CSAPP lab1 datalab实验报告

实验日期：2025,Mar.5-2025,Mar,19

1实验解题思路

1.bitXor

```
int bitXor(int x, int y) {
  /*分别将一个数字取反后与另一个数字与*/
  int a=(~x)&y;
  int b=x&(~y);
  /*将a和b相加后为结果，相加可通过取反后与，再取反实现*/
  return ~((~a)&(~b));
}
```

- a可以提取出**x为0但y为1**的位，b可以提取出**x为1但y为0**的位。
- 结果实际应为**a和b的和**，但不能使用加。`(~a)&(~b)`**后a和b均为0的位为1，a和b中有某一个为1的位为0（a和b不可能同时为1），再取反即为结果。**

2.tmin

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

- 补码最小值为负权重位（最高位）为1，其余皆为0的数。

3.isTmax

```
int isTmax(int x) {
  int reverse_x=~x;//tmax取反是tmin
  int neg_reverse_x=(~reverse_x)+1;//求reverse_x的相反数
  return !((neg_reverse_x^reverse_x)|(!reverse_x));
  //检查reverse_x是否是tmin（相反数是自身且不是0）
}
```

- 通过**tmax按位取反是tmin**来判断。tmin的特征是**tmin的相反数还是tmin自己。**
- 但相反数是自己的除了tmin还有0，所以**还要保证**`reverse_x`**不是0。**

4.allOddBits

```
int allOddBits(int x) {
  int mask=0xAA;//构造掩码
  int res=(x&mask)&((x>>8)&mask)&((x>>16)&mask)&((x>>24)&mask);
  return !(res^mask);
}
```

- 通过掩码来**提取每个字节的奇数位**
- 按位**与**运算来分别检验每个字节的奇数是否位均为1。若有某一字节中奇数位不等于1，则该字节 `x&mask` 将不等于mask，从而res不等于mask，即 `res^mask` 非0， `!(res^mask)=0` 。反之，若奇数位均为1，res应等于mask。

5.negate

```
int negate(int x) {
  return (~x)+1;/*补码中，相反数是取反加1*/
}
```

- 对于补码，因为x+(~x)=-1，所以-x=(~x)+1。

6.isAsciiDigit

```
int isAsciiDigit(int x) {
  int a=!((x>>4)^(0x3));//检查后四位以前的数是否都符合要求
  int last4=x&(0xF);//提取后四位
  int neg_last4=~last4+1;//取相反数
  int b=!((0x9+neg_last4)>>31);//判断后四位是否在范围 比较符号位
  return a&b;
}
```

- 检查输入的十六进制数是否在0x30和0x39范围内，即检查**后四位以前的数是否等于0x3**且**后四位数是否在0-9之间。**
- 判断后四位是否在0-9之间的方法是，后四位的**相反数+9是否大于等于0，可通过符号位是否为0判断。**

7.conditional

```
int conditional(int x, int y, int z) {
  /*若x为0，则!x是1，取反加一后是0xffffffff(111...111)
    若x非0，则!x是0，取反加一后还是0*/
  /*同样的数异或会抵消*/
  int check=!x;
  return y^((z^y)&((~check)+1));
}
```

- **参考**no-branch minimum**的方法，利用两个同样的数异或会互相抵消**，`r=(x<y)?x:y`可以用`r=y^((x^y) & -(x<y))`表示。
- x=0时，check=1，check的相反数（~check+1）为-1（0xffffffff），结果为`y^z^y=z`；x=1时，check=0，check的相反数仍为0，结果为y。

8.isLessOrEqual

```
int isLessOrEqual(int x, int y) {
  int sign_x=(x>>31)&1;
  int sign_y=(y>>31)&1;//取符号位 &1是因为负数会逻辑右移
  int neg_x=(~x)+1;//求x相反数
  int dif=y+neg_x;//y-x
  int sign_dif=(dif>>31)&1;//y-x的差的符号
  //考虑y-x会有溢出 分x和y同号/异号讨论
  return ((!(sign_x^sign_y))&(!sign_dif)) | ((sign_x^sign_y)&sign_x);//判断差的符号位
}
```

