data lab-PB17111568-郭雨轩

实验简介

data lab算是csapp中最水的实验之一了，实验主要涉及到使用各种位运算的奇怪操作来实现特定的功能。

实验过程

0. 实验环境搭建

- 由于我是在wsl里面进行的实验，由于wsl不能运行32位linux程序，所以需要魔改一下Makefile，去掉`-m32`编译选项即可。

1. bitXor

> */** 要求
>
> * *bitXor - x^y using only ~ and &*
>
> * *Example: bitXor(4, 5) = 1*
>
> * *Legal ops: ~ &*
>
> * *Max ops: 14*
>
> * *Rating: 1*
>
> */

- 基本思路是，首先列出真值表，找到最小项表达式，用`&`和`~`实现`|`，因为有`~((~a)&(~b)) == a|b`，所以按照最小项表达式组合一下即可，代码如下：

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

2. tmin

> */** 要求
>
> * *tmin - return minimum two's complement integer*
>
> * *Legal ops: ! ~ & ^ | + << >>*
>
> * *Max ops: 4*
>
> * *Rating: 1*
>
> */

- 返回补码数中最小的一个？~~这是什么沙雕送分题...~~，结果显然是`0x80000000`

```c
int tmin(void) {
    return (1<<31);
}
```

3. isTmax

> */** 要求
>
> * *isTmax - returns 1 if x is the maximum, two's complement number,*
>
> * *and 0 otherwise*
>
> * *Legal ops: ! ~ & ^ | +*
>
> * *Max ops: 10*
>
> * *Rating: 1*
>
> */

- 最大的正补码数是`0x7fffffff`，若x为这个数字，则`~(2x+1)==0`，那么只要返回`!(~(2x+1))`即可，运行测试代码后发现，测试数据在`0xffffffff`不能通过。所以还需要过滤掉这种情况，那么如何过滤呢，只需要检查x是否为-1，使用`!`运算符可以将整数转换为bool类型，而bool类型其实就是一个bit的整形，所以这句的用处`!(!(x+1))`是实现检查是否为-1，最后代码如下：

```c
int isTmax(int x) {
    return (!(~(x+x+1)) & !(!(x+1)));
}
```

4. allOddBits

> */** 要求
>
> * *allOddBits - return 1 if all odd-numbered bits in word set to 1*
>
> * *where bits are numbered from 0 (least significant) to 31 (most significant)*
>
> * *Examples allOddBits(0xFFFFFFFD) = 0, allOddBits(0xAAAAAAAA) = 1*
>
> * *Legal ops: ! ~ & ^ | + << >>*
>
> * *Max ops: 12*
>
> * *Rating: 2*
>
> */

- 要求判断所有的奇数位是否都为1，这里的奇数位指的是2的奇数次方位，其实就是数字中的偶数位，方法跟简单，由于限制使用0-0xff的数字，所以使用`0xaa`作为基础的掩码构造一个32位的掩码即可，代码如下：

```c
int allOddBits(int x) {
    int mask = 0xAA;
    mask = (mask << 8)+mask;
    mask = (mask << 16)+mask;
    return !((x&mask) + ~mask + 1);
}
```

5. negate

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

- 这个题目太水了...常识题目

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

6. isAsciiDigit

> */** 要求
>
> * 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
>
> */

- 分别用上界减去x和用x减去下界，判断两个结果是否都是正数即可。

```c
int isAsciiDigit(int x) {
    int DownBound = x + ~0x30 + 1;
    int UpBound = 0x39 + ~x + 1;
    return (!(DownBound>>31)) & (!(UpBound>>31));
}
```

7. conditional

> /*
>
> * conditional - same as x ? y : z
>
> * Example: conditional(2,4,5) = 4
>
> * Legal ops: ! ~ & ^ | + << >>
>
> * Max ops: 16
>
> * Rating: 3
>
> */

