一、实验目的

本实验目的是加强学生对位级运算的理解及熟练使用的能力。

二、报告要求

本报告要求学生把实验中实现的所有函数逐一进行分析说明，写出实现的依据，也就是推理过程，可以是一个简单的数学证明，也可以是代码分析，根据实现中你的想法不同而异。

三、函数分析

整数

```
Each "Expr" is an expression using ONLY the following:
  1. Integer constants 0 through 255 (0xFF), inclusive. You are
      not allowed to use big constants such as 0xffffffff.
  2. Function arguments and local variables (no global variables).
  3. Unary integer operations ! ~
  4. Binary integer operations & ^ | + << >>

You are expressly forbidden to:
  1. Use any control constructs such as if, do, while, for, switch, etc.
  2. Define or use any macros.
  3. Define any additional functions in this file.
  4. Call any functions.
  5. Use any other operations, such as &&, ||, -, or ?:
  6. Use any form of casting.
  7. Use any data type other than int.  This implies that you
     cannot use arrays, structs, or unions.
```

1. bitXor　要求：

函数名	bitXor
参数	int x, int y
功能实现	x ^ y
要求	~ &, 80

分析：纯纯的数字逻辑。

```
a ^ b = (a & ~b) | (~a & b) = ~~((a & ~b) | (~a & b))
    = ~((~(a & ~b)) & (~(~a & b)))
```

实现：

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

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

2. getByte 要求：

函数名	getByte
参数	int x, int n
功能实现	得到 x 的第 n 个字节（0 ~ 3，从最低位到最高位）
要求	! ~ & ^ | + « », 6

分析：数字右移 $8 \cdot n$ 位后用掩码提取最后 8 位二进制数。

实现：

```c
/*
 * getByte - Extract byte n from word x
 *   Bytes numbered from 0 (least significant) to 3 (most significant)
 *   Examples: getByte(0x12345678,1) = 0x56
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 6
 *   Rating: 2
 */
int getByte (int x, int y) {
  int tmp = n + n + n + n;
  int y = (((x >> tmp) >> tmp) & 0xff);
  return y;
}
```

3. logicalShift 要求：

函数名	logicalShift
参数	int x, int n
功能实现	x 逻辑右移 n 位（算术右移但无符号填充）
要求	! ~ & ^ | + « », 20

分析：此处的 sgn 和 nsgn 用于判断其符号，若 sgn 为 1（x 为负数）则最后结果只需要保留 neg 的，反之只需要 nneg 的。为了能够进行选择，此处的 ((~sgn) + 1) 和 ((~nsgn) + 1) 用以实现类似 MUX 的操作，比如，sgn 为 1，那么其取补码就变为 -1（即 0xffffffff），作为掩码，保留了之后应当留存的 neg，而 nsgn 就成了 0，nneg 也就为 0。

正数直接右移；负数先按有符号填充，算术右移对应位数，之后尝试处理前面多出来的符号位，通过**显而易见**的观察，得到 y 作为掩码（为什么要左移 1 位？略作尝试可以见得。顺带处理掉了右移 0 位的问题。），利用符号右移的特性，将多余的位数掩盖掉（这里是相当于保留右移后的那 1 个符号位跟往右的数）。

实现：

```c
/*
 * logicalShift - shift x to the right by n, using a logical shift
 *   Can assume that 0 <= n <= 31
 *   Examples: logicalShift(0x87654321,4) = 0x08765432
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 20
 *   Rating: 3
 */
int logicalShift (int x, int y) {
  // 右移默认会填充符号位。
  // 尝试直接得到符号位，1 为负，0 为正。
  // 必须 &1，否则负数的话会得到-1。
  int sgn = (x >> 31) & 1;
  int nsgn = !sgn;
  int y = ((1 << 31) >> n) << 1;// 原来，这不叫 integer constant
  //int sgn = (((x >> 31) << 30) >> n) << 1;
  int nneg = ((~nsgn) + 1) & (x >> n);
  int neg = ((~sgn) + 1) & ((~y) & (x>>n));
  return nneg+neg;
}
```

