目　录

一、实验目的

本次实验是对位操作级别的练习，本次报告是对实验中 bits.c 文件的再次说明，目的是帮助理解位级别的操作和表达以及培养同学们写代码之前先思考清楚思路的良好习惯，需要同学们认真完成。

二、实验说明

1. 实验环境

（1） 系统： Ubuntu 20.04.1 LTS 64-bit (VMware15)

（2） 编译工具：gcc version 9.3.0

2. 实验文件

（1） datalab-handout ：实验文件，含 bit.c, btest.c, makefile, test.h

（2） readme.pdf ：实验说明

三、函数分析

1、int bitAnd(int x, int y);

函数实现：

```
int bitAnd(int x, int y)
{
 return ~((~x)|(~y));
}
```

函数分析：函数要求是利用或运算和取反运算实现与运算，根据或运算的定义知有 1 即为 1，而全与运算为全 1 为 1，所以将 x，y 取反再进行或运算，只有全 0 才为 0，再取反即可得到 x&y 的结果。（亦可直接根据离散数学公式得出）

2、int bitNor (int x, int y);

函数实现：

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

函数分析：函数目的是用取反运算和与运算实现~(x|y)，此运算的意义在于x,y 对位只要有 1，结果就是 0，即找出全 0 对应位，返回 1。所以利用~x，~y进行与运算即可得到结果。

3、int copyLSB (int x);

函数实现：

```
int copyLSB(int x)
{
   return ~((x&1)+(-1));
}
```

函数分析：函数要求是将结果的所有位设为 x 的最小有效位，当 x 最小有效位是 1 时，则设为全 1，当 x 最小有效位是 0 时，则设为全 0。先将 x&1，则保留了 x 的最小效位，若为 0，则加上-1（全 1），则变成全 1；若为 1，则加上-1（全 1），则变成全 0，发现此时结果与预测结果恰好相反，所以取反即达到函数目的。

4、int evenBits(void);

函数实现：

```
int evenBits(void)
{
  int t1=(0x55<<8)+0x55;
  return ((t1<<16)+t1);
}
```

函数分析：函数要求是将结果的所有偶数位置 1，通过运算式实现的过程发现只要与 10101010 10101010 10101010 10101010 进行&运算即可得出结果，因为题目要求输入常数为-256~255，所以取 10101010（即 0x55），再将其拓展到 16 位，即 10101010<<8+10101010，同理再将其拓展为 32 位即可，最后进行与运算即可得到目的结果。

5、int logicalShift(int x, int y);

函数实现：

```
int logicalShift(int x, int n)
```

```
{
    int t1=x>>n;
    int t2= (1<<(32-n))+(-1);
    return t1 & t2;
}
```

函数分析：函数要求是利用算术右移的结果实现逻辑右移。算术右移在左端补了 n 个最高有效位的值，所以将左端右移的 k 个位全变为 0 即可。即 t2= 1<<(32-n) +(-1)，例如：00001 00000 +11111 11111 =00000 11111,同理得到目标值 t 0...0(k 个 0) 1... 1(w-k)个 1。最后进行 t1&t2，得出目的结果，实现了算术右移变成逻辑右移。

6、int bang(int x);

函数实现：

```
int bang(int x)
{
  int t1= (x|((~x)+1));
  return (~(t1>>31))&1;
}
```

函数分析：函数要求是不用!实现!x。首先是理解! 运算的意义，！是逻辑运算，当 x 为 0 时返回 1，当 x 不为 0 时返回 0，即区分出了 0 与其他数，通过位级表示发现 0 的所有位均为 0，而正数和负数的所有位中必有 1，至少有一位 1，所以突破点就在于这个 1 上。经过举例观察，发现对 x 取反并加 1 后，只要 x 上有一个 1 的位级，那么进行&或者|运算时必然会出现两个对应位全为 1，如果使用&的话，那么只会出现进位为 1，如果使用|，则发现最左侧进位及其左边全为 1，再注意一下 0 在此部分运算的表现，发现 0 先取反变成全 1，加上 1 后又变为全 0，进行或运算或者与运算后仍然还是全 0，这样就可以区分 0 与非 0 数的不同，从而实现！运算。通过举例测试，发现使用或运算更有利于我们下一步处理，即只要此数含 1 的位，那么处理后最高有效位就为 1，所以第二步则将 t1>>31,将所有位设为最高有效位，测试发现(~(t1>>31))&1 即可实现函数目的。

