

一、实验目标：

1. 了解各种数据类型在计算机中的表示方法
2. 掌握 C 语言数据类型的位级表示及操作

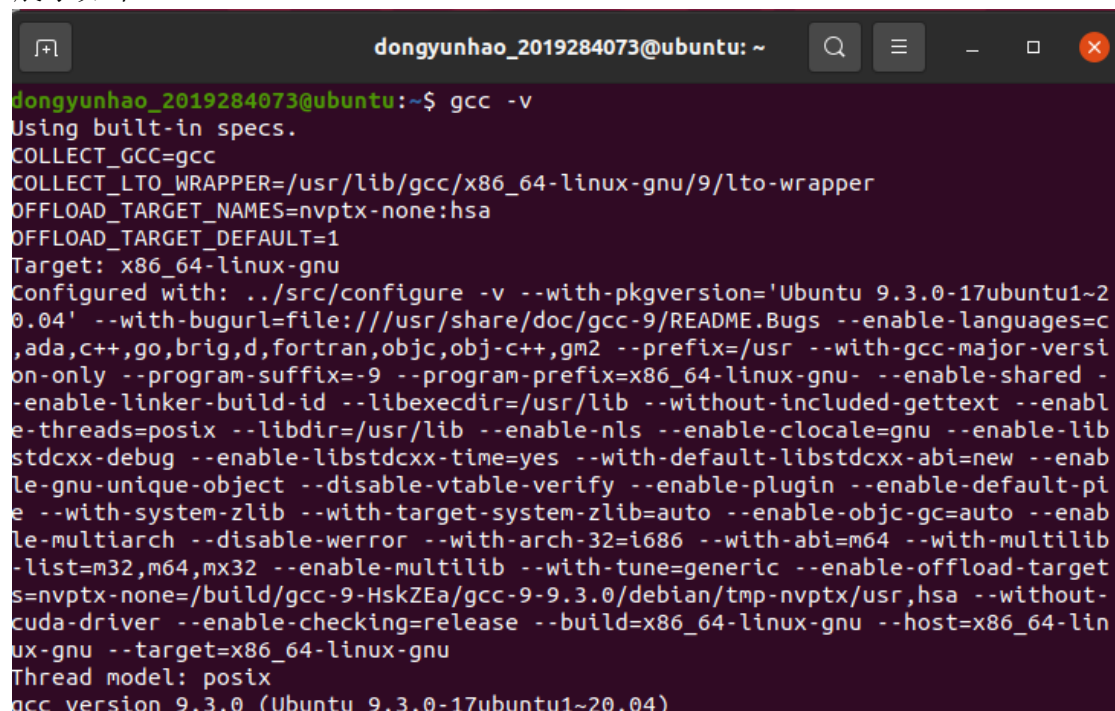
二、实验环境：

1. 计算机（Intel CPU）
2. Linux 操作系统

三、实验内容与步骤

实验前准备：

实验前首先需要安装 GCC 编译环境，在我的 Ubuntu 上已经安装了编译环境，展示如下：



```
dongyunhao_2019284073@ubuntu: ~  
dongyunhao_2019284073@ubuntu:~$ gcc -v  
Using built-in specs.  
COLLECT_GCC=gcc  
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/9/lto-wrapper  
OFFLOAD_TARGET_NAMES=nvptx-none:hsa  
OFFLOAD_TARGET_DEFAULT=1  
Target: x86_64-linux-gnu  
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 9.3.0-17ubuntu1~20.04' --with-bugurl=file:///usr/share/doc/gcc-9/README.Bugs --enable-languages=c,ada,c++,go,brig,d,fortran,objc,obj-c++,gm2 --prefix=/usr --with-gcc-major-version-only --program-suffix=-9 --program-prefix=x86_64-linux-gnu- --enable-shared --enable-linker-build-id --libexecdir=/usr/lib --without-included-gettext --enable-threads=posix --libdir=/usr/lib --enable-nls --enable-clocale=gnu --enable-libstdcxx-debug --enable-libstdcxx-time=yes --with-default-libstdcxx-abi=new --enable-gnu-unique-object --disable-vtable-verify --enable-plugin --enable-default-pie --with-system-zlib --with-target-system-zlib=auto --enable-objc-gc=auto --enable-multiarch --disable-werror --with-arch-32=i686 --with-abi=m64 --with-multilib-list=m32,m64,mx32 --enable-multilib --with-tune=generic --enable-offload-targets=nvptx-none=/build/gcc-9-HskZEa/gcc-9-9.3.0/debian/tmp-nvptx/usr,hsa --without-cuda-driver --enable-checking=release --build=x86_64-linux-gnu --host=x86_64-linux-gnu --target=x86_64-linux-gnu  
Thread model: posix  
gcc version 9.3.0 (Ubuntu 9.3.0-17ubuntu1~20.04)
```

