CSAPP:Lab

Lab官网

Lab1:Data Lab

--about the bit representations,two's complement,IEEE floating point

一些开始前的配置：

> 1.dlc编译你的文件
>
> ```
> unix> ./dlc bits.c
> ```
>
> 2.使用btest进行配置
>
> 编译并运行btest程序（每次更改bit.c的时候都需要重新编译btest）
>
> ```
> unix> make btest
> unix> ./btest [可选命令行参数]
> ```
>
> 仅测试特定函数
>
> ```
> unix> make clean && make btest
> unix> ./btest -f foo
> ```
>
> 3.ishow和fshow程序：用来解读整数和浮点数的表示
>
> ```
> 构建他们
> unix> make
> 使用
> unix> ./ishow 0x27
>  Hex = 0x00000027,  Signed = 39,    Unsigned = 39
> unix> ./fshow 0x15213243
>  Floating point value 3.255334057e-26
>  Bit Representation 0x15213243, sign = 0, exponent = 0x2a, fraction = 0x213243
>  Normalized.  +1.2593463659 X 2^(-85)
> ```
>
> 4.运行./driver.pl进行打分
>
> 整数题编码规则：
>
> 每个“表达式（Expr）”只能使用下列元素：
>
> 1. 整数常量 0 到 255（包含），不能使用像 `0xffffffff` 这样的大常量。
> 2. 函数参数和局部变量（不能使用全局变量）。
> 3. 一元整型运算：`!` 和 `~`。
> 4. 二元整型运算：`&`、`^`、`|`、`+`、`<<`、`>>`。

> 明确禁止：
>
> 1. 使用任何控制结构（如 `if`、`do`、`while`、`for`、`switch` 等）。
> 2. 定义或使用任何宏。
> 3. 在此文件中定义额外的函数。
> 4. 调用任何函数。
> 5. 使用其他运算，例如 `&&`、`||`、`-`（减号）、`?:`（三目运算）等。
> 6. 使用任何形式的类型转换（casting）。
> 7. 使用任何不是 `int` 的数据类型 —— 即不能使用数组、结构体或联合体。
>
> 浮点数编码规则：
>
> 浮点题（实现与浮点相关的位级操作）规则：
>
> 这些题目规则宽松一些：允许使用循环和条件控制；可以用 `int` 和 `unsigned` 类型；可以使用任意整数常量（不限 0–255）；对 `int`/`unsigned` 可用任意算术、逻辑或比较运算。
>
> 但禁止：
>
> 1. 定义或使用宏。
> 2. 定义额外函数。
> 3. 调用任何函数。
> 4. 使用类型转换。
> 5. 使用除 `int` 或 `unsigned` 之外的数据类型（不能用数组/结构体/联合）。
> 6. 使用任何浮点数据类型、浮点运算或浮点常量。

p1:利用~和&实现^运算

思路：首先借鉴一生一芯里面的思路，就是 ^ = (~x&y)|(x&~y)，但是或运算不能用，所以我们就需要寻找其他的方法替代

德摩根律：`A|B = ~(~A & ~B)`,其实自己没用德摩根律，是自己尝试出来的

如果用德摩根律的话， A = `(~x & y)`，B = `(x & ~y)` 代入即可

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

p2:返回最小的二进制补码整数

这个题自己没有完全做下来

> 只能用 ! ~ & ^ | + << >>，最大操作数4

补码复习：最小的二进制补码整数：TMIN：符号位为1，其余全是0（负数最大，正数最小）

注意只能用0-255的数字

int 4byte->32bits

```
              0000 (0)
      (-1) 1111  0001 (1)
   (-2) 1110   ^    0010 (2)
 (-3) 1101     |      0011 (3)
(-4) 1100      +       0100 (4)
 (-5) 1011             0101 (5)
   (-6) 1010         0110 (6)
      (-7) 1001  0111 (7)
              1000 (-8)
```

