CS: APP
DataLab Report

信息学院 张晋恺

Oct 2 2023

目录

1 实验部分

1.1 bitXor

1.1.1 实现代码

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

1.1.2 实现思路

$$
\begin{aligned}
& x \otimes y \\
\Leftrightarrow\ & (\neg x \wedge y) \vee (x \wedge \neg y) \\
\Leftrightarrow\ & ((\neg x \wedge y) \vee x) \wedge ((\neg x \wedge y) \vee \neg y) \\
\Leftrightarrow\ & ((x \vee \neg x) \wedge (x \vee y)) \wedge ((\neg x \vee \neg y) \wedge (y \vee \neg y)) \\
\Leftrightarrow\ & (x \vee y) \wedge (\neg x \vee \neg y) \\
\Leftrightarrow\ & \neg\neg(x \vee y) \wedge \neg\neg(\neg x \vee \neg y) \\
\Leftrightarrow\ & \neg(\neg x \wedge \neg y) \wedge (\neg(x \wedge y))
\end{aligned}
$$

即`~((~x) & (~y)) & (~(x & y))`

使用了逻辑推理中的分配律和德摩根律.

1.2 thirdBits

1.2.1 实现代码

```
int thirdBits(void) {
    int bits = 0x49;
    bits = (bits << 9) + bits;
    bits = (bits << 18 ) + bits;
    return bits;
}
```

1.2.2 实现思路

由于要尽可能少地使用操作符，并且题目中限制使用的自定义数字大小不超过0xFF 因此构造可满足的最大的基础数字，即01001001(binary)，转换成十六进制就是0x49.

该数字中为 1 的最高位是第 3 次“每 3 位设置成 1（从最低位开始）”的位置，因此应将其左移 $3 \times 3 = 9$ 位，并加上原数字.

得到的结果为1001001001001001(binary)，同理，接下来应当左移 $3 \times 6 = 18$ 位，得到的结果就是所求的数字

1.3 fitsShort

1.3.1 实现代码

```
1    int fitsShort(int x) {
2        x >>= 15;
3        x = x ^ (x >> 16);
4        return !x;
5    }
```

1.3.2 实现思路

short类型数字是 2 字节 16 位的，因此它能表示的有符号数字的区间是-32768～32767，在计算机机内的以int类型的表示码为 0x00000000～0x00007FFF 以及 0xFFFF0000～0xFFFFFFFF，即将这个数字的符号位后面的 16 位均与符号位相同，可以理解为由short类型数符号扩展到int 类型的数据.

具体实现为将输入数字左移 15 位得到的数字与将输入数字左移 31 位得到的数字作异或操作，若得到的结果为零则说明 16～31 位均与符号位相同（由于符号扩展，左移 31 位后 2～32 位均与符号位相同），若得到的结果不为零，则说明 16～31 位有某一位与符号位不同，即不满足题面条件，由于输入可截断的数字返回值是 1，所以最后进行了逻辑非运算，这样就可以使最后得到的返回值是题目所要求的

1.4 isTmax

1.4.1 实现代码

```
int isTmax(int x) {
    int y = x + 1;//Tmax + 1 = Tmin
    int z = y + y;//Tmin + Tmin = 0x100000000 = 0
    //if x == -1 = 0xffffffff, x + 1 = 0 then !y == 1
    //but if x != -1, x + 1 != 0 then !y == 0
    return !(z + (!y));
}
```

1.4.2 实现思路

`Tmax = 0x7fffffff`，而`Tmax + 1 = Tmin = 0x80000000`，

`Tmax + Tmax = 0`，可以利用这一特性来判断一个数是否为最大的整型数。但是-1(0xFFFFFFFF) 也符合该式，因此需要排除这一情况，所用方法是对 x 加 1，若`x == Tmax`，那么 x + 1 = 0x80000000，若`x == -1`，则 x + 1 = 0x100000000 = 0(最高位发生溢出)，对其进行非操作时`Tmax`和`-1`就会产生不同的结果，如上述代码所示，若`z = y + y = 0`且`!y = 0`，就说明`x == Tmax`，若`y + y = 0`且 `!y = 1`就说明`x = -1`，不符合题意要求，因此针对这一结果将`z + !y`取非即可将结果转化为所求结果.