4. bitCount 要求：

函数名	bitCount
参数	int x
功能实现	得到 x 的二进制表示里 1 的个数
要求	! ~ & ^ | + « », 40

分析：定义五个掩码,

0101 0101 0101 0101 0101 0101 0101 0101 0x55555555 计算相邻位对的 1 的数量

0011 0011 0011 0011 0011 0011 0011 0011 0x33333333 每 4 位中计算 1 的数量

0000 1111 0000 1111 0000 1111 0000 1111 0x0F0F0F0F 每 8 位中计算 1 的数量

0000 0000 1111 1111 0000 0000 1111 1111 0x00FF00FF 每 16 位中计算 1 的数量

0000 0000 0000 0000 1111 1111 1111 1111 0x0000FFFF 整个 32 位整数中计算 1 的数量

通过特定的掩码筛选和位移，将问题规模缩小，同时将局部计算的结果累加到 sum 中。

实现：

```c
/*
 * bitCount - returns count of number of 1's in word
 *   Examples: bitCount(5) = 2, bitCount(7) = 3
```

```c
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 40
 *   Rating: 4
 */
int bitCount(int x) {
  // Reference 1:
  // Brian Kernighan
  // x & (x - 1) 会将最右的零消除掉。
  // 统计这样操作的次数。
  // 但是吧，这样也还会爆 Op 的啊。。。

  // Reference 2:
  // Hamming Weight
  // 我何德何能想得出来。。。
  int mask1, mask2, mask3, mask4, mask5;
  int sum;

  // 也考虑过 (x >> 1) & 1，但肯定爆炸。
  // 之前也一直考虑分治和掩码，但操作类似于
  // 0101 & 1110 != 0
  // 0101 & 1100 != 0
  // 0101 & 1000 == 0
  // 结果这样只能得到 howManyBits，而且还爆 Ops。
  // 想到的分治，也无非是类似打表，分成类似 4 个 Byte。
  // 每段与上掩码，一方面没有想好，另一方面求和也是老大难。

  // 预定义掩码
  // 俺何德何能
  mask1 = 0x55 | (0x55 << 8);
  mask1 = mask1 | (mask1 << 16); // 0x55555555

  mask2 = 0x33 | (0x33 << 8);
  mask2 = mask2 | (mask2 << 16); // 0x33333333

  mask3 = 0x0F | (0x0F << 8);
  mask3 = mask3 | (mask3 << 16); // 0x0F0F0F0F

  mask4 = 0xFF | (0xFF << 16); // 0x00FF00FF

  mask5 = 0xFF | (0xFF << 8); // 0x0000FFFF

  // 分阶段累加
  sum = (x & mask1) + ((x >> 1) & mask1);
  sum = (sum & mask2) + ((sum >> 2) & mask2);
  sum = (sum & mask3) + ((sum >> 4) & mask3);
  sum = (sum & mask4) + ((sum >> 8) & mask4);
  sum = (sum & mask5) + ((sum >> 16) & mask5);

  return sum;
}
```

