次加法~~）把对应位数用掩码取下来，交换，再放回去。`int get_n = ((x & mask_n) >> octuple_n);`

同学提示“异或”，遂想起来无临时变量交换int值的方法，可以不需要把原数“挖洞”，简化操作：

```c
int octuple_n = n << 3;
int octuple_m = m << 3;
int mask_n = 0xff << octuple_n;
int mask_m = 0xff << octuple_m;
int get_n = ((x & mask_n) >> octuple_n);
int get_m = ((x & mask_m) >> octuple_m);
int mix = (get_n ^ get_m) & 0xff;
return x ^ (mix << octuple_m) ^ (mix << octuple_n);
```

最后发现其实没必要获得掩码 `mask_n`）

```c
int octuple_n = n << 3;
int octuple_m = m << 3;
int get_n = (x >> octuple_n);
int get_m = (x >> octuple_m);
int mix = (get_n ^ get_m) & 0xff;
return x ^ (mix << octuple_m) ^ (mix << octuple_n);
```

absVal

简单版：`(x ^ sign) + (sign & 1)`

压缩版：`(x + sign) ^ sign`（分析发现除加一部分`sign & 1`都不能省略，遂考虑更换运算顺序。`x` 取反之前减一即取反之后加一，遂得。）

divpwr2

（~~蠢蠢的不会算除法~~）右移是向下取整，题目要求是向零取整，区别在于负数。解决方法就是把负数增加一个小量（这里说的是字面值，不是绝对值），强制“退位”。小量试过直接 $2^n - 1$，但运算符太多。遂将 `-1` 利用取反简化：`(sign << n) ^ sign`。

float_neg

一开始直接异或最高位，没有考虑 `NaN` 的情况。

```
if ((uf & 0x7FFFFFFFU) > 0x7F800000U) {
    return uf;
} else {
    return uf ^ 0x80000000U;
}
```

logicalNeg

暴力思路：判断每一位有没有值。$\log_m(32) \times 2(m-1)$ 次运算有点多，其中 m 表示以多少为底数（2 时取得最小值 10）。

聪明思路：与 isTmax 类似，本题找 `0` 的特征：`(~0) == Tmin`，`Tmin + 1 == Tmax`。其中最重要的特征是 `补码(0) == 0 && 补码(Tmin) == Tmin`，不改变符号位。`((x | ((~x) + 1)) >> 31) + 1`

bitMask

首先获取截止到高位的 Mask：

```
int sign = ~1;
int mask = ~(sign << highbit);
```

这里的优化用 `~1` 免除减一操作。接着右移左移移除低位多余 Mask。

isGreater

一开始用减法判断，发现会溢出~~，遂暴力 `(x >> 2) + (x & 3)`，但显然不是正解~~。最重要是判断符号位。

用位运算来模拟条件语句，排除掉下溢出的情况，上溢出和减法为为“真”时满足返回条件。

```
int ry = ~y;
int sub = x + ry;   // x+ y-
int min_overflow = x & ry;   // x- y+
int max_overflow = x ^ ry;   // x- y- or x+ y+
int sign = (min_overflow | (sub & max_overflow)) >> 31;
return !sign;
```

logicalShift

第一思路就是构造符号位的mask，抵消填充符号位带来的影响：

```
int msk_sign = ((x >> 31) << (~n)) << 1;
return msk_sign ^ (x >> n);
```

这里 `n == 31` 左移 `32` 的情况需要拆分成左移 `31+1` 位。考虑优化：

satMut2

判断是否溢出就是判断符号位和第31位是否相同。Vanilla版本：

```
volatile int double_x = x << 1;
volatile int over = (double_x ^ x) >> 31;

volatile int tmp_over = ~over;
volatile int tmax = ~(1 << 31);
volatile int tmp_sign = x >> 31;
volatile int tmp_sum = tmax ^ tmp_sign;

volatile int _double_x = tmp_over & double_x;
// overflow: 0  else: double_x
volatile int _over = over & tmp_sum;
// overflow: 0x7FFFFFFF(+)  0x80000000(-)  else: 0
return _double_x | _over;
```

~~直觉来说是可以优化的，每个式子展开。~~然而直接展开很难能优化，因为与 isGreater 有异曲同工之妙，利用位运算模拟条件语句。

```
volatile int tmp_min = 1 << 31;
volatile int tmp_sign = double_x >> 31;
volatile int tmp_sum = tmp_min + tmp_sign;
```