- 本实验要求实现一个类似于c语言中的3目运算符的函数，思路是这样的。首先根据输入的x的值是否为0来判断是哪一种情况，根据x的值来构造两种掩码，`0xffffffff`和`0x00000000`，再使用类似于数据选择器的思路即可实现这个3目运算符。

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

8. isLessOrEqual

> */**
>
> ** isLessOrEqual - if x <= y then return 1, else return 0*
>
> ** Example: isLessOrEqual(4,5) = 1.*
>
> ** Legal ops: ! ~ & ^ | + << >>*
>
> ** Max ops: 24*
>
> ** Rating: 3*
>
> */

- 本题主要是实现两个补码数比较大小，对于同符号的补码数，只需要相减再看下结果的符号位是否为0就可以，对于符号不一致的补码数，相减可能导致溢出，所以直接比较大小即可。通过将两个数字的符号位进行异或实现分两种情况讨论。

```
int isLessOrEqual(int x, int y) {
    int result = y+~x+1;
    int a=x>>31;
    int b=y>>31;
    int xor = a^b;
    return ((!xor)&!(result >> 31)) | (a&!b);
}
```

9. 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
>
> */

- 我们知道，在所有补码数中，除掉 0x00000000 和 0x80000000 外，其余所有的数字的补码和其相反数的补码符号位不同（一正一负），利用这个性质，我们可以将要特别考虑的整数范围缩小到上面两个数字，其中，对于 `0x80000000`，将其与自身取反加1的结果按位或之后，符号位也为1，所以对于所有的非0补码数，其右移-1，而对于0，其右移结果为0，所以只需要再加上1就好了。

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

10. 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
>
> */

- 这个题一开始我并不会，自己尝试写了一个逐位判断的程序后，发现超出了最大要求的运算符号的数目，在网上搜索后，发现了一个很妙的答案，把它弄懂了。这种方法的基本思路是使用二分法进行查找，将符号位全部转换为1，先查找高16位中是否有1，若有则至少需要16位。再依次使用二分查找的方式进行查找即可。

```
int howManyBits(int x) {
  int b16,b8,b4,b2,b1,b0;
  int sign=x>>31;
```

```
  x = (sign&~x)|(~sign&x);
  b16 = !!(x>>16)<<4;
  x = x>>b16;
  b8 = !!(x>>8)<<3;
  x = x>>b8;
  b4 = !!(x>>4)<<2;
  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
}
```

11. floatScale2

> */**
>
> * floatScale2 - 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
>
> */

- 对于一个`ieee-754`的浮点数，我们知道I[31]是符号位，I[30:23]是指数位，I[22:0]是数值位，因此实现2乘上一个浮点数只需要将其exp位提取出来加上一即可，同时还要考虑一些边界条件。

```
unsigned floatScale2(unsigned uf) {
    int expo = (uf&0x7f800000)>>23;
    int sign = uf&(1<<31);
    if(expo==255)//若为无穷大，返回无穷大
        return uf;
    if(expo==0)//若是0
        return uf<<1|sign;
    expo++;
    if(expo==255)//若乘完是无穷大，返回无穷大
        return 0x7f800000|sign;
    return (expo<<23)|(uf&0x807fffff);
}
```

12. floatFloat2Int

> */**
>
> * floatFloat2Int - 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
>
> */

- 将浮点数转换为整数。因为float和int表示的数字范围不同，首先要先排除int不能表示的范围，int可以表示$-2^{31}\ \ to\ \ (2^{31}-1)$的数字所以所以对于绝对值小于1或者大于$2^{31}$的数字要舍弃。接下来按照浮点数的定义，使用位级表示强行转换即可.

```
int floatFloat2Int(unsigned uf) {
    int s=uf>>31;
    int exp=((uf&0x7f800000)>>23)-127;
    int frac=(uf&0x007fffff)|0x00800000;
    if(!(uf&0x7fffffff))
        return 0;
    if(exp>31)
        return 0x80000000;
    if(exp<0)
        return 0;

    if(exp>23)
        frac<<=(exp_-23);
    else
        frac>>=(23-exp_);
    if(!((frac>>31)^s))
        return frac;
    else if(frac>>31)
        return 0x80000000;
    else
        return ~frac+1;
}
```

