[读书笔记]CSAPP：DataLab

下载地址：

http://www.cs.cmu.edu/afs/cs/academic/class/15213-f15/www/labs/datalab-handout.tarwww.cs.cmu.edu/afs/cs/academic/class/15213-f15/www/labs/datalab-handout.tar

---

该实验主要考查同学对位级运算、无符号数编码、补码以及浮点数编码的掌握程度。**实验过程：**

1. 在`bits.c`中做题目
2. 使用`make clean`和`make`进行编译
3. 调用`./btest -f funcName`测试`funcName`函数的结果，可以在代码中中插入`printf`输出中间结果，但是要记得最后删掉
4. 调用`./dlc bits.c`查看是否使用了非法或过多的运算符
5. 重复以上步骤，最后可以直接运行`./btest`输出最后结果。

bitXor

```
/*
 * bitXor - x^y using only ~ and &
 *   Example: bitXor(4, 5) = 1
 *   Legal ops: ~ &
 *   Max ops: 14
 *   Rating: 1
 */
int bitXor(int x, int y)
{
  int notand = ~(x & y);
  int or = ~(~x&~y);
  return notand & or;
}
```

要求只能使用`^`和`&`实现异或运算符，通过查看`AND`、`OR`和`XOR`的真值表，我们可以发现`x^y=~(x&y)&(x|y)`，而我们这里缺少`|`运算，但是可以通过`~`和`&`构造出来，即`x|y=~(~x&~y)`。

AND			OR			XOR		
INPUT A	INPUT B	OUTPUT	INPUT A	INPUT B	OUTPUT	INPUT A	INPUT B	OUTPUT
TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE	TRUE
FALSE	TRUE	FALSE	FALSE	TRUE	TRUE	FALSE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE

tmin

```
/*
 * tmin - return minimum two's complement integer
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 4
 *   Rating: 1
 */
int tmin(void) {
  return 1<<31;
}
```

这个主要考查补码的定义，通过将最高有效位置1，就能得到补码中的最小值，而且程序是32位的，所以直接对1左移31位就行。

isTmax

```
/*
 * isTmax - returns 1 if x is the maximum, two's complement number,
 *     and 0 otherwise
 *   Legal ops: ! ~ & ^ | +
 *   Max ops: 10
 *   Rating: 2
 */
int isTmax(int x) {
  int a = x+1;
  return !((~a+1)^a)&(!!a); //加一溢出到TMin，并且-TMin==TMin，然后消除全1可能
}
```

同样考查补码的性质，我们知道补码中的最大值是最高有效位为0，其他为1，但是该题中限制了不允许使用移位操作，所以无法直接进行移位。首先，Tmax加一会正溢出得到Tmin，而Tmin一个特殊性质就是Tmin的相反数和Tmin相等，所以我们可以首先通过`a=x+1`，然后判断其相反数是否和自身相同，即`(~a+1)^a`，并且我们要消除全1的情况。

allOddBits

```
/*
 * allOddBits - return 1 if all odd-numbered bits in word set to 1
 *   Examples allOddBits(0xFFFFFFFD) = 0, allOddBits(0xAAAAAAAA) = 1
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 12
 *   Rating: 2
 */
int allOddBits(int x) {
  int a = x&(x>>2); //2 4
  int b = a&(a>>4); //2 4 6 8
  int c = b&(b>>8); //2 4 6 8 10 12 14 16
  int d = c&(c>>16);
  return (d>>1)&0x01;
}
```

该算法要求判断是否全部偶数位都为1。我们通过不断的移位操作，能够将所有偶数位的值都“与”到倒数第二低的有效位上，然后直接判断其值是否为1。

negate

```
/*
 * negate - return -x
 *   Example: negate(1) = -1.
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 5
 *   Rating: 2
 */
int negate(int x) {
  return ~x+1;
}
```