7、int leastBitPos(int x);

函数实现：

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

函数分析：函数要求是获取 x 最低位的 1 所在位置，通过上面第六题的分析知 x&（（~x）+1）即可获得 x 的最低位 1 的所在位置，举例测试发现结果恰好达到函数要求。

8、int tmax(void);

函数实现：

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

函数分析：函数要求是返回最大补码，即 01111111 11111111 11111111 11111111, 所以将 1 左移 31 位取反即可。

9、int negate(int x);

函数实现：

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

函数分析：函数要求是得到 x 的相反数，根据正负数转换及补码与原码的转换，同时结合 0 的特性，很容易得到~x+1 即可得

函数目的结果。

10、int isPositive(int x);

函数实现：

```
int isPositive(int x)
{
  return (!(!x))&(!(x>>31));
}
```

函数分析 ：函数要求是判断 x 是否为正数，很容易发现正数和 0 的第一位

都是 0，负数的第一位都是 1，那么利用!(x>>31)就可以区分负数与非负数，现在得出的结果中含有 0，所以就要剔除 0，由于 0 与！运算之间的特性，很容发现!！x 可以剔除 0，所以(!(!x))&(!(x>>31))实现了函数目的。

11、int isNonNegative(int x);

函数实现：

```
int isNonNegative(int x)
{
   return (!(x>>31));
}
```

函数分析：函数要求是实现判断负数与非负数，通过上面第十题的分析，很容易发现(!(x>>31))即可实现函数功能；

12、int sum3(int x, int y, int z);

函数实现；

```
static int sum(int x, int y)
{
   return x+y;
}
int sum3(int x, int y, int z)
{
   int word1 = 0;
   int word2 = 0;
   word1=(x^y)^z;
   int t1=(x&y);
   int t2=(x&z);
   int t3=(y&z);
   word2=(t1|t2|t3)<<1;

   return sum(word1,word2);
}
```

函数分析：函数要求是不借助加号实现三个数的加法。函数要求中给出了一个 sum 求和函数，由于实验要求显然不能直接调用 sum 函数实现，但是 sum 函数却给出了 word1，word2 两个数，这意味着在整个函数实现过程中，可以借助题设使用一次加法。通过理解数的加法过程，很容易意识到二进制加法等价于

进位加一个无进位的运算。通过举例观察，得到对于两个数的加法，设为 a,b，得到 a+b=a^b+(a&b)<<1 的结果，再利用这个公式拓展到三位数，则得到 x+y+z=(x^y+(x&y)<<1)^z+((x^y+(x&y)<<1)&z)<<1 的结果，很明显符合题目的要求，但是很难化简到仅剩一个加法，且加法是最后一位，显然这个思路不适合这个题目，或者这个三数之和是一种特例，与其他多数之和不一样。通过列出三个二进制加法的竖式运算，仔细观察会发现，sum 之和对应位级运算最多是 3 个 1 相加，所以它进位最多进 1，所以突破点就在这个进位仍然是 1 的现象上。也就是说只要三个数对应位中有两个或者三个 1，就会产生进位，这样三位数之和求类似于求两位数之和的位级运算上。只要三个数对应位中有两个或者三个 1，那么就会产生进位，对于三个二进制数的连续位级运算，拆分成两个二进制数的位级运算，所以只要 x,y,z 任意两个二进制数对应位级产生进位，总进位则加一，且不重复计数，所以令 t1=(x&y); t2=(x&z); t3=(y&z)，word2（表示进位）=t1|t2|t3, 同时 word1（表示无进位相加）=(x^y)^z,最后利用 sum（word1，word2）实现函数功能。