1.5 fitsBits

1.5.1 实现代码

```
int fitsBits(int x, int n) {
    int shift = n + 31;//if t >= 32, n >> t == n >> (t % 32)
    int sign = x >> 31;
    int y = (x >> shift) ^ sign;
    return !y;
}
```

1.5.2 实现思路

与`fitsShort()`函数相似，若一个 32 位整型数字可以被截断成一个 n 位数字且值不发生改变，需要这个 32 位整型数字的第 n 位 (截断后的数字的符号位) 到第 32 位全相等，即由一个 n 位有符号整型数字符号扩展到 32 位，那么就可以先将这个数字右移 n-1 位得到符号位及它后边的所有位的值并将这个数字与原数字右移 31 位得到的全为符号位的数字进行异或操作，并取反得到最终要求的返回值。由于操作符限制不能使用减法运算，我们需

要使用其他方法获取 n-1，常规方式是将 0 取反得到 0xFFFFFFFF=-1，再加上 n 就得到了 n-1，但为了操作符数量尽可能少 (主要是为了 95 分)，我在函数中使用了一个未定义行为，即 32 位整型数字右移超过 32 位会进行右移这个数字对 32 取模的移位运算，即`n >> 32+t == n >> t`, 因此，要想右移 n-1 位，只需要进行`x >> 32 + n - 1`操作即可，较`x >> n + (~0)`操作少一个操作符，这样就可以达到 95 分的 5 个操作符限制.

1.6 upperBits

1.6.1 实现代码

```
int upperBits(int n) {
    int x = !(!n) << 31;
    int y = x >> (n + 31);
    return y;
}
```

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

1.6.2 实现思路

当`n != 0`只需将 1 左移 31 位得到 0x80000000 再右移 n 位即可，而考虑到`n = 0`的情况，需要根据 n 的值将移位后的最高位设为 0 或 1，而将 n 两次取非操作即可得到所求的效果.

当`x != 0`的时候和 32 按位与总能得到 0x20，左移 26 位把这个 1 移到第 32 位，再右移 n-1 位即可；而当 x = 0 的时候得到的是 0.

1.7 anyOddBit

1.7.1 实现代码

```
int anyOddBit(int x) {
    int mask = 0xAA;
    mask = (mask << 8) + mask;
    mask = (mask << 16) + mask;
    int isAnyOddBit = x & mask;
```

```
6         return !(!isAnyOddBit);
7     }
```

1.7.2 实现思路

判断一个int类型整数的是否有任意奇数位为 1，构造奇数位全为 1 的掩码mask = 0xAA，再将其左移 8 位加上原数，再左移 16 位加上原数就构造出了0xAAAAAAAA，再将 mask 与 x 进行与操作，若 x 的奇数位全为 0，得到的isAnntOddBit == 0，那么!(!isAnyOddBit) = 0，若存在某一个奇数位为 1，则isAnyOddBit != 0，有!(!isAnyOddBit) = 1，即为所求.

1.8 byteSwap

1.8.1 实现代码

```
1     int byteSwap(int x, int n, int m) {
2         n = n << 3;//n * 8
3         m = m << 3;//m * 8
4         int temp = ((x >> n) ^ (x >> m)) & 0xFF;
5         x = x ^ ((temp << m) | (temp << n));
6         return x;
7     }
```

1.8.2 实现思路

先将字节数转化为位，即字节数 *8，用位运算则表示为左移三位，注意到a ^ (a ^ b) = b与 b ^ (a ^ b) = a。

因此构造temp = ((x >> n) ^ (x >> m)) & 0xFF，这就构造出了 x 的 n 位与 m 位异或后的结果，并且与 0xFF 作与运算清除掉了所有高位，最后再将 temp 分别左移 m 位与 n 位构造出在需要交换的字节处为(a ^ b)这样的结果，将 x 与该数异或即为所求.

1.9 absVal

1.9.1 实现代码

```
1     int absVal(int x) {
2         int t = x >> 31;
3         x = (x + t) ^ t;
```

```
        return x;
    }
```

1.9.2 实现思路

正数的绝对值是其本身，负数的绝对值是将其减一按位取反再得到，对于正数，`x >> 31 = 0`，对于负数`x >> 31 = 0xFFFFFFFF = -1`，因此可以根据这一性质，当 x 为正数时，`x + t = x`，`(x + t) ^ t = x`，当 x 为负数时，`x + t = x - 1`，`(x + t) ^ t`刚好是按位取反操作，最终得到的结果就是任意数的绝对值.