神奇的事情是加了 `volatile` 依然出现本地和服务器结果不同，原因是一些中间变量仍然是没有被 `volatile` 修饰的。

subOK

一开始觉得和 isGreater 很像：$Tmin <= x - y <= Tmax \implies 0 <= x - y + Tmax + 1$ 但是发现不是那么好做。列出 x 和 y 的真值表：

x	y	safe	overflow
+	+	?	不会溢出
+	-	+	-
-	+	-	+
-	-	?	不会溢出

可以找到其中的规律。

```c
int ry = ~y;
int res = x + ry + 1;
int sign = ((x ^ y) & (x ^ res)) >> 31;
return !sign;
```

trueThreeFourths

损失量 `f(two) → min` 的主要规律是：

1. 正数：`f_3(3)=2`，`f_3(2)=1`，`f_3(1)=0`，`f_3(0)=0`
2. 负数：`f_0(3)=3`，`f_0(2)=2`，`f_0(1)=1`，`f_0(0)=0`

直接构造：

```c
int quarter = x >> 2;
int sign = x >> 31;
int two = (x & 0x3);
int min = two + ((sign & 1) | !two) + ~0;
return (quarter << 1) + quarter + min;
```

但是 op 数比较多，而且很难优化。换一个思路来想，考虑公式 $x * 3/4 == quarter * 3/4 + two * 3/4$ 其中 $two * 3/4$ 不会溢出可以先乘后除：`int min = (two + two + two + _two) >> 2;`

isPower2

判断是否只有一个 1 且不是 Tmin。一个有趣的性质：$(\cdots 100000)_2 - 1 = (\cdots 011111)$ 即从右向左数第一个 1 开始，右边的位全部取反。如果只有一个 1 那么没有交集，若有多个则有交集。

```c
int intersection = x & (x + (~0));
return !(intersection | !(x ^ (1 << 31)));
```

类似 isTmax，判断不是 Tmin 除了直接比较还可以强制右移变成 0：`!(x << 1)`

float_i2f

这道题直接模拟整型转浮点型的步骤就行，需要精通整套步骤包括 IEEE754、四舍六入五成双、负浮点数的储存、规范化等。

首先要模拟 normalized 的步骤。尝试了两种方法，用 while 循环逐个模拟进位，以及用 howManyBits 的方法。但不知道为什么后一种方法 op 数更多）。~~竟然直接禁掉了 `goto` ……~~

其次还有一些细节要处理：

- IEEE754标准的负数储存中，mantissa 部分存绝对值，最高位存符号。顺带一提，exponent 的存储方式类似“补码”，但是其“符号位”与一般有符号数的“符号位”相反。
- 而进位以 `((mantissa & 0x100U) && (mantissa & 0x2FFU))` 为条件，需要同时满足“五”和“六入或成双”（不能将两个条件合并）。

```c
unsigned sign = 0;
unsigned exponent = 159;
unsigned mantissa = x;
if (x < 0) {
    sign = 0x80000000U;
    mantissa = -x;
}
while (exponent) {
    unsigned tmp = mantissa;
    mantissa <<= 1;
    exponent--;
    if (tmp >= 0x80000000U) {
        break;
    }
}
return sign + (exponent << 23) + (mantissa >> 9) + ((mantissa &
0x100U) && (mantissa & 0x2FFU));
```

howManyBits

一开始以为是 fitsBits 的翻版，~~结果看到 max ops = 90……~~ 一步一步下手。首先排除符号位影响：`int _x = (x >> 31) ^ x;` ~~然后获得最高位的 1 ~~然后用二分法计算最高位的 1 的位置（需要注意 0 的情况）。

```c
int pos16 = (!!(_x >> 16)) << 4;
_x >>= pos16;
int pos8 = (!!(_x >> 8)) << 3;
_x >>= pos8;
int pos4 = (!!(_x >> 4)) << 2;
_x >>= pos4;
int pos2 = (!!(_x >> 2)) << 1;
_x >>= pos2;
int pos1 = !!(_x >> 1);  // _x >> 1;
_x >>= pos1;
return pos16 + pos8 + pos4 + pos2 + pos1 + 1 + _x;
```

一些失败（op 数更多）的优化尝试：

```c
...
_x >>= pos2;
_x += !(~_x);
return pos16 + pos8 + pos4 + pos2 + pos1 + _x;
```

float_half

分两种情况，exponent 除二以及 mantisa 除二。

```c
unsigned exponents = 0x7f800000U;
unsigned div_man = 0x00800000U;
unsigned _sign = 0x80000000U & uf;
unsigned uf_exp = uf & exponents;
if (uf_exp > div_man) {
    return uf - div_man * (uf_exp != exponents);
} else {
    unsigned div_exp = (uf + ((uf & 3) == 3) ^ _sign) >> 1;
    return _sign | div_exp;
}
```