这题主要考察对补码的理解。我们补码对应的有符号数的运算可以直接在补码的位向量上进行计算，所以要保证相反数相加为0，表示需要使得两个位向量相加刚好低32位为0，所以直接对其取反加一，这样就能使得相加结果溢出1位，截断后就为0。

isAsciiDigit

```
/*
 * isAsciiDigit - return 1 if 0x30 <= x <= 0x39 (ASCII codes for characters '0' to '9')
 *   Example: isAsciiDigit(0x35) = 1.
 *            isAsciiDigit(0x3a) = 0.
 *            isAsciiDigit(0x05) = 0.
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 15
 *   Rating: 3
 */
int isAsciiDigit(int x) {
  //x-0x30>=0-->x+~0x30+1>=0-->!((x+~0x30+1)>>31)
  //x-0x39<=0-->x+~0x39+1<=0-->(x+~0x39+1)>>31
  int less = !((x+~0x30+1)>>31);
  int judge = x+~0x39+1;
  return less&((!judge)|(judge>>31));
}
```

这题主要考察如何实现大小比较。通过两数相减结果的正负来判断两个数之间的大小关系，但是这里不支持减法，但是`a-b`可以转化为`a+(-b)`，而`-b`就是`~b+1`，所以`a-b=a+(~b+1)`。然后我们可以判断该结果最高有效位是否为1，来判断结果的正负。

conditional

```
/*
 * conditional - same as x ? y : z
 *   Example: conditional(2,4,5) = 4
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 16
 *   Rating: 3
 */
int conditional(int x, int y, int z) {
  //!x: x is False-->zero-->0x01, x is True-->non-zero-->0x00
  //!x<<31>>31: 0x01-->0xFFFFFF, 0x00-->0x00
  int mask = !x<<31>>31;
  return (mask&z)+((~mask)&y);
}
```

要实现条件分支，有一个思路是如果将条件结果转化为全0或全1，由此乘到分支上进行筛选，即`(mask&z)+(~mask&y)`，如果mask为全1，则选择z，如果mask为全0，则为y。

但是这里要如何将条件转化为全0和全1呢？这里利用了补码右移时是使用算数右移的特点。这里条件`x`是`int`类型，表示使用补码编码， 首先使用`!`判断x是否为0，如果不是0，则最低有效位为1，我们通过左移31位将其置为最高有效位，然后右移时就能全部置为1。

isLessOrEqual

```
/*
 * isLessOrEqual - if x <= y  then return 1, else return 0
 *   Example: isLessOrEqual(4,5) = 1.
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 24
 *   Rating: 3
 */
int isLessOrEqual(int x, int y) {
  //不同符号相减容易出现溢出问题，所以先判断符号是否相同，只有相同时才进行相减
  int diff = (x>>31)^(y>>31);
  //diff ? !(y>>31) : y-x
  int part1 = x+~y+1;
  int part2 = !part1;
  int part3 = !!(part1>>31);

  int judge = part2 | part3;

  int ans = (!diff&judge)+(diff&(!(y>>31)));
  return ans;
}
```

这里也可以通过相减然后判断结果的正负来得到结果，但是要注意两个符号不同的补码相减容易造成溢出，使得结果比较难判断，而符号相同进行相减，就不会有溢出的问题了。我们首先可以通过两个数的最高有效位来判断符号是否相同，如果不同，可以很容易得到结果，如果相同，则需要相减来判断。

logicalNeg

```
/*
 * logicalNeg - implement the ! operator, using all of
 *              the legal operators except !
 *   Examples: logicalNeg(3) = 0, logicalNeg(0) = 1
 *   Legal ops: ~ & ^ | + << >>
 *   Max ops: 12
 *   Rating: 4
 */
int logicalNeg(int x) {
  //0的相反数还是0，其他的数字，要么自己，要么相反数最高位为1，就能由此判断
  return (~(x|(~x+1))>>31)&1;
}
```