- 通过`y+(-x)`的符号来判断。
- **但`y+(-x)`可能会溢出**（在y为负数，x为正数时可能会负溢出；在y为正数，x为负数时可能会正溢出；x和y同号不会发生溢出），所以**分x和y同号或异号讨论。**
- `!(sign_x^sign_y)`为1时是**x和y同号**，通过y-x的符号`sign_dif`来判断；`(sign_x^sign_y)`为1时**x和y异号**，直接通过x的符号`sign_x`判断（若是此时`sign_x`为1，则x为负，y为正，x小于等于y成立，返回1）

9.logicalNeg

```
int logicalNeg(int x) {
  int neg_x=(~x)+1;////利用0是唯一 相反数与原数符号相同的数
  return ((x|neg_x)>>31)+1;
}
```

- 题目**本质是如何判断x是否为0。**
- 0的**相反数**还是自己，不过tmin的相反数也等于自己，所以不能用`x^(-x)`(仅判断相反数是否等于自己)来判断，而是应该用`(x|(-x)>>>31`判断原数和相反数符号为是否都为零来实现。
- 对于异或符号位后逻辑右移的结果，符号位均为0右移后仍未0【此情况应返回1】，**符号位有1逻辑右移后为-1**（补1）【此情况应返回1】，本题不能用逻辑非，右移结果+1刚好为所求答案。

10.howManyBits

```
int howManyBits(int x) {
  /*需要用到的mask*/
  int m1=~(((~1)+1)<<16);//0x0000ffff
  int m2=(m1<<8)^m1;//0x00ff00ff
  int m3=(m2<<4)^m2;//0x0f0f0f0f
  int m4=(m3<<2)^m3;//0x33333333
  int m5=(m4<<1)^m4;//0xaaaaaaaa
  /*处理符号*/
  int sign=x>>31;//负数为-1 正数为0
  x=x^((x^(~x))&sign);
  //no-branch minimum的方法处理负数 负数要变为~x 正数还是x
  /*统计最高位1的位数*/
  //拓展最高位1
  x|=x>>1;
  x|=x>>2;
  x|=x>>4;
  x|=x>>8;
  x|=x>>16;//x|=x>>k后可以使2k为1（幂数级复制式拓展）
  //统计1的个数（构造掩码 并行分治）
  x=(x&m5)+((x>>1)&m5);
  x=(x&m4)+((x>>2)&m4);
  x=(x&m3)+((x>>4)&m3);
  x=(x&m2)+((x>>8)&m2);
  x=(x&m1)+((x>>16)&m1);
  return x+1;//加1位 符号位
}
```

- 补码正数可以通过舍去最高位前多余的零（只保留一个符号位0）表示，补码负数可以通过舍去最高位0前多余的1（仅保留一个符号1）表示。
- 所以表示所需最少位数问题转化为**求补码正数的最高位1（负数的最高位0）在从低往高的第几位**，将**负数按位取反**后，问题统一为**求最高位1在第几位。**
- 通过将最高位1以后的位数都**拓展为1**，再**统计数中有几个1（构造掩码 并行分治）**，可求出最高位1的位数。
- 正数最高位1（负数最高位0）的位数**再+1符号位**就是表示所需最小位数。

11.floatScale2

```
unsigned floatScale2(unsigned uf) {
  unsigned int sign=uf&(1<<31);
  unsigned int exp=((0xff<<23)&uf)>>23;
  unsigned int frac=0x7fffff&uf;
  if(exp==255) return uf;//特殊值
  if(exp==0){//非规格化
    frac<<=1;
    if(frac>>23){//如果发生溢出
      exp=1;
```

```
      frac=frac&0x7fffff;//溢出的那一位（2的零次方位）舍掉 因为规格化后f的解释规则已加1
    }
  }else{//规格化
    exp+=1;//乘2
    if(exp==255) frac=0;//无穷大
  }
  return sign | (exp<<23) |frac;
}
```

- 根据浮点数IEEE的表示规则，**分别提取浮点数的符号位，阶数位，小数位**进行处理。
- 根据**阶数**情况进行条件分类。
  - 如果参数的 `exp=255` （特殊值），返回参数；
  - 如果参数的 `exp=0` （非规格化），小数部分乘2（左移1位），如果小数位发生溢出，则变为规格化数（解释规则也改变）；
  - 如果参数为规格化数，则直接指数+1（相当于乘2），若阶数达到255，则返回无穷大（exp=255，frac=0）。

