~/Downloads/CSAPP之详解DataLab.md

实验1—DataLab

实验材料

1. 一个能够运行的Linux系统或Unix
2. 下载好make
3. 能够运行32位程序的gcc
4. 官网的实习手册

实验要求

整数编码规则:

将每个函数中的“return”语句替换为一个或实现该函数的多行C代码。你的代码必须符合以下风格:

```
int Funct(arg1, arg2, …){
    /*简单描述你的实现如何工作*/
    int var1 = Expr1;
    …
    int varM = ExprM;
    varJ = ExprJ;
    …
    varN = ExprN;
    return ExprR;
}
```

每个"Expr"都是只使用以下语句的表达式:

1. 整数常量0到255 (0xFF)，包括。你是不允许使用像0xffffffff这样的大常量。

2. 函数参数和局部变量(没有全局变量)。

3.一元整型运算 !~

4. 二进制整数运算& ^ | + << >>

有些问题进一步限制了允许的运算符的集合。每个“Expr”可以包含多个运算符。你不不会被限制每行一个操作符。您被明确禁止:

1. 使用任何控件结构，如if、do、while、for、switch等。

2. 定义或使用任何宏。

3. 在此文件中定义任何附加函数。

4. 调用任何函数。

5. 使用任何其他操作，如 &&，||，-，或 ?: 。

6. 使用任何形式的对象转换。

7. 使用除int以外的任何数据类型，这意味着你不能使用数组、结构体或联合。

你可以假设你的机器:

1. 使用二进制补码，int类型为32位表示。

2. 使用算术右移。

3.在转换时可能会发生溢出

浮点编码规则对于需要实现浮点运算的问题，编码规则不那么严格。你可以使用循环和有条件的控制。你可以同时使用整数和无符号。您可以使用任意整数和无符号常量。你可以使用任何算法，对int或unsigned数据进行逻辑或比较操作。

但您被明确禁止:

1. 定义或使用任何宏。

2. 在此文件中定义任何附加函数。

3. 调用任何函数。

4. 使用任何形式的对象转换。

5. 使用除int或unsigned以外的任何数据类型。这意味着你不能使用数组、结构体或联合。

6. 使用任何浮点数据类型、操作或常量。

实验规则

你需要通过修改bits.c中的函数来完成实验。每次修改后，你都需要键入下面两条指令来检查正确性，这意味着你需要安装好make。

```
make clean
make btest
```

然后，你可以键入下面这条指令来输出结果

```
./btest
```

2.62 DataLab

image.png

题目一：只使用 ~ 和 & 实现 ^

题目代码与解答:

```
/*
 * 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);
}
```

解答：

根据布尔代数，代码~(x & y)实现的是只要不同时为1则为1，只要我们除掉同时为0取1的情况便实现了异或，而代码**~x & ~y实现的是只有同时为0则为1，则~(~x & ~y)**实现的是同时为0则为0，这样就排除掉了同时为0取1的情况。

题目二：返回最小的补码

题目代码与解答：

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

解答：

0x80000000即为最小的补码。

题目三：判断是否为补码的最大值

题目代码与解答：

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

解答：

补码的最大值为0x7fffffff，如果x是该值的话，那么x + x + 1应该等于0xffffffff，按位取反则为0，逻辑运算中，0为假，非0为真，逻辑非取逻辑反，则我们应该返回 !(~(x+x+1))，它在x = 0x7fffffffff时返回1，而在其它时候返回0，不过请注意，当x = 0xffffffff时，它仍然会返回1，所以我们需要特判这个数。

题目四：判断补码所有的奇数位是否都为1

题目代码与解答：

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

解答:

奇数位全为1的掩码为0xAAAAAAAA，我们需要利用移位技巧构造这个掩码，然后我们将x的偶数位置为0，最后再判断它是否等于0xAAAAAAAA即可，判断x == y可以采用!(x ^ y)来实现，它利用了异或的性质。

题目五：不用负号实现 -x

题目代码与解答:

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

解答:

这个题目很简单，我们返回它的按位反再加一即可。因为按位取反的数与原来数相加为0xffffffff，再加1即等于0。

题目六：判断x是否为ASCII码

题目代码与解答:

```
/*
 * 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) {
  int a = x + (~0x30 + 1);
  int b = 0x39 + (~x + 1);
  int c = 0x1 << 31;
  return !(a & c) & !(b & c);
}
```

解答:

这个题目要求我们判断x是否大于等于0x30且小于等于0x39，我们可以推出这样两个不等式，① x + (- 0x30) >= 0；2②0x39 + (- x) >= 0。我们只需要比较最终结果是否大于等于0就可以了，这等同于符号位是否为0，这可以通过 & 0x80000000得出，如果符号位为0，& 0x80000000的结果一定是0，取逻辑反则为1；反之，如果符号位为1，取逻辑反的结果应该为0。

我们要思考一下加法溢出是否会对结果造成影响，如果第一个等式发生溢出，则有 x - 0x30 < -2^{32}，那么x < -2^{32}+ 0x30，如果x = Tmin，那么-x = Tmin，则 0x39 + (-x) = 0x39 + Tmin < 0，这不满足不等式②；如果x != Tmin，那么有 -x > 2^{32} - 0x30，那么有0x39 + (-x) = 0x39 + 2^{32} - 0x30 = 2^{32} + 0x9，这发生了正溢出，因此它的结果会小于0，同样不满足等式②，因此第一个等式溢出并不会影响答案。第二个等式溢出可同理分析。

题目七：实现表达式 x ? y : z

题目代码与解答：

```
/*
 * 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) {
  int a = !!x;
  int b = (a << 31) >> 31;
  return (y & b) | (z & ~b);
}
```

解答：

分析代码，如果x真则a = 1，b = 0xffffffff，y & b则置y为原数，z & ~b则值z为0，那么 (y & b) | (z & ~b)返回的便是y；如果x假则a = 0，b = 0，y & b则置y为0，z & ~b则值z为原数，那么 (y & b) | (z & ~b)返回的便是z

题目八：x <= y

题目代码与解答：

```
 * 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 s = y + (~x + 1);
  int m = 1 << 31;
  int a = !!(x & m);
  int b = !!(y & m);
  int c = !!(a ^ b);
  int f = s & m;
  return (c & !b) | (!c & !f);
}
```

解答：

这道题可以采用y + (- x) >= 0解答，但我们必须要保证它没有溢出。如果x与y异号，此时如果y为非负则返回1，否则返回0；如果x与y同号，我们判断y + (-x)是否大于0即可。代码中a,b,f计算x,y,s的符号，而c判断x与y是否异号。

题目九：不使用！实现 !x

题目代码与解答:

```
/*
 * 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 | (~x + 1)) >> 31) + 1;
}
```

解答:

对于x而言，除了0和Tmin之外，x 与~x + 1互为相反(符号位相反)数，这意味着肯定有一个数为负，即符号位为1，将一个符号位为1的数逻辑右移31位后会得到0xffffffff，加1则为0；对于Tmin而言，~x + 1 = Tmin，这说明它们的符号位也为1，即答案也会是0；对于0而言，x 与~x + 1的符号位都为0，这意味着最终的结果会是1。

题目十：一个数用补码表示最少需要几位

题目代码与解答:

```
/* 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 s = x >> 31;
        x = x ^ s;
        int b16 = !!(x >> 16) << 4;
        x = x >> b16;
        int b8 = !!(x >> 8) << 3;;
        x = x >> b8;
        int b4 = !!(x >> 4) << 2;
        x = x >> b4;
        int b2 = !!(x >> 2) << 1;
        x = x >> b2;
        int b1 = !!(x >> 1) << 0;
        x = x >> b1;
   return b16 + b8 + b4 + b2 + b1 + x + 1;
```

```
}
```

解答：

不断缩小范围即可，b16表示32位中，高16为是否有一，如果有b16 = 1 << 4 = 1，那么x就会右移16位，把返回缩小到高16位到高32位这个范围，如果没有，则把范围缩小到0位到高16位这个范围(二分？)，如此反复，请不要忘了加上符号位！