0的相反数还是0，而其他数或者它的相反数一定有一个是负的，而负数算数右移31位能使得最低有效位置为1。

howManyBits

```
/* howManyBits - return the minimum number of bits required to represent x in
 *             two's complement
 *  Examples: howManyBits(12) = 5
 *            howManyBits(298) = 10
 *            howManyBits(-5) = 4
 *            howManyBits(0)  = 1
 *            howManyBits(-1) = 1
```

```
 *            howManyBits(0x80000000) = 32
 *  Legal ops: ! ~ & ^ | + << >>
 *  Max ops: 90
 *  Rating: 4
 */
int howManyBits(int x) {
  int b16,b8,b4,b2,b1,b0;
  int sign=x>>31;
  x = (sign&~x)|(~sign&x);//如果x为正则不变，否则按位取反（这样好找最高位为1的，原来是最高位为0的，这样也将符号位去掉了）
  // 不断缩小范围
  b16 = !!(x>>16)<<4;//高十六位是否有1
  x = x>>b16;//如果有（至少需要16位），则将原数右移16位
  b8 = !!(x>>8)<<3;//剩余位高8位是否有1
  x = x>>b8;//如果有（至少需要16+8=24位），则右移8位
  b4 = !!(x>>4)<<2;//同理
  x = x>>b4;
  b2 = !!(x>>2)<<1;
  x = x>>b2;
  b1 = !!(x>>1);
  x = x>>b1;
  b0 = x;
  return b16+b8+b4+b2+b1+b0+1;//+1表示加上符号位
}
```

网上抄的.....

float_twice

```
/*
 * float_twice - Return bit-level equivalent of expression 2*f for
 *   floating point argument f.
 *   Both the argument and result are passed as unsigned int's, but
 *   they are to be interpreted as the bit-level representation of
 *   single-precision floating point values.
 *   When argument is NaN, return argument
 *   Legal ops: Any integer/unsigned operations incl. ||, &&. also if, while
 *   Max ops: 30
 *   Rating: 4
 */
unsigned float_twice(unsigned uf) {
  //参数定义放在函数头
  unsigned exp = uf>>23;
  unsigned ans;
  //NaN and inf
  if(!((exp&0xFF)^0xFF)){
      return uf;
  }
  //non-normalized 直接左移
  if(!(exp&0xFF)){
      unsigned sign = uf>>31<<31;
      return (uf<<1)|sign;
  }
  //normalized 指数部分加一
  ans = (1<<23)+uf;
  //inf
  if(!(((ans>>23)&0xFF)^0xFF)){
      return ans>>23<<23;
  }else{
      return ans;
```

```
  }
}
```

这里主要考察对浮点数编码的理解。浮点数的编码分成三种情况：

- 规格化数：要求尾数表示为`1.f1,f2,...`，所以我们无法直接对尾数部分进行左移来乘2，但是我们可以直接对阶码部分加1来乘上2，需要判断阶码部分是否超出规格化数的表示范围，变成了无穷。
- 非规格化数，要求阶码全为0，所以无法直接对解码进行加1，但是尾数表示为`0.f1,f2,...`所以我们可以直接对尾数部分左移来乘上2。而在非规格化数的最大值（尾数部分全为1）为 $\sum_{i=1}^{n} 2^{-i} \times 2^{2-2^{k-1}}$，如果对其右移，会变成规格化数，其值为 $(\sum_{i=1}^{n-1} 2^{-i} + 1) \times 2^{2+2^{k-1}} = (\sum_{i=0}^{n-1} 2^{-i}) \times 2^{2+2^{k-1}} = 2 \times (\sum_{i=1}^{n} 2^{-i}) \times 2^{2+2^{k-1}}$，刚好是两倍关系，所以对所有非规格化数都能直接左移。

