ICS Lab1-Datalab 实验报告

—— 邹懿 21302010073

实验结果

./dlc -e bits.c 运行截图

```
zou@LAPTOP-7S0OF19G:~/datalab-handoutzy$ ./dlc -e bits.c
dlc:bits.c:151:bitNor: 3 operators
dlc:bits.c:163:tmax: 2 operators
dlc:bits.c:176:isTmin: 8 operators
dlc:bits.c:188:minusOne: 1 operators
dlc:bits.c:205:absVal: 4 operators
dlc:bits.c:220:leastBitPos: 4 operators
dlc:bits.c:247:byteSwap: 14 operators
dlc:bits.c:266:logicalShift: 9 operators
dlc:bits.c:286:isLessOrEqual: 20 operators
dlc:bits.c:305:multFiveEighths: 8 operators
dlc:bits.c:332:bitCount: 37 operators
dlc:bits.c:355:greatestBitPos: 40 operators
dlc:bits.c:368:bang: 5 operators
dlc:bits.c:395:bitReverse: 40 operators
dlc:bits.c:426:mod3: 55 operators
dlc:bits.c:447:float_neg: 7 operators
dlc:bits.c:489:float_i2f: 29 operators
dlc:bits.c:514:float_twice: 16 operators
```

./btest 运行截图

```
zou@LAPTOP-7SOOF19G:~/datalab-handoutzy$ ./btest
Score   Rating  Errors  Function
 1      1       0       bitNor
 1      1       0       tmax
 1      1       0       isTmin
 1      1       0       minusOne
 2      2       0       absVal
 2      2       0       leastBitPos
 2      2       0       byteSwap
 3      3       0       logicalShift
 3      3       0       isLessOrEqual
 3      3       0       multFiveEighths
 4      4       0       bitCount
 4      4       0       greatestBitPos
 4      4       0       bang
 4      4       0       bitReverse
 4      4       0       mod3
 2      2       0       float_neg
 4      4       0       float_i2f
 4      4       0       float_twice
Total points: 49/49
```

实验思路

P1 bitNor

不难发现~(x|y)中只有两个0遇到才会取1，那就联想到&——只有两个1遇到才会取1，但同时需要取反才能满足情况，因此就等于(~x)&(~y)

P2 tmax

二进制的最大补码数是01...11(31个1)，我们让1左移31位便得到10...00(31个0)，再取反则得到我们想要的数字，即~(1<<31)

P3 isTmin

这里主要利用**二进制最小补码数取反加一后还是自身**这一特点，前半部分!(x^(~x+1))判断是否取反加一还是与自身相同，相同则为1，但需要排除0(~0+1还是0)，后半部分(~(!x)+1)),就是让x为0则前半部分减去1，x不等于0则值不变从而排除0这一特殊情况

P4 minusOne

-1在二进制中表示为11...11(32个1)，因此直接对0取反便能得到

P5 absVal

我首先是考虑到**1异或0or1相当于对其取反，而0异或0or1值不变**，我们就利用正负数的符号位不同，将其右移31位得到全1（负数）或者全0（非负数），再和原来的数取异或则非负数不变，负数取反，然后再用(1&(1<<31))区分是否加1,从而获得其绝对值

P6 leastBitPos

这里主要是考虑消掉最低位1前面的1，则将该数加-1（1111....111),能够将最低位1变为0，0变为1或者0，最低位前面的1变为1，然后我想到用类似这样的形式x&(*)保证原来的0不变，再用((x+~0)^x)使得最低位前面的1变为0，最低位1不变，则((x+~0)^x)&x即为所求

P7 byteSwap

这道题我主要利用的还是异或的性质——相同数异或为0，0异或0 or 1可以保存原来的值，然后我将nth byte和mth byte的值都搬到了0th byte上去(等价于0000...00nth byte 0000...00mth byte)并分别赋给不同变量temp1,temp2，然后再将temp1左移8n位与原来的数异或则将原来的nth byte的位置全部变为0，temp2同理，然后再将temp1左移8m位和原来的数异或从而将nth byte放在了mth位置，temp2同理，从而实现byte的交换

P8 logicalShift

这一题主要还是需要区分正负数，我们同样可以利用其符号位不同，temp=x>>31用来存储数字特征，因为负数算术右移高位补1我们需要将补的n个1变为0，则让temp<<(31+(~n+1)),temp<<1（避免n等于0）,实现temp左移(32-n)位，因此如果为非负数那么temp全0左移不变，如果为负数那么temp则为11..11(n个1)000..00(32-n个0)，再对temp取反与原来的数取&则能得到

P9 isLessOrEqual

考虑溢出等，判断x<=y我们不能只是根据(x-y)的符号，需要分三种情况，函数中用sign, sign_x, sign_y, sign_yx, sign_eq来分别判断xy符号是否相同、x符号、y符号、(x-y)符号、以及xy是否相等。

第一种若sign为1(异号)并且sign_x为1(负数)；

第二种！sign为1（同号）并且sign_yx为1(x-y<=0);

