

CSAPP:datalab详解

本文介绍CSAPP中datalab各小题的解题步骤

Int and boolean algebra

bitXor

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

题目要求: 用&和~实现^

思路: 异或运算，即x与y不同时结果为1， $(\sim x) \& y$ 能够使得结果中x为0，y为1的情况，而 $x \& (\sim y)$ 则处理x为1，y为0的情况，两结果的并即可实现 x^y

tmin

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

题目要求: 求最小的二进制补码int

思路: 直接用 $1 \ll 31$ 即可

isTmax

```

/*
 * 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));
}

```

题目要求: 若 $x=0x80000000u$ 则返回1, 否则返回0

思路: 观察 $0x80000000u$ 的特点, 只有首位为1, 且左移一位后就变为0, 注意到左移1位后为0除 $0x80000000u$ 外只有0, 故再排除0即可

allOddBits

```

/*
 * 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(0xFFFFFFFF) = 0, allOddBits(0xAAAAAAAA) = 1
 * Legal ops: ! ~ & ^ | + << >>
 * Max ops: 12
 * Rating: 2
 */
int allOddBits(int x) {
    int mask = 0xAA+(0xAA<<8);
    mask=mask+(mask<<16);
    return !((mask&x)^mask);
}

```

题目要求: 若参数x的奇数位都是1则返回1, 否则返回0

思路: 先构造一个奇数位全部为1的 $mask=0xAAAAAAAA$, 然后x与mask做与运算, 当且仅当x奇数位均为1时, $x \& mask = mask$, 所以只有x奇数位均为1时, $x \& mask$ 与mask的异或为0, 再取反即可完成

negate

```

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

```

```
return (~x) + 1;
}
```

题目要求: 返回参数的相反数

思路: 取反+1即可

isAsciiDigit

```
/*
 * 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) {
    return !((x + (~0x30) + 1) >> 31) & !((0x39 + (~x) + 1) >> 31);
}
```

题目要求: 若x是处于0-9之间的ASCII码(也就是0x30-0x39)则返回1, 否则返回0

思路: 由上一轮的取反得到灵感, 本题中需要满足 $x - 0x30 \geq 0 \wedge 0x39 - x \geq 0$, 判断两式首位符号位是否为0即可确定两式均大于等于0

conditional

```
/*
 * 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) {
    x = !!x;
    x = ~x + 1;
    return (x & y) | (~x & z);
}
```

题目要求: 用给定运算符完成等同于三目运算符?:的作用

思路: 注意到关键点: -1的补码为0xFFFFFFFF, 0的补码为0x00000000, 且它们相互互为反码, 所以对任意int x, 有 $x \& (-1) = x$; $x \& 0 = 0$, 所以对x调整, 使得x为0的时候, x可取z, x非0则取y, 利用上述

性质，先将任意非0的x调整为1，然后取-1，这两部操作对为0值的x则没有变化，然后利用此性质取最后一式即可

isLessOrEqual

```
/*
 * 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 b1 = x >> 31, b2 = y >> 31;
    x = x + ~(1 << 31) + 1;
    y = y + ~(1 << 31) + 1;
    int NotbitXor = !(b1 ^ b2);
    //printf("%d %d\n", !!(b1 ^ b2), !!(b1 & (!b2)));
    return (!NotbitXor) & (b1 >> 31) | NotbitXor & (!(y + (~x) + 1) >> 31 & 1);
}
```

题目要求: 给定int参数x, y: 若 $x \leq y$, 则返回1, 否则返回0

思路: 最朴素的思路是 $y - x \geq 0$, 然后判断结果的符号位是否为0, 但是会出现问题在于当x, y均为负数时可能会发生溢出, 导致结果出现偏差, 所以要对两个参数的首位符号位进行分类讨论: 若 $x[31] = 1$, 且 $y[31] = 0$, 时可直接判断; 若 $x[31] = 1$, 且 $y[31] = 1$, 时, 比较剩余31位的大小, 有 $y[30:0] \geq x[30:0]$, 时结果为1; 而若 $x[31] = 0$, 且 $y[31] = 0$, 时, 比较剩余31位的大小, 有 $y[30:0] \leq x[30:0]$, 时结果为1, 而这两种不同情况可以用同一个表达式判断, 这样就在Max ops内得到了结果

logicalNeg

```
/*
 * 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;
}
```