没想到这种最简单的方法，直接记住这种左移的思路求最小值吧

```
int tmin(void) {
  return ~0 << 31;
}
```

p3：判断是否为最大的整数（补码编码）

这个题自己没有完全做下来

TMAX：符号位为0,剩下的全为1

> 只能用！~ & ^ | +

思路：需要计算x+1,为什么需要x+1,一定要抓住TMAX特殊的地方，然后想办法将其变成0000,但是这里还需排除-1和0的情况

```
int isTmax(int x) {
  /*int y =x + x+1+1; 这里溢出了不能这么算,先+1再+x
  return (!y) ^(!(x+1));*/
  int y = ~((x+1)+x);
  return (!y)^(!(x+1));
}
```

p4：判断奇数位是不是全为1--mask的使用

这个题自己没有完全做下来

一定要区分逻辑非!和按位取反~的区别：

`!`：是非0的就变成0,0就变成1（**"满足条件返回1,不满足返回0"：这种就用到！，我们需要将满足条件的变为0然后就可以使用！用来返回结果了**）

`~`:这个就是按位取反，跟逻辑非有本质的区别

```
int allOddBits(int x) {
  int mask = 170;
  mask = (mask << 8) + mask;
  mask = (mask << 16) + mask;
  return !(mask^(mask&x));
}
```

p5：补码取反（没什么好说的）

```
int negate(int x) {
  return ~x+1;
}
```

p6：判断ASCII码--mask的使用

思路：需要判断高24位是不是全是0,低8位中上半个4位是不是0011（即是不是0x**3**.）,判断低4位的值（这里观察二进制码写即可）

```
int isAsciiDigit(int x) {
  //不仅要检查低8位，还要检查高24位需要全是0
  int x1 = (8&x)^8;
  int x2 = ((14&x)^8);
  return (!(((~15)&x)^48)) &( (!!x1)|(!x2));
}
```

p7：用位运算实现三目运算符--x非0返回y，是0返回z

不是自己完全做下来的

这里注意!只返回最低位，因为得到的是00000000000000000000000000000001,这样和y/z&只会留下最低位

如何将00000000000000000000000000000001扩展成11111111111111111111111111111111是个问题

可以取反再+1

```
int conditional(int x, int y, int z) {
  return ((~(!(x^0))+1)&z) | ((~(!!(x^0))+1)&y);
}
```

p8：用位运算实现大小比较

思路欠缺：使用符号位判定，并且考虑溢出的情况

思路：如果符号不同，那么x为负则一定小

如果符号相同比较y-x（这里不能在不同符号的情况下比较sub，排除TMAX，TMIN溢出的情况）

如果sign（y-x）为1，那么y小，否则x小

这里需要参照一生一芯的关于补码溢出的那里，如果是异号相加不会出现溢出，如果是同号相加就会出现溢出的现象（即符号位反转的现象，所以这里计算sub不能在不同符号（取反就变成相同符号的情况下进行相加））

```
int isLessOrEqual(int x, int y) {
  //计算差值
  int signx = (x>>31)&1;
  int signy = (y>>31)&1;
  int sub = y+(~x+1);
  int signsub = (sub >> 31)&1;
  return ((signx^signy) & signx) | ((!signsub)&!(signx^signy));
}
```

p9：用其他运算符实现!运算符

思路欠缺：学习如何区分0和非0

观察可以发现，只有0其相反数（补码下为~x+1）符号位不变，其他数相反数符号位是一定会变的，所以我们可以利用这个思路区分0和非0

所以可以计算x和其相反数~x+1的符号位是否相同

```
x>>31
~x+1 >> 31
如果是0的话那么二者的值都为0,如果非0二者一个为0一个为1
所以这时候就可以利用位运算将其合并起来，并加以区分二者，然后再进行计算就可以了
```

p10返回整数x的二进制补码的最少位数

很难，一点思路都没有

首先应该明确这个题的意思是，需要多少位可以将这个数和其他的数区分开来，而不是说实际计算机存储这个数需要多少位

特殊，如果是0的话，那么我们只需要1位即可

正数：如果是正数的话，我们需要1位符号位加上最高位的1来表示

负数：如果使用补码的思路就错了，并不是最小位数（比如-1需要用32位，其实只需要用一位），因为负数的表示是原码取反+1来的，所以我们可以用同样的思路，将负数取反映射到正数上面，变成找最前面的1来简化