第三种主要是因为**二进制最小数取反加一后存在溢出仍是数本身，因此当两个数均为二进制最小数无法通过第二种判断**，需要单独判断两个数字是否相等

P10 multiFiveEighths

对于(x*5/8)来说，数字如果先乘5则容易超过范围，因此我们先让x/8，相当于右移3位，然后乘5，但**在这其中我们丢失了最后3个bit表示的数**，因此我用tail = x&0x7保存，tail部分先乘5再/8，将两个部分的结果相加即可

P11 bitCount

以2个bit为单位考虑

⇒ number =
- 0　00 ⇔ 0+0
- 1　01 / 10 ⇔ 0+1 / 1+0
- 2　11 ⇔ 1+1

则当我们把相邻bit的值相加后，2个bit存储的便是1的个数 {00, 01, 10}

用x&mask_1保存偶数位bit存储数值，(x>>1)&mask_1保存奇数位，相加后使得2个bit保存之前这2个bit上1个的个数

以4个bit为单位考虑

⇒ number =
- 0　$\underbrace{0000}_{0+0}$
- 1　$\underbrace{0001}_{0+1}$ / $\underbrace{0100}_{1+0}$
- 2　$\underbrace{0101}_{1+1}$ / $\underbrace{1000}_{2+0}$ / $\underbrace{0010}_{0+2}$
- 3　$\underbrace{0110}_{1+2}$ / $\underbrace{1001}_{2+1}$
- 4　$\underbrace{1010}_{2+2}$

同上操作，用 x&mask_2 + (x>>2)&mask_2 可使得4个bit保存1个数

重复此最终32个bit的x的值便是1的个数

P12 greatestBitPos

这里采用的是分半观察，我们先用 pos += ((!!(x&(~0<<16))) << 4)判断x前16位是否有1，若有则pos为16接下来则对前16bit观察，若无则pos为0接下来则对后16bit观察；紧接着我们再用pos += ((!!(x&(~0<<(pos+8)))) << 3)来判断16bit中1在前8bit还是后面；重复的下去最后我们便能获得最高位1的具体位置pos，最后返回1<<pos,同时考虑特殊情况0则应该return (1<<pos) & x;

P13 bang

这里是区分0和非0，则考虑**0和其相反数符号位相同**的特征，用（x | (~x+1))>>31反映相反数之间符号位是否相同，若相同则为0，不同则为-1，再加1则可以得到!x

p14 bitReverse

思路与P11类似，我们首先通过x&mask_1保存奇数位的信息后再往左移1位将偶数位移到奇数位，(x>>1)&mask_1将奇数位移到偶数位同时保存信息，从而实现每2个Bit的reverse,同理可以依次实现4、8和16个bit的reverse，从而实现bitReverse

P15 mod3

因为相反数mod3的余数也互为相反数，因此我们首先对x取绝对值。再根据2进制数对3取余特点,则前16bit的数表示的数mod3加上后16bit表示的数mod3等于32bitmod3，因此我们可以将前16bit加上后16bit,用16bit存储和原值同余的数，同时我们还需要判断相加是否有溢出，即第17bit为1还是0，是1则将其右移16位保存；同理一直重复下去，最后使得末尾2个bit保存余数信息，这其中包括00 01 10 11四种情况，其中(11)mod3余数仍然为0，我们需要让前三种不变，最后一种变为0（函数体中最后三行执行功能），从而获得|x|mod3，最后再判断返回mod还是（-mod)

P16 float_neg

根据ieee规则，我们实际上只需要将最高位取反即可，但需要考虑NAN情况。若fraction部分非0并且exp等于255则返回原值；否则利用1的异或性质将x^0x80000000(100..00)即可改变符号位

p17 float_i2f

t函数中temp记录x的最高有效位位置，exp考虑bias等于temp+126记录指数值

这里我们需要考虑三种情况：

第一种是特殊值——0和0x80000000,因为规格化数不能表示0因此我们直接return 0,同时因为0x80000000对应的相反数超过int表示范围我们直接return 0xcf000000;

第二种temp小于等于24，由于fraction部分等价于1.frac相当于24位，则此时frac足够存储;

第三种temp>24,我们只能存储从x有效位起24个bit的值，因此frac = (x>>(temp-24))&frac_mask, 同时我们需要判断被丢弃的是否需要进位，因为我们需要保证temp=32时丢弃的8个Bit都包含在判断范围，因此我们将被丢弃的数字与8bit表示的中间数0x80比较，若大于0x80或者等于0x80并且frac最后一位为1则frac进位,同时如果frac溢出则需要将exp++同时保留frac后23位

P18 float_twice

sign存储符号位，exp存储指数的值，则需要讨论以下几种情况：若exp为0，则为非规格化数，则直接将fraction部分左移1，等价于(sign | uf<<1);若exp为255，则为无穷大或者NAN，直接返回原值即可；否则利用ieee浮点数表示法则，直接将exp++，则相当于乘2，但需要考虑exp若加1后为255则返回无穷大，否则便用uf = uf^((exp-1)<<23)将原来数字中exp的值变为0，然后用uf^(exp<<23)存储新值。

实验感受

- 近期不想看见0和1了（bushi
- 对位运算认识更深刻吧，感觉位运算很强大，可以实现很多基础运算(mod 3属实佩服)
- 写lab后期对位运算比较上头，以至于0x7我用了~(（~0）<<3)去表示。。。。