5. conditional 要求：

函数名	getByte
参数	int x, int y, int z
功能实现	x ? y : z
要求	! ~ & ^ | + « », 16

分析：凡是非 0 的数，返回都为 0。

mux 用以判断是否 x 为 0，流程：

若 x != 0，则!x = 0，mux = 0，nzero = y，zero = 0，反之同理。

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) {
  // 非 0 的全都是 true!!!
  int mux = ((~(!x)) + 1);
  int zero =  mux & z;
  int nzero = ~mux & y;
  return zero + nzero;
}
```

6. tmin 要求：

函数名	getByte
参数	void
功能实现	32 位二补数最小値
要求	! ~ & ^ | + « », 4

分析：32 位 int 范围 $-2^{31} \sim 2^{31} - 1$

实现：

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

7. fitsBits 要求：

函数名	fitsBits
参数	int x, int n
功能实现	判断 x 是否能被 n 位（1 <= n <= 32）二补数表示
要求	! ~ & ^ | + « », 15

分析：首先采用和 `logicalShift` 类似的选择操作；

想法是，对于正数，比如 `0x000000ff`，若是能够被 9 位二补数表示，考虑到符号位的存在，其应当能够被 `0xffffff00` 这一 8 位的"掩码"所"湮灭"，也就是应当在 8 位的右移操作后能归零，采用"掩码"是为了方便正负数的分类。

对于负数，在 32 位二补数表示下，数字前会有大量的符号位，即便通过右移操作也只能得到 -1，即 `0xffffffff`。

某种化归的方法是，由于 n 位二补数表示的 int 范围是 $-2^n \sim 2^n - 1$，试图通过取补的操作转换成正数，但是会遇到 `0x80000000` 的障碍。

实现：

```
/*
 * fitsBits - return 1 if x can be represented as an
 *  n-bit, two's complement integer.
 *   1 <= n <= 32
 *   Examples: fitsBits(5,3) = 0, fitsBits(-4,3) = 1
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 15
 *   Rating: 2
 */
int fitsBits(int x, int n) {
  int sgn = (x >> 31) & 1;
  int nsgn = !sgn;
  // 原来负号不算 op，但是，怪，为什么没有判-1 非法。原来是没跑 dlc
  // dlc 蠢得很
  // 假如先左移再右移的话，正数可能变号。
  //int nneg = nsgn & !((x >> nsub1));
  //int neg = sgn & (!(((~(x>>n)))));
  int neg1 = ~0;//((~1)+1);

  // 憨了，n 不可能为 0!!!
  // 应该提前判断是否 n 为 0，为 0 必定不可表示。
  // 又不让用 if
  // 那就别让它溢出，n 要是 0 就给要右移的 (n - 1) 位加上 1。
  int shift = n + neg1; // + ((!n) & 1);
  int mask = neg1 << shift; // 之前老是忘记要减 1 的操作。
  // n = 0 时，mask = -2147483648[0x80000000]

  //int mask = (neg1 << n) >> 1;
  // n = 0 时，mask = -1[0xffffffff]
  // 是有误的操作，因为在左移 32 位后会变成 0，即便再右移动也没用。
```

```
  int nneg = nsgn & !(x & mask);
  int neg = sgn & !(~(x | (~mask)));
  // -2147483648[0x80000000]
  // 取补后仍为它本身。
  return nneg + neg;
}
```

8. dividePower2 要求：

函数名	dividePower2
参数	int x, int n
功能实现	x / (2 ^ n), 0 <= n <= 30
要求	! ~ & ^ | + « », 15

分析：考虑直接右移，正数直接得到结果；

负数的话，由于右移会自动向下取整，当末 n 位中有 1 时（也就是在右移过程中出现了奇数），会需要加 1（经过验证，一次就行）。可以见底下的注释。

注意，不能考虑先全部转换为正数，右移，再变号，0x80000000 取补会维持原样。

实现：

```
/*
 * dividePower2 - Compute x/(2^n), for 0 <= n <= 30
 *  Round toward zero
 *   Examples: dividePower2(15,1) = 7, dividePower2(-33,4) = -2
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 15
 *   Rating: 2
 */
int dividePower2(int x, int n) {
  int sgn = (x >> 31) & 1;
  //int nsgn = !sgn;
  int direct = (x >> n);
  // Old Code 3
  //int nneg = ((~nsgn) + 1) & direct;
  //int neg = ((~sgn) + 1) & (direct + odd);
  // Old Code 4
  //int mux = ((~sgn) + 1); // 这里可以砍，可见前几个代码处也可以删
  //int neg =  mux & (direct + odd);
  //int nneg = ~mux & direct;
  int mask = ~((~0) << n);
  int odd = (!(!(mask & x)));
  int y = direct + (odd & sgn);
  // Old Code 1
  //int leftmost = sgn << 31;
  //int comple = (~x)+1;
  //int neg = ((~sgn) + 1) & (((~(comple >> n))+1)|leftmost);
  // Old Code 2
```