题目要求: 用所给符号实现!运算符

思路: 同conditional一题中得到的性质, 只有0本身和其相反数的符号位均为0, 利用此性质我们可以取x本身和-x的或的符号位(通过左移), 这样若符号位非0, 则取值为-1, 再加1即可得到正确结果

howManyBits

```

/* 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 b16,b8,b4,b2,b1,b0;
    int flag=x>>31;
    x=(flag&~x)|(~flag&x);
    b16=!!(x>>16) <<4;
    x>>=b16;
    b8=!!(x>>8)<<3;
    x >>= b8;
    b4 = !(x >> 4) << 2;
    x >>= b4;
    b2 = !(x >> 2) << 1;
    x >>= b2;
    b1 = !(x >> 1);
    x >>= b1;
    b0 = x;
    return b0+b1+b2+b4+b8+b16+1;
}

```

题目要求: 在90个运算符内实现计算参数x的位数的功能

思路: 本题采用二分法的思想简化步骤，由题目逻辑，可将参数取绝对值(该操作对该数的最小位数表示的数值未进行改变)，然后寻找第一个1，再加上一表示符号位要占用1位即可

Float

IEEE Float Standard

floatScale2

```

/*
 * 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 = (0x7f800000 & uf) >> 23;
    int sign = uf & 0x80000000;
    if (exp == 0)
        return (uf << 1) | sign;
    if (exp == 255)
        return uf;
    exp++;
    if (exp == 255)
        return (0x7f800000 | sign);
    return (uf & 0x807fffff) | (exp << 23);
}

```

floatFloat2Int

```

int floatFloat2Int(unsigned uf) {
    int sign = uf >> 31;
    int exp = ((uf & 0x7f800000) >> 23) - 127;
    int frac = (uf & 0x007fffff) | 0x00800000;
    if (exp > 31) return 0x80000000;
    if (exp < 0 || !(uf & 0x7fffffff)) return 0;
    if (exp > 23) frac = frac << (exp - 23);
    else frac = frac >> (23 - exp);
    if (!(frac >> 31 ^ sign)) return frac;
    else if (frac >> 31) return 0x80000000;
    else return -frac;
}

```

floatPower2

```

/*
* 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) {

```

```

if (x > 127)
    return 0x7f800000;
else if (x < -127)
    return 0x0;
else
    return (x + 127) << 23;
}

```

测试结果

The screenshot shows a Zya-VirtualMachine terminal window. The left pane displays the source code of a C program, and the right pane shows the output of the btest tool.

Source Code (Left Pane):

```

290     return (uf << 1) | sign;
291     if (exp == 255)
292         return uf;
293     exp++;
294     if (exp == 255)
295         return (0x7f800000 | sign);
296     return (uf & 0x807fffff) | (exp << 23);
297 }
298 /*
299  * floatFloat2Int - Return bit-level equivalent
300  * for floating point argument f.
301  * Argument is passed as unsigned int, but
302  * it is to be interpreted as the bit-level
303  * single-precision floating point value.
304  * Anything out of range (including NaN and
305  * 0x80000000).
306  * Legal ops: Any integer/unsigned operation
307  * Max ops: 30
308  * Rating: 4
309  */
310 int floatFloat2Int(unsigned uf) {
311     int sign = uf >> 31;
312     int exp = ((uf & 0x7f800000) >> 23) - 127;
313     int frac = (uf & 0x007fffff) | 0x00800000;
314     if (exp > 31) return 0x80000000;
315     if (exp < 0 || !(uf & 0x7fffff)) return 0;
316     if (exp > 23) frac = frac << (exp - 23);
317     else frac = frac >> (23 - exp);
318     if (!(frac >> 31 ^ sign)) return frac;
319     else if (frac >> 31) return 0x80000000;
320     else return -frac;
321 }
322 /*
323  * floatPower2 - Return bit-level equivalent of the expression 2.0^x
324  * (2.0 raised to the power x) for any 32-bit integer x.
325  *
326  * The unsigned value that is returned should have the identical bit
327  * representation as the single-precision floating-point number 2.0^x.
328  * If the result is too small to be represented as a denorm, return
329  * 0. If too large, return +INF.
330  *
331  * Legal ops: Any integer/unsigned operations incl. ||, &&. Also if, while
332  * Max ops: 30

```

Terminal Output (Right Pane):

```

1 & 1));
|
btest.c: In function 'test_function':
btest.c:332:23: warning: 'arg_test_range[1]' may be used uninitialized in this f
unction [-Wmaybe-uninitialized]
332 |         if (arg_test_range[1] < 1)
    |
^[[Azya1412@ubuntu:~/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
4 4 0 floatPower2
Total points: 36/36
zya1412@ubuntu:~/datalab-handout$

```