Lab1.datalab

1.

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

分析: x ^ y = ~x&y | x&~y = ~(~(~x&y)&~(x&~y))

```
int bitXor(int x, int y) {
  // x ^ y = ~x&y | x&~y = ~(~(~x&y)&~(x&~y))
  return ~(~(~x&y)&~(x&~y));
}
```

2.

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

分析: 最小 32 位二进制补码为 0x80000000

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

3.

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

分析: 最大 32 位二进制补码设为 x

x = 0b0111⋯ x + 1 = 0b1000⋯

x ^ (x + 1) = 0xff⋯ 同时要求 x != 0xff⋯

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

4.

```
/*
 * 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
 */
```

分析：首先获得掩码 0xaaaaaaaa

返回与掩码相与是否等于掩码

```
int allOddBits(int x) {
  int a = 0xAA;
  int b = (a << 8) | a;
  int c = (b << 16) | b;
  return !((x & c) ^ c);
}
```

5.

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

分析：一个数按位取反加一得到其相反数

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

6.

```
/*
 * 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
 */
```

分析：判断第二个字节是 0x3 并且第一个字节在 0x1~0x9

```
int isAsciiDigit(int x) {
  int a = x >> 4; // a == 0x03
  int b = x & 0x0f;
  int c = b >> 3;
  return !(a ^ 0x03) & (!c | !(b ^ 0x09) | !(b ^ 0x08));
}
```

7.

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

分析: x == 0 -> return z & 0xff... | y & 0x00...

令 b == 0x00.. z & ~b | y & b

x != 0 -> return z & 0x00... | y & 0xff...

令 b == 0xff.. z & ~b | y & b

```c
int conditional(int x, int y, int z) {
  int a = !!x;
  int b = a << 31 >> 31; // if x == 0 : b = 0x00...
                         // if x != 0 : b = 0xff...
  return z & ~b | y & b;
}
```

8.

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

分析: x >= 0 且 y < 0 不满足

x < 0 且 y >= 0 满足

其余情况看 y – x 是否为正或 0

```c
int isLessOrEqual(int x, int y) {
  int a = y + (~x + 1); // a = y - x
  return (!(a >> 31) | !!(!(y >> 31) & (x >> 31))) & !((y >> 31) & !(x >> 31));
}
```

9.

```c
/*
 * 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
 */
```

分析:x == 0 -> 令 a = 0 -> ret = a + 1 = 1

x != 0 -> 令 a = -1 -> ret = a + 1 = 0

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

10.

```
/* 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
 */
```

分析：正数只需要看最高位的 1 所在的位数(从 1 计数) 再加上 1 位的符号位,负数直接取反然后按照正数一样处理

```
int howManyBits(int x) {
  int a = x >> 31;
  int b = a & (~x) | ~a & x;
  int c16 = !!(b >> 16) << 4;
  b >>= c16;
  int c8 = !!(b >> 8) << 3;
  b >>= c8;
  int c4 = !!(b >> 4) << 2;
  b >>= c4;
  int c2 = !!(b >> 2) << 1;
  b >>= c2;
  int c1 = !!(b >> 1);
  b >>= c1;
  return c16 + c8 + c4 + c2 + c1 + b + 1;
}
```

11.

```
/*
 * 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
 */
```

分析：特判 无穷 或 NaN，如果是直接返回传入参数

特判 非规格数，如果是直接保留符号，其余数位左移一位（由非规格数转化成规格化数的那一次左移也刚好可以满足要求

规格化数可以直接返回 exp + 1 的结果

```c
unsigned floatScale2(unsigned uf) {
  unsigned a = 0xff;
  unsigned flag = a << 23;
  if (!((uf & flag) - flag)){ // inf or NaN
    return uf;
  }
  if (!(uf & flag)){ // non-regular
    int sign = uf & (1 << 31);
    return sign | (uf << 1);
  }
  // regular
  return uf + (1 << 23);
}
```

12.

```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
 */
```

分析：特判 无穷 或 NaN，如果是直接返回 0x80000000u

特判 非规格数直接返回 0

否则根据 exp 对 frac 进行操作，同时保留符号位

```c
int floatFloat2Int(unsigned uf) {
  unsigned a = 0xff;
  unsigned flag = a << 23;
  if (!((uf & flag) - flag)){ // inf or NaN
    return 0x80000000u;
  }
  if (!(uf & flag)){ // non-regular
    return 0;
  }
  int frac = (uf & 0x7fffff) | (1 << 23);
  int exp = ((uf & 0x7f800000) >> 23) - 127;
  int sign = (uf >> 31) & 1;
  int u_res;
  if (exp < 0) u_res = 0;
  else if (exp < 23) u_res = frac >> (23 - exp);
  else if (exp >= 31) return 0x80000000u;
  else u_res = frac << (exp - 23);
  if (sign) return -1 * u_res;
  return u_res;
}
```

13.

```
/*
 * 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
 */
```

分析: x > 127： 去穷大

x < -149 : 0

x >= -126 && x <= 127 规格化数处理

否则非规格化数处理

```
unsigned floatPower2(int x) {
    if (x > 127) return 0xff << 23;
    else if (x < -149) return 0;
    else if ( x >= -126) return (x + 127) << 23;
    // x = -127  ->  1 << 22
    else return 1 << (149 + x);
}
```

14. result

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