```
  //int odd = x & 1;
  //int neg = ((~sgn) + 1) & ((x >> n) + (!(!n) & odd));
  // 猜想：末 n 位里如果有 1（也就意味着算数的中间过程中会出现奇数）
  // 奇数又会被强行向下取整，但是只需要加一回 1 就可以了。
  // 如何得到末 n 位里 1 的情况（这里只考虑负数）
  // (-2147483647[0x80000001] 假如直接 x 右移...
  // Gives -1073741824[0xc0000000]. Should be -1073741823[0xc0000001]
  // 右移对于正数是向下取整，对于负数也是向下取整取整。
  // 所以问题：30 >> 1 = 15;
  // -30 >> 1 = -15，但是-15 >> 1 = -8。
  // 一开始老是想要先换成正数再右移
  // -2147483648[0x80000000]
  // 不能表示成正数，会超限制。
  // 取补后仍为它本身。
  return y;
}
```

9. negate 要求：

函数名	negate
参数	int x
功能实现	- x
要求	! ~ & ^ | + « », 5

分析：取反，再加 1。(0x80000000 会维持原样)

实现：

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

10. howManyBits 要求：

函数名	howManyBits
参数	int x
功能实现	能表示 x 的最小二补数位数。
要求	! ~ & ^ | + « », 90

分析：通过二分查找来确定最高位 1 的位置。

首先将所有负数转换成正数，0x80000000 维持不变即可。

二分查找过程，以第一步（16 位区间）为例：

`b16 = !!(x >> 16) << 4;`：首先检查 `x` 的高 16 位中是否有 1。如果有，那么`!!(x >> 16)` 结果为 1，通过 `b16 << 4`（等于乘以 16 ）将其转换为数值 16，表示最高位 1 在高 16 位中（也是已经右移的位数），然后，`x` 被右移 `b16` 位，高 16 位移到了低 16 位。如果没有，则接下来相当于直接处理低 16 位。

之后对剩下的数依次进行 8 位、4 位、2 位、1 位区间的查找。

这一过程结束后，我们需要将 `b16`, `b8`, `b4`, `b2`, `b1` 求和，在二分过程中，比如最后一位为 1，`b1 = !!(x >> 1)` 的操作会直接将其抹除，我们必须多加一个 1，再者，即便结果是 0，也需要至少一位才能表示。

实现：

```c
/* 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) {
  // http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetNaive
  // 不对，以上链接给的是 bitCount
  // ~ 评价为做不出来 bitCount 就别想做对。~
  // 想错了，不是简单的 #0 + #1 就能做对
  // 正数右移 n 次后为 0，再 + 1。(以表示符号位)
  // 负数右移 n 次后变为 -1，最少也得有一位。

  // Claude 3 Sonnet 给的几乎是对的。
  int b16, b8, b4, b2, b1, b0;
  int sgn = x >> 31;
  x = (sgn & ~x) | (~sgn & x); // x = abs(x), except when x == 0x80000000