13、int addOK(int x, int y);

函数实现：

```
int addOK(int x, int y)
{
   int t1=x>>31;
   int t2=y>>31;
   int t3=(x+y)>>31;
   int p=(t1&t2)&(!t3);
   int q=(!t1&!t2)&(t3);
   return !(p|q);
}
```

函数分析：函数要求判断两个数相加是否溢出。有符号数溢出分为两种，正溢出和负溢出。正溢出即两数均为非负，和为负；负溢出即两数均为负，和为非负。令 p，q 分别为正溢出和负溢出，发生溢出时返回 1。判断负与正负利用符号位即可，最终进行!(p|q)运算即可以实现函数功能。

14、int logicalAbs(int x) ;

函数实现：

```
int logicalAbs(int x)
{
  int t1=(x>>31)^x;
    int t2=!!(x>>31);
    return t1+t2;
}
```

函数分析：函数要求是实现返回 x 的绝对值，对于非负数，绝对值对于其本身, 对于负数，进行（~x）+1 即可返回绝对值，但是这种方法涉及到判断符号位，不适合本题。所以需要寻找一种对于非负数无影响但是对负数有作用的运算，通过举例发现 x^(x>>31)对非负数返回值不变，但是对负数返回了一个比实际绝对值小 1 的数，所以仅需要再加上 1 即可，同时这个加 1 的运算对于非负数为 0，即可以取！！（x>>31）,t1+t2 即满足题目要求。

15、int isNonZero(int x);

函数实现：

```
int isNonZero(int x)
{
     int t1= (x|((~x)+1));
    return (t1>>31)&1;
}
```

函数分析：函数要求是实现判断 x 是否为 0。函数原理与上面的第六题几乎一样，先使用(x|((~x)+1))区分 0 与非 0 数的区别，再将 t1>>31,得到全 0 或者全 1，最后进行和 1 与运算即可实现函数功能。

四、特定题目分析

*被选中的题目是：*isNonZero

逐行分析：

```
int isNonZero(int x)
{
    int t1= (x|((~x)+1));       //区分 0 与非 0 的区别
/*  第一行代码实现的是区分 0 与非 0 数的区别，通过位级表示发现 0 的所有位
```

均为 0，而正数和负数的所有位中必有 1，至少有一位 1，所以突破点就在于这个 1 上。经过举例观察，发现对 x 取反并加 1 后，只要 x 上有一个 1 的位级，那么进行&或者|运算时必然会出现两个对应位全为 1，如果使用&的话，那么只会出现进位为 1，如果使用|，则发现最左侧进位及其左边全为 1，再注意一下 0 在此部分运算的表现，发现 0 先取反变成全 1，加上 1 后又变为全 0，进行或运算或者与运算后仍然还是全 0，这样就可以区分 0 与非 0 数的不同*/

return (t1>>31)&1;　　//将上一步的结果表示为 1,0 显示

/*通过举例测试，发现使用或运算更有利于我们下一步处理，即只要此数含 1 的位，那么处理后最高有效位就为 1，所以第二步则将 t1>>31,将所有位设为最高有效位,然后与 1 进行与运算即可实现函数功能*/

}

五、实验运行结果

```
jeremyhua@jeremy-vm:~/Desktop/datalab-handout$ make
gcc -O0 -lm -o btest btest.c bits.c
jeremyhua@jeremy-vm:~/Desktop/datalab-handout$ ./btest
ID      Score   Sum
1       100     100
2       100     100
3       100     100
4       100     100
5       100     100
6       100     100
7       100     100
8       100     100
9       100     100
10      100     100
11      100     100
12      100     100
13      100     100
14      100     100
15      100     100
total score/sum score: 1500 / 1500
Congratuations! You passed all puzzles.
jeremyhua@jeremy-vm:~/Desktop/datalab-handout$
```