- 1、根据 bits.c 中的要求补全以下的函数：

```
int bitXor(int x, int y);  
int tmin(void);  
int isTmax(int x);  
int allOddBits(int x);  
int negate(int x);  
int isAsciiDigit(int x);  
int conditional(int x, int y, int z);  
int isLessOrEqual(int x, int y);  
int logicalNeg(int x);
```

```
int howManyBits(int x);
unsigned float_twice(unsigned uf);
unsigned float_i2f(int x);
int float_f2i(unsigned uf);
```

2、在 Linux 下测试以上函数是否正确，指令如下：

*编译：./dlc bits.c

*测试：make btest

./btest

四、实验结果

1、根据 bits.c 中的要求补全以下的函数：

(1) int bitXor(int x, int y);

①题目描述：

仅允许使用~和&来实现异或

例子: bitXor(4, 5) = 1

允许的操作符: ~ &

最多操作符数目: 14

②大致思路：

依离散数学知识，可以列出对于单个 bit 的真值表如下：

X	Y	Output
0	0	0
0	1	1
1	0	1
1	1	0

即有 $Output = (\sim X \& Y) | (X \& \sim Y)$ 。又因为按位或不为合法运算符，故依德摩根律，有 $Output = \sim(\sim(\sim X \& Y) \& \sim(X \& \sim Y))$

③编程实现：

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

(2) int tmin(void);

①题目描述：

返回最小的二进制补码

允许的操作符: ! ~ & ^ | + << >>

最多操作符数目: 4

②大致思路:

对于 32 位整数, 最小值即为 0X 8000 0000, 即将 1 左移 31 位。

③编程实现:

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

(3) int isTmax(int x);

①题目描述:

如果 x 是最大的二进制补码, 返回 1; 否则, 返回 0

允许的操作符: ! ~ & ^ | +

最多操作符数目: 10

分值: 2

②大致思路:

可以知道, 最大值为 0x7fff ffff, 加一后将变为 0x8000 0000, 且此数加上本身后将变为 0。本身加本身为 0 的数只有 0 和 0x8000 0000, 因此, 只需将 0xffffffff 排除即可:

③编程实现:

```
/*
 * 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) {
    return (!(x + 1 + x + 1)) & (!! (x + 1));
}
```

(4) int allOddBits(int x);

①题目描述:

如果所有奇数位都为 1 则返回 1;否则返回 0

例子: `allOddBits(0xFFFFFFFF) = 0`

`allOddBits(0xAAAAAAAA) = 1`

允许的操作符: `! ~ & ^ | + << >>`

最多操作符数目: 12

②大致思路:

在二进制下, 有且仅有所有位为奇数的数与 `0x5555 5555` 进行与运算后由 `0xffff ffff`, 变为 `0x0000 0000`。因此可以通过对其与 `0x5555 5555` 进行与运算并取反再进行取逻辑反获得结果。又因为题干中不允许使用大于 256 的整数, 故需要通过一些操作获得 `0x5555 5555`。