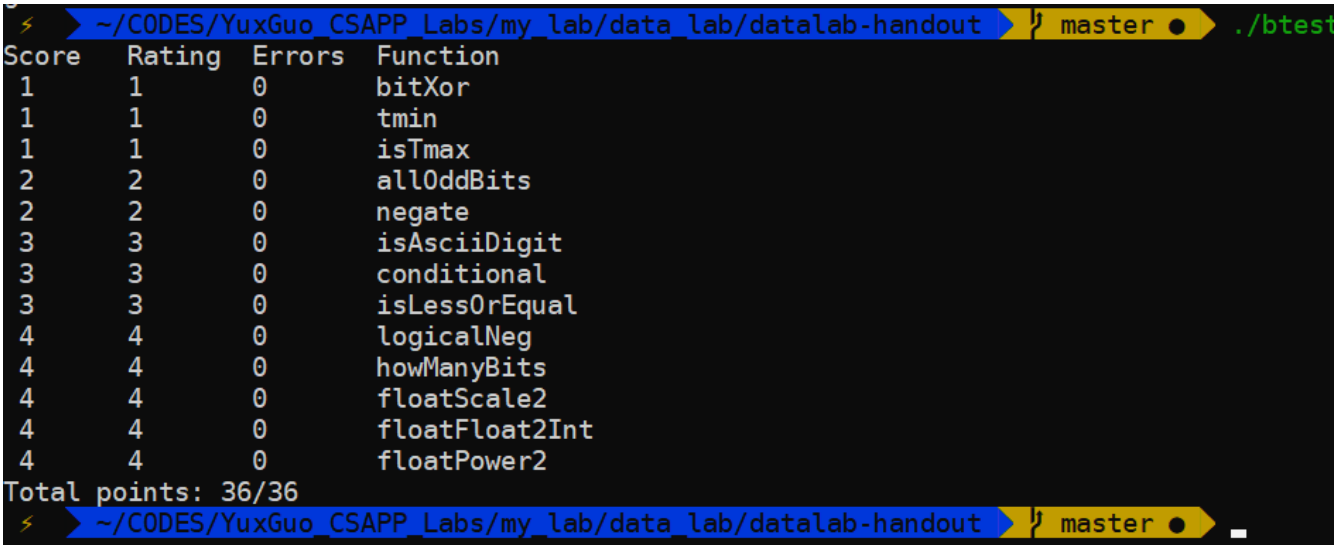
13. floatPower2

> */**
>
> ** floatPower2 - Return bit-level equivalent of the expression 2.0^x*
>
> ** (2.0 raised to the power x) for any 32-bit integer x.*
>
> ---
>
> ** The unsigned value that is returned should have the identical bit*
>
> ** representation as the single-precision floating-point number 2.0^x.*
>
> ** If the result is too small to be represented as a denorm, return*
>
> ** 0. If too large, return +INF.*
>
> ---
>
> ** Legal ops: Any integer/unsigned operations incl. ||, &&. Also if, while*
>
> ** Max ops: 30*
>
> ** Rating: 4*
>
> */

- 这个比较简单，首先分析得2的浮点数表示为，指数为128，其余都是0。表示为1×2^1。那么$2^x = (1 \times 2^1)^x = 1 \times 2^x$。所以其实结果的指数位就是$x + 127$，代码如下。

```
unsigned floatPower2(int x) {
    int INF = 0x7f800000;
    int expo = x + 127;
    if(expo <= 0)
    {
      return 0;
    }
    if(expo >= 255)
    {
      return INF;
    }
    return expo << 23;
}
```

实验截图



实验收获

本次实验加深了我对位操作的熟悉同时对IEEE754浮点数有了更加深刻的了解，同时，对于本次实验中的一些算法也有了了解，收获很大。