1.10 divpwr2

1.10.1 实现代码

```
    int divpwr2(int x, int n) {
        int mask = x >> 31;
        //int shift = ((1 << n) + mask) & mask;
        int shift = (mask << n) ^ mask;
        x = x + shift;
        x = x >> n;
        return x;
    }
```

1.10.2 实现思路

当 x 为正数时，x 除以 2^n 即为`x >> n`，当 x 为负数时，x 除以 2^n 做右移操作会发现得到的结果是向 $-\infty$ 取整，而题目要求的是向 0 取整，注意到只需将 x 加上 $2^n - 1$ 再做右移运算所求出的结果就是向 0 取整的结果，因此需要构造出 $2^n - 1$，又注意到当 x 为负数时符号位为 1，x 为正数时符号位为 0，因此可以构造`mask = x >> 31`，再进行`(mask << n) ^ mask`就可以构造出这一偏移量，并且当 x 为正数时这一偏移量为 0，将 x 加上这一偏移量在进行右移操作即可.

1.11 float_neg

1.11.1 实现代码

```
unsigned float_neg(unsigned uf) {
    unsigned nan = 0xFF000000;
    unsigned E = uf << 1;
    return uf ^ ((E <= nan) << 31);
}
```

1.11.2 实现思路

判断 uf 的阶码是否为 nan 再返回原数或符号位取反即可，构造 `nan = 0xFF000000`，将 uf 左移一位与 nan 进行大小比较即可判断 uf 是否合法，将比较得到的 bool 值左移 31 位与 uf 异或即可得到合法浮点数的相反数或 nan.

1.12 logicalNeg

1.12.1 实现代码

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

1.12.2 实现思路

当 $x \neq 0$ 时，`(~x + 1)` x| 的符号位必为 1，右移 31 位后为-1；当 $x = 0$ 时，`(~x + 1)` x| 的 0，右移 31 位仍然为 0，根据 的定义，将得到的数字加 1 即为所求.

1.13 bitMask

1.13.1 实现代码

```
int bitMask(int highbit, int lowbit) {
    int allBits1 = ~0;
    //int two = 0x2;
    //int high = allBits1 << highbit << 1;
    int high = (0x2 << highbit) + allBits1;
    return high >> lowbit << lowbit;
    //int low = allBits1 << lowbit;
    //int mask = high ^ low;
    //int mask = allBits1 << (highbit ^ lowbit);
```

```
        //return mask & low;
        //return ~high & low;
    }
```

1.13.2 实现思路

将给定位区间设置成一，先构造一个全为 1 的数 0，由于给定的 highbit 实际在移位操作时需要加一，所以使用 0x2 即 10 进行左移操作，再加上 allBits1 即可将 highbit 以后高位置 0，再右移 lowbit 左移 lowbit 即可将 lowbit 之后的低位置 0.

1.14 isGreater

1.14.1 实现代码

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

1.14.2 实现思路

溢出的情况仅在 x, y 异号时出现，因此使用`(x ^ y) >> 31`判断符号位是否不同，与 y 进行按位或操作，当 x, y 同号时结果为 y，当 x, y 异号时结果为 0xFFFFFFFF。又注意到 $x = -(x + 1)$，因此，当 x，y 同号时代码中结果为 $x - y - 1$，当该式大于 0 时即为 x > y，当 x，y 异号时代码中结果为 x 的值，只需判断 x 的正负即可得出 x，y 的大小关系.

1.15 logicalShift

1.15.1 实现代码

```
    int logicalShift(int x, int n) {
        //return (x >> n) & ~(1 << 31 >> n << 1);
        int mask = 1 << (31 ^ n);//31 ^ n = 31 - n
        return ((x >> n) + mask) ^ mask;
        // printf("%x\n%x\n",~zero, ~zero >> (n + 32));
        // printf("%x\n",(x >> n) & ((~zero) >> n << 1));
        //return (x >> n) & ((~zero) >> (n + 32));
```

```
        // return (x >> n) & ((~zero) >> (n));
    }
```

1.15.2 实现思路

只需考虑 x 为负数时的右移情况，构造第 31-n 位为 1 的掩码，当 x 为负数时，`(x >> n) + mask`使前 n 位置 0，再与 mask 异或使符号位复位；当 x 为正数时，`(x >> n) + mask`使第 32-n 位置 1，再与 mask 异或使其归零，最终得到的值即为所求逻辑右移.