  // Binary Search
  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;
}
```

11. isLessOrEqual 要求：

函数名	isLessOrEqual
参数	int x, int y
功能实现	x <= y
要求	! ~ & ^ | + « », 24

分析：

首先判断二者是否相等，采用`!(x ^ y)`；

其次判断是否 x 为负, y 非负,(也即 x 最高位为 1, y 最高位为 0), 有 `(sgnx & (~sgny))`；

最后是二者同号的情形，可判断 `x - y` 的符号正负（用 `sgn` 表示），因为此时已经不会出现比如 `0x80000000` 取补后不变的问题，可以直接用 `x + ((~y)+1)` 来规避，得到 `(!(sgnx ^ sgny) & sgn)`。

以上三者或起来即得。

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) {
  // -2147483648[0x80000000]
  // 不能表示成正数，会超限制。
  // 还要考虑溢出。
  int sgnx = (x >> 31) & 1;
  int sgny = (y >> 31) & 1;
  int sum = x + ((~y)+1); // x - y
  int sgn = (sum >> 31) & 1;
  // 当 sgn == 0 时候，还需要判断一下 sum 是否为 0，即 !sum == 1
  // Warning: suggest parentheses around arithmetic in operand of |
  int leq =  (!(x ^ y)) | (sgnx & (~sgny)) | (!(sgnx ^ sgny) & sgn);
  return leq;
}
```

12. intLog2 要求：

函数名	intLog2
参数	int x
功能实现	log_2{x}
要求	! ~ & ^ | + « », 90

分析：通过类似 `howManyBits` 中的二分查找来进行，其实也是查找其中最高位的 1，区别在于这里 $\log_2 1 = 0$，所以不需要再加上 1。

实现：

```
/*
 * intLog2 - return floor(log base 2 of x), where x > 0
 *   Example: intLog2(16) = 4
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 90
 *   Rating: 4
 */
int intLog2(int x) {
  // http://graphics.stanford.edu/~seander/bithacks.html#IntegerLogObvious
  int b0 = 0x2;
  int b1 = 0xC;
  int b2 = 0xF0;
  int b3 = 0xFF << 8; // 0xFF00;
  int b4 = (b3 | 0xFF) << 16; // 0xFFFF0000;

  int r = 0; // result of log2(v) will go here
  int mask;
  int shift;

  // 以下形式上采取与 howManyBits 略有不同但实际一样的操作。
  // 32 位整数，检查高 16 位
  mask = !!(b4 & x);
  shift = mask << 4;
  x = x >> shift;
  r = r | shift;

  // 接下来检查高 8 位
  mask = !!(b3 & x);
  shift = mask << 3;
  x = x >> shift;
  r = r | shift;

  // 接下来检查高 4 位
  mask = !!(b2 & x);
  shift = mask << 2;
  x = x >> shift;
  r = r | shift;

  // 检查高 2 位
```

```c
  mask = !!(b1 & x);
  shift = mask << 1;
  x = x >> shift;
  r = r | shift;

  // 最后检查最高位
  mask = !!(b0 & x);
  r = r | mask;

  return r;
}
```

浮点数

```
You are expressly forbidden to:
  1. Define or use any macros.
  2. Define any additional functions in this file.
  3. Call any functions.
  4. Use any form of casting.
  5. Use any data type other than int or unsigned.  This means that you
     cannot use arrays, structs, or unions.
  6. Use any floating point data types, operations, or constants.
```

13. floatAbsVal 要求：

函数名	floatAbsVal
参数	unsigned uf
功能实现	abs(uf)
要求	10

分析：遵照 IEEE754 32-bit float 来返回 32 位无符号整数。

首先，NaN 要直接返回，于是考虑阶码全 0 和尾数非 0。

本题中，将 Inf 和 -Inf 视为扩展实数线上的满足特定代数性质的“数”，其绝对值的处理仍然将负号变为正号，有关 Inf 参见 Analysis I (Herbert Amann&Joachim Escher) 以及 IEEE Std 754-2019。