float_i2f

```
/*
 * float_i2f - Return bit-level equivalent of expression (float) x
 *   Result is returned as unsigned int, but
 *   it is to be interpreted as the bit-level representation of a
 *   single-precision floating point values.
 *   Legal ops: Any integer/unsigned operations incl. ||, &&. also if, while
 *   Max ops: 30
 *   Rating: 4
 */
unsigned float_i2f(int x) {
    unsigned ux, mask, temp, e, sign = 0;
    int E = 0, count;

    if(!x) return 0; //0就直接返回

    //将有符号数转化为无符号数
    if(x&0x80000000){
        ux = ~x+1;
        sign = 0x80000000;
    }
    else ux=x;

    //统计有几位
    temp = ux;
    while(temp){
        E += 1;
        temp = temp>>1;
    }
    ux = ux&(~(1<<(E-1))); //去掉最高位
    e = E+126; //计算e的值
    //对尾数进行移位
    if(E<=24){
        ux = ux<<(24-E); //尾数位数小于23的，直接将其移到顶
    }else{//尾数位数大于23的，要进行截断，需要考虑舍入问题
        count = 0;
        while(E>25){
            if(ux&0x01) count+=1;
            ux = ux>>1;
            E -= 1;
```

```
        }
        mask = ux&0x01;
        ux = ux>>1;
        if(mask){
            if(count) ux+=1;
            else{
                if(ux&0x01) ux+=1;
            }
        }
        if(ux>>23){//进位造成多一位
            e+=1;
            ux = ux&0x7FFFFF;//(~(1<<23)); //去掉最高位
        }
    }

    return sign+(e<<23)+ux;
}
```

将补码转化为浮点数编码步骤：

1. 将补码转化为无符号数，并根据补码的符号来设置浮点数的符号位
2. 因为补码一定是大于等于0的数，所以要么为0，要么为规格化数。如果是规格化数，首先统计除了最高有效位外一共需要几位，得到的就是E，然后通过 $E = e + 1 - 2^{k-1}$ 得到解码位为 $e = E - 1 + 2^{k-1}$ 。
3. 无符号数后面E位就是尾数部分，但是需要判断该部分是否23位，如果小于23位，直接将其左移填充；如果大于23位，需要对其进行舍入：
4.
    1. 如果是中间值，就需要向偶数舍入
    2. 如果不是中间值，就需要向最近的进行舍入

float_f2i

```
/*
 * float_f2i - Return bit-level equivalent of expression (int) f
 *   for floating point argument f.
 *   Argument is passed as unsigned int, but
 *   it is to be interpreted as the bit-level representation of a
 *   single-precision floating point value.
 *   Anything out of range (including NaN and infinity) should return
 *   0x80000000u.
 *   Legal ops: Any integer/unsigned operations incl. ||, &&. also if, while
 *   Max ops: 30
 *   Rating: 4
 */
int float_f2i(unsigned uf) {
  //只可能是规格化数
  int e = ((uf>>23)&0xFF)-127;
  unsigned sign = uf>>31;
  unsigned frac = (uf&0x7FFFFF)|0x800000; //加上1
  if(e<0) return 0; //小数
  //先转到无符号数
  if(e>31) return 0x80000000u;  //超出范围
  if(e>23) frac = frac<<(e-23);
  else frac = frac>>(23-e);
  //将无符号数转化为补码
  if(sign){ //负数
    if(frac>>31)    //溢出
        return 0x80000000u;
```

```
    else
        return ~frac+1;
  }else{    //正数
    if(frac>>31)    //溢出
        return 0x80000000u;
    else
        return frac;
  }
}
```

将浮点数转化为补码步骤：

1. 首先假设浮点数为规格化数，则 $E = e - Bias$ 得到指数部分，我们知道如果 $E < 0$，则计算出来的结果一定是小数（包括非规格化数），此时能直接舍入到0；如果 $E > 31$，表示至少要将尾数部分右移31位，此时一定会超过补码的表示范围，所以直接将其溢出。
2. 可通过最低23位得到尾数部分
3. 尾数部分需要自己在最高有效位添1，如果是负数，则补码的最高位为1，就要求其对应的无符号编码最高位不为1，否则是负溢出溢出；如果是整数，则补码的最高位为0，就要求其编码的最高位为0，否则是正溢出。

最终结果