1.16 satMul2

1.16.1 实现代码

```
    int satMul2(int x) {
        int xMul2 = x << 1;
        int reset = xMul2 >> 1;
        int xor = x ^ reset;
        //if x * 2 not overflow
        //32nd bit and 31st bit of x is same
        int judgeOverflow = xor >> 31;
        return (xMul2 >> judgeOverflow) ^ xor;
        //if overflow : judgeOverflow = 0xFFFFFFFF = -1
        //else : judgeOverflow = 0x0 = 0
        //if xMul2 valid return xMul2 | judgeOverflow = xMul2
    xMul2 & ~judgeOverflow = xMul2
        //if xMul2 overflow xMul2 & ~judgeOverflow = 0
        //  if x >> 31 == 0xFFFFFFFF(xMul2 is negitive) : return
    Tmin = 0x80000000
        //  if x >> 31 == 0(xMul2 is positivve) : return Tmax = 0
    x7FFFFFFF
        //overflow : xMul2 & ~judgeOverflow = 0 x < 0 xMul2 = Tmin
     = xMul2 >> 31 ^ Tmin
        //int Tmin = 1 << 31;
        //return (xMul2 & ~judgeOverflow) | (judgeOverflow & ((
    xMul2 >> 31) ^ Tmin));
    }
```

1.16.2 实现思路

当 $x \times 2$ 不会溢出时，x 的符号位和它的次高位相同，此时直接进行左移一位的操作即可. 因此，将 x 左移一位再右移一位，与原数异或再右移 31 位就可以判断出 x 是否会溢出，若溢出则 $judgeOverflow = 0xFFFFFFFF$，若不溢出则 $judgeOverflow = 0$，注意到 `x >> 0xFFFFFFFF`等效于`x >> 31`，利用这一性质可以将溢出的数据置为 Tmax 或 Tmin.

1.17 subOK

1.17.1 实现代码

```
int subOK(int x, int y) {
    int result = x + ~y + 1;
    int sign_diff = x ^ y;
    int isOverflow = result ^ x;
    return ((sign_diff & isOverflow) >> 31) + 1;;
    //int result = x + (~y + 1);
    //int t = (x ^ y) & (x ^ result);
    // int t = x ^ (y & result);
    // return !(t >> 31);
}
```

1.17.2 实现思路

只有当 x，y 异号时才可能出现溢出，先计算 $x - y$ 的值，即 x + ~y + 1，若 $x - y$ 的值与 x 的符号位相同则不会发生溢出，若符号位不同则发生溢出，用 sign_diff 判断符号位是否相同，用 isOverflow 与 sign_diff 按位与并右移 31 位即可得到是否会发生溢出，若溢出得到的值为-1，若不溢出得到的值为 0，加一即为所求.

1.18 trueThreeFourths

1.18.1 实现代码

```
int trueThreeFourths(int x) {
    int mask = ~(x >> 31) & 3;
    int xDiv4 = ((x + mask) >> 2) & (~(mask << 30));
    return x + ~xDiv4 + 1;
}
```

1.18.2 实现思路

$x \times \frac{3}{4} = x - \frac{1}{4}x$，构造 $\frac{1}{4}x$，并且需要向下取整。当 x 为负数时，x 右移两位即为所求，当 x 为正数时，需要计算 $\frac{x+3}{4}$，因此构造掩码以便于负数不变正数加三的操作，构造`mask = ~(x >> 31) & 3`，然后再加上 x 右移两位，由于 x 为正数时，$x + 3$ 可能会产生溢出，因此需要与 0x3FFFFFFF 进行与操作，注意到当 x 为正数时 mask 恰好为 0x3，左移 30 位取反即为 0x3FFFFFFF. 最后`x + ~xDiv4 + 1`即可.

1.19 isPower2

1.19.1 实现代码

```
1    int isPower2(int x) {
2        int y = x + ~0;
3        return !((x & y) | (y >> 30));
4    }
```

1.19.2 实现思路

当 $x = 2^n$ 或 $x = 0$ 或 $x = 0x80000000$ 时，`x & x-1 == 0`，为了排除这两个特殊数，使用 x-1 右移 30 位判断，当 x 为这两个特殊值时该操作得到的数不为 0，与 x & x-1 进行或操作即可排除，最后取非即为所求.