所以，非规格数、Inf、和规格化数一视同仁，该变号变号。

实现：

```c
/*
 * floatAbsVal - Return bit-level equivalent of absolute value of 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 representations of
 *   single-precision floating point values.
 *   When argument is NaN, return argument..
 *   Legal ops: Any integer/unsigned operations incl. ||, &&. also if, while
 *   Max ops: 10
```

```c
 *   Rating: 2
 */
unsigned floatAbsVal(unsigned uf) {
  // A zero value, null pointer value,
  // or null member pointer value is converted to false;
  // any other value is converted to true.
  // 32-bit
  // 1-bit 8-bit 23-bit
  // Inf can be abs
  // NaN just return
  // 这里用不用 unsigned 都没事。
  unsigned exp = (0xff << 23);
  unsigned frac = (1 << 23 - 1);
  unsigned neg = (1 << 31);
  if((exp & uf) == exp) // 抽象，之前用成了 &&
  {
    if((frac & uf) != 0)
      return uf;
  }
  return uf & ~neg; // 无穷等同“正常”（其实也包括非规格化）的数

  // 这是对的，思路不太一样。
  /*
  unsigned mask = (1 << 31) - 1;// 0x7FFFFFFF; // Mask to clear the sign bit
  unsigned minNaN = 0x7F800001; // Smallest NaN in positive space
  if ((uf & mask) >= minNaN) { // Check for NaN
    return uf; // If NaN, just return the original value
  }
  return uf & mask; // Clear the sign bit for absolute value
  */
  // 这也是对的，思路一样，之前把 && 写成了 & 所以的到了不同的结果。
  /*
  unsigned expMask = (0xff << 23); // 0x7F800000; // 阶码掩码，用于提取阶码部分
  unsigned fracMask =  (1 << 23 - 1); // 0x007FFFFF; // 尾数掩码，用于提取尾数部分
  unsigned signMask =  0x80000000; // 符号位掩码，用于提取符号位

  unsigned exp = uf & expMask; // 提取阶码部分
  unsigned frac = uf & fracMask; // 提取尾数部分

  // 检查是否为 NaN：阶码全为 1 且尾数不为 0
  if (exp == expMask && frac != 0) {
    return uf; // 如果是 NaN，直接返回原值
  }

  // 取绝对值：清除符号位
  return uf & ~signMask;
  */
}
```

14. floatScale1d2 要求：

函数名	floatScale1d2
参数	unsigned uf
功能实现	0.5 * uf
要求	30

分析： +0 -0 直接返回。

$$x \cdot \infty = \begin{cases} \infty, \ x > 0 \\ -\infty, \ x < 0 \end{cases}$$

负无穷类似，符号相反。所以我们在处理 Inf 和 -Inf 时，直接将其返回。

$$uf = 1.frac \ \cdot \ 2^{exp-127} = (1 + x_1 \cdot 2^{-1} + x_2 \cdot 2^{-2} + ... + x_{23} \cdot 2^{-23}) \ \cdot \ 2^{exp-127}$$

- 当 exp 大于 1 时，减半的操作可以直接通过右移阶码实现。
- 当 exp 恰好为 1 时，此时要实现从规格化数到非规格化数的转变。

$$uf = 1.frac \ \cdot \ 2^{1-127} = (1 + x_1 \cdot 2^{-1} + x_2 \cdot 2^{-2} + ... + x_{23} \cdot 2^{-23}) \ \cdot \ 2^{-126}$$

$$half_{unrounded} = 0.frac_{new} \cdot 2^{-127} = (1 \cdot 2^{-1} + x_1 \cdot 2^{-2} + x_2 \cdot 2^{-3} ... + x_{23} \cdot 2^{-24}) \cdot 2^{-126}$$

阶码减 1，常规操作；尾数右移 1，看似寻常；但是要考虑偶数舍入 !!!