可以通过将 `0x0505` 分别进行左移 4、8、16、24 得到 4 个数, 并将四个数求和获得 `0x5555 5555`。

③编程实现:

```
/*
 * allOddBits - return 1 if all odd-numbered bits in word set to 1
 * Examples allOddBits(0xFFFFFFFF) = 0, allOddBits(0xAAAAAAAA) = 1
 * Legal ops: ! ~ & ^ | + << >>
 * Max ops: 12
 * Rating: 2
 */
int allOddBits(int x) {
    unsigned int a = 85;
    unsigned int b = a + (a << 8) + (a << 16) + (a << 24);
    return !(b | x);
}
```

(5) `int negate(int x);`

①题目描述:

返回 x 的相反数

例子: `negate(1) = -1`.

允许的操作符: `! ~ & ^ | + << >>`

最多操作符数目: 5

②大致思路:

即取反并加一返回即可。

③编程实现:

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

(6) int isAsciiDigit(int x);

①题目描述:

如果 x 是 ascii 码中的 0~9, 返回 1; 否则返回 0

例子: isAsciiDigit(0x35) = 1.

isAsciiDigit(0x3a) = 0.

isAsciiDigit(0x05) = 0.

允许的操作符: ! ~ & ^ | + << >>

最多操作符数目: 15

②大致思路:

即对于每个输入的 x, 需要满足 $x \geq '0'$ 且 $x \leq '9'$, 因此可以将 x 与临界值进行作差。并通过右移 31 位判断对符号位进行判断是 0 还是 1 即可。

③编程实现:

```
/*
 * 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) {
    return (!((x + ~48 + 1) >> 31)) & !((x + ~58 + 1) >> 31);
}
```

(7) int conditional(int x, int y, int z);

①题目描述:

实现 $x ? y : z$

例子: conditional(2,4,5) = 4

允许的操作符: ! ~ & ^ | + << >>

最多操作符数目: 16

②大致思路:

首先, 不难想到, 对于 x 的判断可以通过 $t = !x$ 进行判断实现, 当 x 为 0 时返回 1; 当 x 不为 0 时返回 0。因此, 可以将表达式大致转成 $(_ \&y) | (_ \&z)$ 的格式进行配凑。

对于前面的空格, 当 x 不为 0, 即 $t=0$ 时, 需要 t 转换为 0xffff ffff (-1)。可以通过对 1 按位取反再加一 1 获得。

对于后面的空格, 当 x 为 0, 即 $t=1$ 时, 也需要 t 转换为 0xffff ffff (-1)。此时直接对 t 进行取反并加一即可。

③编程实现:

```

/*
 * 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) {
    return ((!x + ~1 + 1) & y) | ((~!x + 1) & z);
}

```

(8) `int isLessOrEqual(int x, int y);`

①题目描述:

如果 $x \leq y$ 返回 1 否则返回 0

例子: `isLessOrEqual(4,5) = 1.`

允许的操作符: `! ~ & ^ | + << >>`

最多操作符数目: 24

②大致思路:

首先应该很容易会想到直接采用 $y-x$ 并判断符号位的方法进行判断, 但如果作差相减, 有可能会发生 `int` 类型溢出, 因此需要考虑其他方法。

a. 当 x, y 同号: 此时, 即可将问题转化为转换为 $p=y-x \geq 0$, 并对 p 的符号位 (通过右移获得) 进行判断, 获得运行结果。

b. 当 x, y 异号: 此时只要 $x \geq 0$, 就可以返回 0, 否则返回 1。

c. 是否同号的判断: 可以通过对符号位进行求和判断是否同号。

③编程实现:

```

/*
 * 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) {
    int a = x >> 31;
    int b = y >> 31;
    int c = a + b;
    int p = y + (~x + 1);
    int q = !((p >> 31) & 1);
    int r = (c & (a & 1)) | ((~c) & q);
    return r;
}

```

(9) `int logicalNeg(int x);`

①题目描述:

实现! 运算符的功能

例子: $\text{logicalNeg}(3) = 0$, $\text{logicalNeg}(0) = 1$

允许的操作符: $\sim \& \wedge | + \ll \gg$

最多操作符数目: 12

②大致思路:

可以通过取相反数进行非零判断。令 $y = \sim x + 1$ ($y = -x$) 并讨论 x 与 y 的符号位, 有如下几种情况:

- 当 x 为 0 时, 两者符号位都为 0.
 - 当 $x = 0x8000\ 0000$ 时, 两者符号位都为 1.
 - 当 x 既不为 0 也不为 $0x8000\ 0000$ 时, 两者符号位为 01 或 10.
- 因此, 可依真值表得 $\text{ans} = (\sim x) \& (\sim y)$.

③编程实现:

```
/*
 * 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) {
    return ((~(~x + 1) & ~x) >> 31) & 1;
}
```

(10) int howManyBits(int x);

①题目描述:

返回将 X 表示为补码所需的最小有效位数。

例子: $\text{howManyBits}(12) = 5$

$\text{howManyBits}(298) = 10$

$\text{howManyBits}(-5) = 4$

$\text{howManyBits}(0) = 1$

$\text{howManyBits}(-1) = 1$

$\text{howManyBits}(0x80000000) = 32$

允许的操作符: $! \sim \& \wedge | + \ll \gg$

最多操作符数目: 90

②大致思路:

通过二分完成算法实现, 具体设计如下:

我们举一个例子进行说明: $x = 0000\ 1000\ 1001\ 0000\ 0000\ 0000\ 0000\ 0000$

a. 令 $y = x \gg 16 = 0000\ 1000\ 1001\ 0000$, $\text{shift16} = (!y) \ll 4$ 使用 $(!y)$ 判断 y 是否为 0, 如果 y 为 0, 则返回 0, 说明前 16 位都为 0, $\text{shift16} = 0$, 否则 $!y = 1$, $\text{shift16} = 16$,

使 $x = x \gg \text{shift16} = 10001001\ 0000$;

b.同理, 令 $y = x \gg 8 = 0000\ 1000$, $\text{shift8} = (!y) \ll 3$ 使用 $(!y)$ 判断 y 是否为 0, 如果 y 为 0, 则返回 0, 说明前 8 位都为 0, $\text{shift8} = 0$, 否则 $!y = 1$, $\text{shift8} = 8$, 使 $x = x \gg \text{shift8} = 00001000$;

c.令 $y = x \gg 4 = 0000$, $\text{shift4} = (!y) \ll 2$ 使用 $(!y)$ 判断 y 是否为 0, 如果 y 为 0, 则返回 0, 说明前 4 位都为 0, $\text{shift4} = 0$, 否则 $!y = 1$, $\text{shift4} = 4$, 使 $x = x \gg \text{shift4} = 00001000$;

d.令 $y = x \gg 2 = 0000\ 10$, $\text{shift2} = (!y) \ll 1$ 使用 $(!y)$ 判断 y 是否为 0, 如果 y 为 0, 则返回 0, 说明前 2 位都为 0, $\text{shift2} = 0$, 否则 $!y = 1$, $\text{shift2} = 2$, 使 $x = x \gg \text{shift2} = 000010$;

e.令 $y = x \gg 1 = 0000\ 1$, $\text{shift1} = (!y)$, 使用 $(!y)$ 判断 y 是否为 0, 如果 y 为 0, 则返回 0, 说明前 1 位都为 0, $\text{shift1} = 0$, 否则 $!y = 1$, $\text{shift1} = 1$, 使 $x = x \gg \text{shift1} = 00001$;

f.最后对各个二分进行求和即可;

g.对于负数取反码即可, 对于 -1 和 0 需要进行特殊考虑。

③编程实现:

```
/* 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 shift1, shift2, shift4, shift8, shift16;
    int sum;
    int t = ((!x) << 31) >> 31;
    int t2 = ((!~x) << 31) >> 31;
    int op = x ^ ((x >> 31));
    shift16 = (!!(op >> 16)) << 4;
    op = op >> shift16;
    shift8 = (!!(op >> 8)) << 3;
    op = op >> shift8;
    shift4 = (!!(op >> 4)) << 2;
    op = op >> shift4;
    shift2 = (!!(op >> 2)) << 1;
    op = op >> shift2;
    shift1 = (!!(op >> 1));
    op = op >> shift1;
    sum = 2 + shift16 + shift8 + shift4 + shift2 + shift1;
    return (t2 & 1) | ((~t2) & ((t & 1) | ((~t) & sum)));
}
```