这样负数和正数的逻辑就明朗了，都是**符号位加上最高位的1来表示**

现在的思路就是如何找到最高位的1了----**二分查找**

学习如何使用位运算去模拟if语句的进行

```
int howManyBits(int x) {
  //首先判断是不是0的情形
  int isZero = x^0;//是0返回0
  int sum = 1; //提前加上符号位的1
  //对于负数我们需要先取反
  int sign = x>>31; //注意这里是符号右移int类型默认是符号右移，负数就是全1,正数就是全0
  x = (sign & ~x) | (~sign & x);
  //开始二分查找
  int hb16,hb8,hb4,hb2,hb1,hb0;
  //先查找高半区有没有1, 如果有1,!!(x>>16)返回1,没1返回0
  //<<4是因为如果有1的话就变成10000,没1的话还是0
```

```
  hb16 = !!(x>>16)<<4;
  //有1的话就右移16位，没1的话就是不移动，这一行体现了if语句的分支情况
  //不管移动还是不移动，下一次分析的位数就只有16位
  x = x>>hb16;
  hb8 = !!(x>>8)<<3;
  x = x>>hb8;
  hb4 = !!(x>>4)<<2;
  x = x>>hb4;
  hb2 = !!(x>>2)<<1;
  x = x>> hb2;
  hb1 = !!(x>>1);
  x = x >> hb1;
  hb0 = x;
  sum = sum + hb16+hb8+hb4+hb2+hb1+hb0;
  return !isZero | ((~(!!isZero)+1)&sum);
}
```

p11:使用位运算将浮点数×2

这个题使用**unsigned32位无符号数来编码**的

思路：如果是特殊值直接返回即可，如果是0也直接返回

如果是非规格化数，一般来说就是直接把**frac<<1,**但是这种需要计算是否溢出的问题，如果溢出了，说明**变成了规格化数，这时候frac的编码从0.f变成1.f，但是我们计算的frac只需要将溢出位去掉即可，然后再将exp+1**，如果没溢出，那么直接返回即可

如果是规格化数，一般来说就是直接将**exp+1**,×2有可能变成无穷大，如果变成无穷大需要将frac置0

```
unsigned floatScale2(unsigned uf) {
  unsigned exp = (uf >> 23)&0x000000FF;
  unsigned s = uf >> 31;
  unsigned frac = uf & 0x007FFFFF;
  if(exp == 0x000000FF){
      return uf;
  }else if(exp == 0x0){
    if(frac == 0x0) return uf;
    else{
      //非规格化数的话0.f先左移一位（×2）
      frac = frac << 1;
      //检查第23位是否为1,如果是的话说明溢出了，需要转换成规格化数
      if(frac &0x800000){
        exp +=1;
        frac &= 0x007FFFFF;
      }

    }
  }else{
    //规格化数直接给指数×2即可
    exp +=1;
    //但是要判断是否为255,如果是的话需要返回无穷大
    if(exp == 0xFF){
      frac = 0;
    }
```

```
  }
  unsigned f = (s<<31)|(exp<<23)|frac;
  return f;
}
```

p12：使用位运算将浮点数转换为int

思路：当特殊数的时候肯定是返回0x80000000u这么一个无效值了

接下来分析非规格数的情况，无论是0还是接近0的数经过分析都比1小，所以直接返回0

接下来就是着重分析规格化数（规格化）E从-126到127

1. 因为int的范围是从-2^31到2^31-1，所以E从-126到0直接舍入为0（不够1），E从31到127（int表示不了）直接返回无效值
2. 接下来分析E>= 0 && E<=30