1.20 float_i2f

1.20.1 实现代码

```
unsigned float_i2f(int x) {
    unsigned ux = x;
    unsigned signE = 0x4E800000;
    int flag = 0;
    if(x)
    {
        while(1)
        {
            if(ux & 0x80000000)
            {
                if(flag)
                break;
                else
                {
                    ux = -x;
                    signE = 0xCE800000;
                }
            }
            else
            {
                ux <<= 1;
                signE -= 0x800000;
            }
            flag = 1;
        }
        if(ux & 0x17F)
            ux += 0x80;
        ux >>= 8;
        return signE + (ux ? ux : 0x01000000);
    }
    else
        return 0;
}
```

1.20.2 实现思路

当 abs_x 可以被 24 位无符号整数表示时，x 可以被精确地表示为 float 浮点数类型，构造阶码为 30 的初始值（由于浮点数的有效数字位隐藏了小数点前的 1）signE=0x4E800000，将 x 赋值给 ux，防止移位操作产生影响，由于浮点数的有效数字位使用 x 的绝对值表示，所以当 x<0 时，将 ux 设置为-x，并将 signE 的符号位设为 1，并进入下一次循环求阶码及有效数字位；当 x>0 时，由于有效数字位有隐藏 1 的存在，所以将 ux 右移 1 位使其值变为 2 倍并将阶码的值减 1 直至 ux 最高位的 1（即隐藏 1 超过第 32 位）移至第 32 位，退出循环，判断移位后的 ux 的末 7 位是否有 1，判断 rounding 进位，最后将 ux 右移 8 位设置为浮点数的有效数字位，与 signE 相加即为所求整数强制转化为浮点数的二进制表示.

1.21 howManyBits

1.21.1 实现代码

```
1   int howManyBits(int x) {
2       // int mask = x >> 31;
3       // int x = x ^ mask;
4       int shift16, shift8, shift4, shift2, shift1;
5       x = x ^ (x << 1);
6       shift16 = (!!(x >> 16)) << 4;
7       x = x >> shift16;
8       shift8 = (!!(x >> 8)) << 3;
9       x = x >> shift8;
10      shift4 = (!!(x >> 4)) << 2;
11      x = x >> shift4;
12      shift2 = (!!(x >> 2)) << 1;
13      x = x >> shift2;
14      shift1 = (!!(x >> 1)) & 1;
15      return shift16 + shift8 + shift4 + shift2 + shift1 + 1;
16      }
```

1.21.2 实现思路

正数找到为 1 的最高位，负数找到为 0 的最高位，位数加 1 即为最小的能表示出这个数字的位数，计算位数采用二分法的思路.

1.22 float_half

1.22.1 实现代码

```
unsigned float_half(unsigned uf) {
    unsigned sign = uf & 0x80000000;
    unsigned abs_uf = uf - sign;
    int shift, add;
    if(abs_uf >= 0x7F800000)//nan
    {
        shift = 0;
        add = 0;
    }
    else if(abs_uf >= 0x900000)//normalization
    {
        shift = 0;
        add = -0x800000;
    }
    else//not normalization
    {
        shift = 1;
        add = (abs_uf & 3) == 3;
    }
    return sign + (abs_uf >> shift) + add;
}
```

1.22.2 实现思路

考虑三种情况，nan，除 2 后为非规格化浮点数，正常数. 先计算出符号位，uf&0x80000000 即可，再计算 uf 的绝对值，使用 uf-sign 即为所求，若 abs_uf >= 0x7F800000 则表示 uf 的阶码为 0xFF，即 nan；若 abs_uf >= 0x900000，则表示 uf 除以 2 后规格化浮点数，其他情况则表示 uf 除以 2 为非规格化浮点数. 若 uf 为 nan，直接返回；若 uf 除以 2 后为规格化浮点数，只需将阶码减一即可；若 uf 除以 2 后为非规格化浮点数，则需要将 abs_uf 右移一位并加上可能产生的进位（当小数部分最后两位为 1 时）. 使用 if 条件分支语句区分三种情况.

2 总结

DataLab 解决了大家卷度不够的问题，DataLab 包括一系列的编程任务和挑战，学生需要完成这些任务，以练习和巩固他们在课程中学到的关于计算机底层操作的概念；DataLab 帮助学生将理论知识与实际的编程练习相结合，以加深他们对计算机系统工作原理的理解.

3 致谢

感谢各位卷卷卷卷卷王给我带来嗯卷 DataLab 的动（疲）力（惫）

4 吐槽

别卷了别卷了，要卷死了