考虑尾数的最后两位：

尾数末 2 位	舍入方式
11	向上舍入
10	不舍入
01	向下舍入
00	不舍入

总结：只需要考虑尾数末 2 位为 11 时, “进位”1。

不要忘了，前面减没了的阶码是”挪”到了尾数里。

- 当 exp 为 0，对非规格化数，右移，考虑偶数舍入即可。

实现：

```c
/*
 * floatScale1d2 - Return bit-level equivalent of expression 0.5*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 floatScale1d2(unsigned uf) {
  /*
  int exp_ = (uf&0x7f800000) >> 23;
  int s_ = uf&0x80000000;
  if((uf&0x7fffffff) >= 0x7f800000) return uf;
  if(exp_ > 1) return (uf&0x807fffff)|(--exp_)<<23;
  if((uf&0x3) == 0x3) uf = uf + 0x2;
  return ((uf>>1)&0x007fffff)|s_;
  */

  //uf = 8388607;
  unsigned expMask = (0xff << 23);
  unsigned fracMask = ((1 << 23) - 1); // 之前写成了 1 << 23 - 1
  unsigned sgnMask = (1 << 31);
  unsigned nsgnMask = ~sgnMask;
  unsigned exp = expMask & uf;
  unsigned frac = fracMask & uf;
  unsigned sgn = sgnMask & uf;
  unsigned nsgn = nsgnMask & uf;
  unsigned exp_0x01, mot_2Mask, mot_2, carry;

  if(nsgn == 0) // 注意运算结合优先级
  // ((nsgnMask & uf) == 0) 和 (nsgnMask & uf == 0) 是不一样的!!!
    return uf;

  // Inf ∞ 应该是要求如同扩展实数线里一般，满足种种特性。
  if(exp == expMask)
  {
    //if(frac)
      return uf;
  }
  // 需要考虑最小的非规格变成 0，直接减会变成 0!!!
  // 右移需要考虑舍入，向上？向下？偶数？这里采用了偶数舍入。
  exp_0x01 = (1 << 23);
  mot_2Mask = 3; // 00...11
  mot_2 = mot_2Mask & uf;
  carry = (mot_2 == mot_2Mask);

  if(exp > exp_0x01)
```

```c
  {
    exp -= exp_0x01;
  }
  else if((exp == exp_0x01))
  {
    exp -= exp_0x01;
    frac >>= 1;
    frac += carry;
    frac += (1 << 22);
    //return 1;
  }
  else
  {
    frac >>= 1; // 只用这条会有
    // ERROR: Test floatScale1d2(8388607[0x7fffff]) failed...
    // ...Gives 4194303[0x3fffff]. Should be 4194304[0x400000]
    frac += carry; // 考虑偶数舍入
    //return 4;
  }

  exp += frac;
  exp |= sgn;


  return exp;

}
```

15. floatFloat2Int 要求：

函数名	floatFloat2Int
参数	unsigned uf
功能实现	(int) float
要求	30

分析：无穷直接返回特定值。

+0 -0 直接返回 0（不能返回负 0 !!!）。

非规格化数清零。

NaN 视同无穷。

尾数加上前导 1。【注意，这样的话我们就可以将 frac 视同一个整数】

32 位整数的范围是 $-2^{31} \sim 2^{31}-1$，而 $2^{30} \cdot (1+1/2+1/4+...+1/2^{23}) < 2^{31}$