(11) unsigned float_twice(unsigned uf);

①题目描述:

以 unsigned 表示的浮点数二进制的二倍的二进制 unsigned 型
参数和结果都会被作为 unsigned 返回,但是会表示为二进制的单精度浮点值。
允许的操作符: 任何整数或者无符号数操作符包括: ||, &&. also if, while
最多操作符数目: 30

②大致思路: (exp 表示阶码位、frac 表示尾数位)

通过分析可知, 共存在三种可能的情况, 分别考虑如下:

a. 当 exp=0xff 时, 直接返回本身即可;

b. 当 exp=0 时, 分两种情况考虑:

 当 uf[22]=0 时, 然后将 frac 左移一位即可

 当 uf[22]=1 时, 将 exp 自增 1, 然后再将 frac 左移一位即可

c. 对于其他情况, 将 exp 自增 1, 然后再分两种情况:

 当 exp==0xff, 令 frac=0 即可。

 其余情况正常左移并返回即可

③编程实现:

```

/*
 * 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) {
    unsigned int exp = 0x7f800000 & uf;
    unsigned int frac = 0x007fffff & uf;
    unsigned int s = 0x80000000 & uf;
    if (((~exp) & 0x7f800000) == 0) {
        return uf;
    } else {
        if (exp == 0) {
            if ((0x00400000 & uf) == 0) {
                frac = frac << 1;
            } else {
                exp = 0x00800000;
                frac = frac << 1;
            }
        } else {
            exp = exp + 0x00800000;
            if (((~exp) & 0x7f800000) == 0)
                frac = 0;
        }
        return (exp | frac | s);
    }
}

```

(12) unsigned float_i2f(int x);

①题目描述:

返回 int x 的 unsigned 浮点数的二进制形式

参数和结果都会被作为 unsigned 返回，但是会表示为二进制的单精度浮点值

允许的操作符: 任何整数或者无符号数操作符包括: ||, &&. also if, while

最多操作符数目: 30

②大致思路: (exp 表示阶码位、frac 表示尾数位)

首先，我们可以知道 int 型数据的表示范围为 $-2^{31} \sim 2^{31}-1$ 。

通过分析可知，主要有三种情况:

a. 用二进制下科学计数法表示 int 型数时，尾数位数 ≤ 23 ，例如 0x00008001，此时将 0x8001 左移 $24-16=8$ 位得到 frac，而 exp 则为 $127+16-1$;

b. 当尾数位数 > 23 时，找到位数最末一位记作 x[i]，然后对尾数的舍去分 3 种情况考虑，并初始化 c=0:

当 $x[i-1]=1$ 且 $x[i-2]$ 、 $x[i-3] \cdots x[0]$ 都为 0，且 $x[i]=1$ ，令 $c=1$;

当 $x[i-1]=1$ 且 $x[i-2]$ 、 $x[i-3] \cdots x[0]$ 不都为 0，令 $c=1$;