题目十一：求2乘以一个浮点数

题目代码与解答：

```
/*
 * 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
 */
unsigned floatScale2(unsigned uf) {
    int exp = (uf >> 23) & 0xff;
    int sign = (uf & (1 << 31));
    int inf = 0x7f800000 | sign;
    if (exp == 255)
      return uf;
    if (exp == 0)
      return (uf << 1) | sign;
    exp++;
    if (exp == 255)
      return inf;
    return (uf & 0x807fffff) | (exp << 23);
}
```

解答： 一个数×2，相当于左移一位。

这个题目要求我们熟悉浮点数的表示，先取出uf的阶码域exp，如果这个数是一个特殊值，我们什么也不做，直接返回；如果这个数是非规格化数，得益于非规格化数到规格化数的平滑转变，我们直接将uf左移一位即可，由于移除了符号位，请不要忘记添加上来；如果这个数是规格化的数，我们便将阶码加1，如果此时阶码 = 255，我们返回无穷大，否则的话，我们更新阶码并返回。

题目十二：将float转换成int

题目代码与解答：

```
/*
 * 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 (incleuding 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) {
        int exp = (uf >> 23) & 0xff;//阶码域
        int sign = uf & (1 << 31); //符号位
        if (exp == 255)
                return 0x80000000;
        if (exp == 0)
                return 0;
        int k = exp - 127;   //真实指数
        int f = (uf & 0x007fffff) | 0x00800000;//小数域23位，加一个1
        if (k < 0)
                return 0;
        if (k > 31)
                return 0x80000000;
        if (k < 23)
                f >>= (23 - k);
        else
                f <<= (k - 23);
        if (sign)
                return ~f + 1;
        else
                return f;
}
```

解答： 由小数域右移实际指数加上正负得到整数表示。

先考虑特殊情况，当exp == 255或exp == 0时，此时表示特殊值和非规格化的值，我们按要求返回。否则计算出阶码值k和小数域f，请注意这里f是需要多加一个1的，如果k < 0的话，我们返回0；如果k > 31的话，小数域由于规格范，肯定大于1，整数为1 * 2^k，此时一定发生了溢出，返回题意要求的数。其他情况下，我们根据k的值进行移位，可以设浮点数 f = [0000,0000,1 X1 X2,.......X23]，它的真实值以二进制表示为[00000000000001.X1 X2,.......X23]。

如果k < 23的话，意味着并不是所有的小数都能移位至整数部分(有23位，只能移动k位)，则我们必须要舍去剩下的(23 - k)的小数值，所以选择f右移23-k位舍掉23-k位，例如k = 20，二进制小数[00000000000001.X1 X2,.......X23]右移20位得到T[1 x1 x2....x20]，还有3位未能移过来，与f[0000,0000,1 X1 X2,.......X23]比较，我们必须把f右移3(23 - k)位，这样f才能与整数t相同。**因此我们构造f必须多加一个1.**

如果k >= 23，说明所有的小数都可以移动到整数部分，如果k = 23的话，这正是整数[0000,0000,0 X1 X2,.......X23]的位表示；如果k > 23，我们仍需向左移动k-23位。

最后，别忘了它的正负。

题目十三：计算2.0^{x}

题目代码与解答：

```
/*
 * 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
 */
unsigned floatPower2(int x) {
        int e = x + 127;
        if (e >= 255)
                return 0x7f800000;//0b 0111,1111,1000,0000--INF
        if (x < 0)      //写成e<=0更好理解
                return 0;
        return e << 23;
}
```

解答:

我们可以计算1.0 * 2^x来达到这个目的，先排除掉几个值之外，我们将x + Bias作为阶码域返回即可，此时规格化的值会默认小数为1.0。

这一题学校电脑超时了，而自己电脑却可以通过，可以键入`./btest -T 20`修改评测时间为20ms或更多。

指数加上偏移127得到阶码域e，如果e全为1(255)，即这个数是无穷大INF，如果e为0，则这个数是无穷小，返回0，其他情况下是一个正确的数，将阶码域移到正确的位置。