   f为24位，如果E<=23是保存不下来整个f的，这时候我们直接舍去保存不下来的f（向下舍入）

   如果E>23，那么我们就应该扩大f了

这里有个坑就是不能在计算int数值之前先把s符号位加到res里面，因为不能简单的符号位+绝对值组合成int，必须按照二进制补码的形式进行计算，所以这里我们就不计算了，直接（int）类型转换交给编译器去转换吧

```
int floatFloat2Int(unsigned uf) {
  unsigned exp = (uf >> 23)&0x000000FF;
  unsigned s = uf >> 31;
  unsigned frac = uf & 0x007FFFFF;
  int bias = 127;
    //这里E一定要用int，unsigned只有正数范围，无法表示E的负数情况，会出错
  int E = exp - bias;
  unsigned f = frac + 0x800000; //1+frac
  int res = 0;
  if(exp == 0x000000FF){
    return 0x80000000u;
  }else if(exp == 0x0){
    return 0;
  }else{
      //这里开始考虑规格化数的情况
      //首先应该知道int的范围应该是从-2^31到2^31-1这么大的范围
      //规格化数E从-126到127,从-126到0都应该返回0
      //然后E从31到127的过程，这时候都应该返回无效值
      if(E>=-126 && E<0) return 0;
      else if(E >30) return 0x80000000u;
      else{
        //f24位，需要根据E的范围舍入,E从0到23位的范围内，f是右移的，即直接舍去小数点后面的数字,这里采取的舍入方式是向下舍入
        if(E>=0 && E<= 23) res += (f>> (23-E));
        else{
          //E从24到30的范围，f是左移动的，这时是可以完整保留下来f的
          f = f << (E-23);
          res += f;
        }
      }
  }
```

```
  //设置符号位
  if(s == 0x1) res = -res;
  else res = res;
  return res;
}
```

p13：通过给定x计算2^x--手动拼出IEEE浮点表示形式

思路：因为是2^x所以尾数位应该全是0，符号位也恒为正，所以这个题只需要分析指数位即可，思路很简单

这里不再赘述了

```
unsigned floatPower2(int x) {
  //因为表示2^x所以尾数位只能为0,NAN和0相近的数其实是不用考虑的因为尾数不为0
  /*规格化数E从-126到127,非规格化数E为-126,特殊值128*/
  unsigned res = 0;
  if(x<= -126) return 0;
  else if(x >= 128) return 0x7F800000;
  else{
    unsigned e =(x+127);
    res += (e<<23);
  }

  return res;
}
```

最终结果

```
lyjy@Lenovo:~/csapplab/datalab/datalab-handout$ ./btest
Score   Rating  Errors  Function
 1      1       0       bitXor
 1      1       0       tmin
 1      1       0       isTmax
 2      2       0       allOddBits
 2      2       0       negate
 3      3       0       isAsciiDigit
 3      3       0       conditional
 3      3       0       isLessOrEqual
 4      4       0       logicalNeg
 4      4       0       howManyBits
 4      4       0       floatScale2
 4      4       0       floatFloat2Int
 4      4       0       floatPower2
Total points: 36/36
```

该了一点小问题最终的得分结果

```
Correctness Results     Perf Results
Points  Rating  Errors  Points  Ops     Puzzle
1       1       0       2       8       bitXor
1       1       0       2       2       tmin
1       1       0       2       7       isTmax
```

```
2       2       0       2       7       allOddBits
2       2       0       2       2       negate
3       3       0       2       13      isAsciiDigit
3       3       0       2       12      conditional
3       3       0       2       16      isLessOrEqual
4       4       0       2       5       logicalNeg
4       4       0       2       44      howManyBits
4       4       0       2       17      floatScale2
4       4       0       2       24      floatFloat2Int
4       4       0       2       6       floatPower2

Score = 62/62 [36/36 Corr + 26/26 Perf] (163 total operators)
```