最大能表示的 E 为 30，则 exp 最大为 157，比这大的等同无穷。

比这小的需要面临另一个问题：尾数的最低位是 $1/2^{23}$

当 E 大于 23 时，可以将所有的尾数都乘以 2^E 转变为整数，由于 frac 已经被看作了一个整数直接左移 (E - 23) 位即可。

当 E 小于 23 时，右移 (E - 23) 位，因为截断所以不用考虑舍弃的位数。

不要忘记给 frac 加上符号。

实现：

```c
/*
 * 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
 */
int floatFloat2Int(unsigned uf) {
  unsigned expMask = (0xff << 23);
  unsigned fracMask = ((1 << 23) - 1);
  unsigned sgnMask = (1 << 31);
  unsigned exp = expMask & uf;
  unsigned exponent = (exp >> 23) - 127;
  unsigned frac = fracMask & uf;
  unsigned sgn = sgnMask & uf;

  if(exp == expMask)
    return 0x80000000u;
  if(exp == 0)
    return 0;

  // Integer Truncation makes life easy.
  // Bias = 127
  // exp_127 = 0111 1111 0...0
  // 若 exp < 0x7f 0...0
  // 那么最大也就是 2^(-1) * (1 + 1/2 + 1/4 + ... + 1/(2^23)) < 1
  if(exp < (0x7f << 23))
    return 0;
  // int -2^31 ~ 2^31 - 1
  // 2^30 * (1 + 1/2 + 1/4 + ... + 1/(2^23)) < 2^31
  // 157 = 128 + 29 = 128 + 32 - 3 = 127 + 16 + 8 + 4 + 1 = 10011101 = 9d
  // exp_157 = 1001 1101 0...0
  if(exp > (0x9d << 23))
    return 0x80000000u;
  // 加上前导 1
  frac = frac | 0x800000;
  // 接下来的操作类似于左移（阶码算出来与偏置作差）位。
  // e.g. 2^13 * (1 + 1/2 + 1/4 + ... + 1/(2^23)) 之后的是怎么办，直接不动了？
```

```c
  // 当正好为 2^23 的时候，相当于尾数就不用再动了，其他时候需要左移或右移。
  if(exponent > 23)
  {
    frac <<= (exponent - 23); // >>= 有等号！
  }
  else
  {
    frac >>= (23 - exponent);
  }
  // 以上其实都是操作的绝对值
  // 负数需要取补码
  if(sgn == sgnMask)
  {
    frac = (~frac) + 1;
  }
  return frac;
}
```

四、实验总结

对位运算以及部分 C 语言特性、和对浮点数的不熟悉，比如 ! 数次把我绕晕，比如 `if(-1)` 是怎么个机制，比如到底偶数舍入是怎么个舍入，比如浮点数截断是如何截断。

实验报告不好写，当初一些函数是试错法试出来的，想法尚未成型，有些则是查阅了资料，比如（其实没能用上的）Brian Kernighan's Algorithm, Hamming Weight，有些是算法未能与实际的操作相结合，比如具体怎么用二分查找。

以上的解决大都是写在了函数的分析与实现里，以下是部分参考资料。

至于建议，实验很好，难度很大，涉及面很广，但甚至有点好玩。主要是实验用时很长，没有实验课（这其实不重要），更重要的是学院方面的问题，ICS 课很好，老师很用心，但学院在"宽慰"时也只能声称是复习有用。一个编译原理，一个计算机系统导论，一个数据库系统，这三个实验就够天天忙活的了。于我个人，这课该早开的，而今消磨了激情，真只觉苦涩。

Negative Numbers 颠覆认知

Anything that is not `0` will be converted to `true`(*1 in the case of C*) a `zero` value will be converted to `false`(*0 in the case of C*).

https://stackoverflow.com/questions/18840422/do-negative-numbers-return-false-in-c-c

Integer Truncation https://stackoverflow.com/posts/11128755/timeline In case of casting a `float`/`double` value to `int`, you generally loose the *fractional* part due to **integer truncation**.

https://stackoverflow.com/questions/11128741/cast-variable-to-int-vs-round-function

https://stackoverflow.com/questions/24723180/c-convert-floating-point-to-int

Infinity IEEE Standard for Floating-Point Arithmetic

Analysis I (Herbert Amann&Joachim Escher)

Markdown & Pandoc & LaTeX Typora 中的 `\infin` 可以用，但 Zettlr 导出时不行；

Typora 和 Zettlr 中的 `gt`, `lt` 可以用，但 Zettlr 导出时不行。

提示均为 Undefined Control Sequence

https://tex.stackexchange.com/questions/257160/undefined-control-sequence-infty-f-i-right-sum-i-1-infy-underf

https://tex.stackexchange.com/questions/302554/why-am-i-getting-an-undefined-control-sequence-error-in-this-line

https://tex.stackexchange.com/questions/12519/what-library-do-i-have-to-use-such-that-the-document-can-render-lt-and-gt-as-l