12.floatFloat2Int

```
int floatFloat2Int(unsigned uf) {
  unsigned int exp=((0xff<<23)&uf)>>23;//提取阶数
  if(exp==255){ //注意exp是无符号整数 11..111为255 不能为-1
    return 0x80000000u;
  }else if(exp==0){
    return 0;//frac 本身是 23 位的值，除以 2^-126(右移126位) 之后肯定是 0，所以直接返回 0
  }else{
    int sign=(uf>>31)?-1:1;//符号位 //通过三目运算符将其转换为正/负因式
    unsigned frac=0x7fffff&uf;//提取小数
    unsigned F=frac|(1<<23);//补1 （注意不是直接+1） 1.f1f2...
    int E=exp-127;//减取bias //注意exp为无符号数，E为有符号数，中间有类型转换
    if(E>31){
      return 0x80000000u;//超过范围
    }else if(E<0){
      return 0;
    }
    if(E>23){
      return sign*(F<<(E-23));
    }else{
      return sign*(F>>(23-E));
    }
  }
}
```

- 首先**提取阶数**进行条件分类，**非规格化数（过小）以及特殊值无法用整数表示**，直接返回相应的结果。
- 对于规格化数，**提取符号位**并转换为相应正/负因式，**提取小数位**并补上frac在规格化数解释规则中要加的1（1.f1f2f3....），**提取阶数**并减去bias。然后根据IEEE的浮点表示规则，以E的大小分情况进行运算，有效位数超过/不到正数所能表示范围的直接返回，大于23的左移，小于23的右移（因为单精度浮点数的小数部分位数为23）。

2实验中遇到和解决的问题

2.1通过相反数进行判断时未考虑全面

- 问题：在 `3.isTmax` 和 `9.logicalNeg` 中，尝试通过 `tmin` （或 `0` ）的**相反数仍是自身**性质来检验数字是否为 `tmin` （或 `0` ），但忽略了 `0` （ `tmin` ）也符合该性质。在用 `.\btest` 进行测试时发现输入为 `0` （ `tmin` ）时出现错误。
- 解决：在 `3.isTmax` 的结果判断时**增加了"还需不为0"的条件**： `!((neg_reverse_x^reverse_x)|(!reverse_x))` ；在 `9.logicalNeg` 中，**调整判断条件为相反数与原数符号位相同**，排除了 `tmin` 的误判。

2.2右移时未考虑逻辑右移/算术右移

- 问题：在 `8.isLessOrEqual` 提取 `x` 和 `y` 的符号位时，原先使用 `x>>31` 的方式直接右移，这样对于正数没问题，但因为负数采用逻辑右移（补1），**符号位提取后结果为-1，而不是所期望的1**，导致在符号间异或判断时出现错误。
- 解决：使用 `(x>>31)&1` ，右移后与1取并，排除高位的干扰。
- 启发：意识到负数的逻辑右移后，利用逻辑右移也可以更方便的解决一些题目，比如 `9.logicalNeg` 中，若是x非0， `x|neg_x` 最高为1（负数），逻辑右移后位-1，+1后刚好可以返回结果为0；类似地，在 `10.howManyBits` 中， `sign=x>>31` 负数 `sign=-1` ，正数 `sign=0` ，刚好适合使用no-branch minimum的方法将负数按位取反： `x=x^((x^(~x))&sign)` 。

2.3边界溢出问题

- 问题： `8.isLessOrEqual` 中，试图直接通过 `y-x` 的符号位来判断是否小于等于，但测试后发现当x和y异号时可能发生**正/负溢出**，导致结果错误。
- 解决：溢出仅可能发生在x和y异号时，**分情况讨论。**异号时直接判断负数小于正数，同号时按原先的 `y-x` 的差的符号判断。

3实验最终结果截图

```
wcx@LAPTOP-OMTCB5PG:~/lab1/datalab-handout$ ./btest
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
ERROR: Test floatPower2(0[0x0]) failed...
...Gives 2[0x2]. Should be 1065353216[0x3f800000]
Total points: 32/36
```