其余情况 令 $c=0$;

c. 特殊情况。

对于 x 为 0 或为 0x8000 0000 的情况，在处理前进行特判即可。

③编程实现:

```
* 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 abs = x;
    unsigned s = 0x80000000 & x;
    unsigned tem = 0x40000000;
    unsigned exp_sign = 0;
    unsigned exp = 0;
    unsigned frac;
    unsigned c = 0;
    if (x == 0)
        return x;
    else if (x == 0x80000000)
        return (s + (158 << 23));
    else {
        if (x < 0)
            abs = -x;
        while (1) {
            if (tem & abs)
                break;
            tem = tem >> 1;
            exp_sign = exp_sign + 1;
        }
        frac = (tem - 1) & abs;
        if (31 - 1 - exp_sign > 23) {
            int i = 30 - exp_sign - 23;
            if ((frac << (31 - (i - 1))) == 0x80000000) {
                if ((frac & (1 << i)) != 0)
                    c = 1;
            } else if ((frac & (1 << (i - 1))) != 0)
                c = 1;
            frac = frac >> i;
        } else
            frac = frac << (23 - (31 - exp_sign - 1));
        exp = 157 - exp_sign;
        return (s + (exp << 23) + frac + c);
    }
}
```

(13) int float_f2i(unsigned uf);

①题目描述:

返回 unsigned uf 的整型数的二进制形式

参数和结果都会被作为 unsigned 返回，但是会表示为二进制的单精度浮点值
任何超过范围的数都应该返回 0x80000000u.

允许的操作符: 任何整数或者无符号数操作符包括: ||, &&. also if, while
最多操作符数目: 30

②大致思路: (exp 表示阶码位、frac 表示尾数位)

为了方便进行计算, 可以将 uf (unsigned float) 切割成符号位 s, 阶码 exp 和位数 frac, 分情况讨论:

- 当 $\text{exp}=0$ 或 $\text{exp}-127<0$ 时, 返回 0;
- 当 $\text{exp}-127\geq 31$ 时候, 超出表示范围, 于是返回 $0x80000000u$;
- 当 $\text{exp}-127\leq 23$, 根据符号位返回值 $\text{num}\gg(23-(\text{exp}-127))$

③编程实现:

```
/*
 * 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) {
    int exp = 0x7f800000 & uf;
    unsigned s = 0x80000000 & uf;
    int frac = 0x007fffff & uf;
    int frac2 = frac + 0x008fffff;
    int exp_sign = ((exp >> 23) - 127);
    if (exp == 0x7f800000)
        return 0x80000000u;
    else if (exp == 0)
        return 0;
    else if (exp_sign <= -1 && exp_sign >= -126)
        return 0;
    else if (exp == 31 && frac == 0 && s != 0)
        return 0x80000000u;
    else if (exp_sign >= 31)
        return 0x80000000u;
    else if (exp_sign <= 23)
        frac2 = frac2 >> (23 - exp);
    else
        frac2 = frac2 << (exp - 23);
    if (s == 0)
        return frac2;
    else
        return -frac2;
}
```

2、在 Linux 下测试以上函数是否正确，指令如下：

*编译：./dlc bits.c

*测试：make btest
./btest

完成代码编写后进行编译测试：

```
dongyunhao_2019284073@ubuntu:~/datalab-handout$ make  
gcc -O1 -Wall -m32 -lm -o btest bits.c btest.c decl.c tests.c
```

运行程序并进行测试：

```
dongyunhao_2019284073@ubuntu:~/datalab-handout$ ./btest
```

Score	Rating	Errors	Function
1	1	0	bitXor
1	1	0	tmin
2	2	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	float_twice
4	4	0	float_i2f
4	4	0	float_f2i
Total points: 37/37			

可以看到，代码全部正确

五、实验总结与体会

1、位操作比高级语言难得多

相对于高级语言，通过使用移位取反等位操作对数值进行操作要比高级语言进行运算难的多。而且在运算过程中需要注意更多的细节。位运算也需要对底层逻辑更清楚更明确。在实验过程中，通过上网查阅资料了解不熟悉的细节，我最终顺利完成了实验。

2、数学技巧可以起到另辟蹊径的作用：

在本次实验中，例如第一题等，可以利用离散数学中的真值表技巧，列出真值表对函数进行求解，这也是一种好方法。

3、不能忽略特判：

对于后三个题中，需要特别注意特判的情况，需要区别全 0 情况和全 1 情况进行分开讨论。对特殊值进行特判后不仅能提高代码的容错性，更能使程序逻辑更清晰。