code/data/bits.c

Problem 2.66 Solution:

The key idea is given in the hint. We can create a cascade of 1's to the right—first one, then two, then four, up to half the word size, using just 10 operations.

code/data/bits.c

```
/*
 * Generate mask indicating leftmost 1 in x.
 * For example 0xFF00 -> 0x8000, and 0x6600 --> 0x4000
 * If x = 0, then return 0.
 */
int leftmost_one(unsigned x) {
    /* First, convert to pattern of the form 0...011...1 */
    x |= (x>>1);
    x |= (x>>2);
    x |= (x>>4);
    x |= (x>>8);
    x |= (x>>16);
    /* Now knock out all but leading 1 bit */
    x ^= (x>>1);
    return x;
}
```

code/data/bits.c

Problem 2.67 Solution:

This problem illustrates some of the challenges of writing portable code. The fact that `1<<32` yields 0 on some 32-bit machines and 1 on others is common source of bugs.

A. The C standard does not define the effect of a shift by 32 of a 32-bit datum. On the SPARC (and many other machines), the expression `x << k` shifts by $k \bmod 32$, i.e., it ignores all but the least significant 5 bits of the shift amount. Thus, the expression `1 << 32` yields 1.

B. Compute `beyond_msb` as `2 << 31`.

C. We cannot shift by more than 15 bits at a time, but we can compose multiple shifts to get the desired effect. Thus, we can compute `set_msb` as `2 << 15 << 15`, and `beyond_msb` as `set_msb << 1`.

Problem 2.68 Solution:

Here is the code:

code/data/bits.c

```
/*
```

```
 3  * 0 otherwise
 4  * Assume 1 <= n <= w
 5  */
 6 int fits_bits(int x, int n) {
 7     /*
 8      * Use left shift then right shift
 9      * to sign extend from n bits to full int
10      */
11     int count = (sizeof(int)<<3)-n;
12     int leftright = (x << count) >> count;
13     /* See if still have same value */
14     return x == leftright;
15 }
```

code/data/bits.c

This code uses a common trick, demonstrated in Problem 2.23, of first shifting left by some amount k and then arithmetically shifting right by k. This has the effect of sign-extending from bit $w-k-1$ leftward.

Problem 2.71 Solution:

This problem highlights the difference between zero extension and sign extension.

A. The function does not perform any sign extension. For example, if we attempt to extract byte 0 from word `0xFF`, we will get 255, rather than -1.

B. The following code uses the trick shown in Problem 2.23 to isolate a particular range of bits and to perform sign extension at the same time. First, we perform a left shift so that the most significant bit of the desired byte is at bit position 31. Then we right shift by 24, moving the byte into the proper position and performing sign extension at the same time.

code/data/xbyte.c

```
1 int xbyte(packed_t word, int bytenum) {
2     int left = word << ((3-bytenum) << 3);
3     return left >> 24;
4 }
```

code/data/xbyte.c

Problem 2.72 Solution:

This code illustrates the hidden dangers of data type `size_t`, which is defined to be `unsigned` on most machines.

A. Since this one data value has type `unsigned`, the entire expression is evaluated according to the rules of unsigned arithmetic. As a result, the conditional expression will always succeed, since every value is greater or equal to 0.

B. The code can be corrected by rewriting the conditional test:

```
        if (maxbytes >= sizeof(val))
```

Problem 2.73 Solution:

Here is the solution.

code/data/bits.c

```
1 /* Addition that saturates to TMin or TMax */
2 int saturating_add(int x, int y) {
3     int sum = x + y;
4     int wm1 = (sizeof(int)<<3)-1;
5     /* In the following we create "masks" consisting of all 1's
6        when a condition is true, and all 0's when it is false */
7     int xneg_mask = (x >> wm1);
8     int yneg_mask = (y >> wm1);
9     int sneg_mask = (sum >> wm1);
10    int pos_over_mask = ~xneg_mask & ~yneg_mask & sneg_mask;
11    int neg_over_mask = xneg_mask & yneg_mask & ~sneg_mask;
12    int over_mask = pos_over_mask | neg_over_mask;
13    /* Choose between sum, INT_MAX, and INT_MIN */
14    int result =
15        (~over_mask & sum)|
16        (pos_over_mask & INT_MAX)|(neg_over_mask & INT_MIN);
17    return result;
18 }
```

code/data/bits.c

Logically, this code is a straightforward application of the overflow rules for two's complement addition. Avoiding conditionals, however, requires expressing the conditions in terms of masks consisting of all zeros or all ones.

Problem 2.74 Solution:

code/data/taddcheck.c

```
1 /* Determine whether arguments can be subtracted without overflow */
2 int tsub_ok(int x, int y) {
3     int diff = x-y;
4     int neg_over = x <  0 && y >= 0 && diff >= 0;
5     int pos_over = x >= 0 && y <  0 && diff <  0;
6     return !neg_over && !pos_over;
7 }
```

code/data/taddcheck.c

This is a straightforward application of the rules for addition, modified to change the conditions for argument `y`. This avoids the shortcoming of the proposed solution given in Problem 2.32.

Problem 2.75 Solution:

Problem 2.81 Solution:

These "C puzzle" problems are a great way to motivate students to think about the properties of computer arithmetic from a programmer's perspective. Our standard lecture on computer arithmetic starts by showing a set of C puzzles. We then go over the answers at the end.

A. `(x<y) == (-x>-y)`. No, Let `x` = $TMin_{32}$, `y` = 0.

B. `((x+y)<<4) + y-x == 17*y+15*x`. Yes, from the ring properties of two's complement arithmetic.

C. `~x+~y+1 == ~(x+y)`. Yes, $\tilde{}x+\tilde{}y+1 = (-x-1)+(-y-1)+1 = -(x+y)-1 = \tilde{}(x+y)$.

D. `(ux-uy) == -(unsigned)(y-x)`. Yes. Due to the isomorphism between two's complement and unsigned arithmetic.

E. `((x >> 2) << 2) <= x`. Yes. Right shift rounds toward minus infinity.

Problem 2.82 Solution:

This problem helps students think about fractional binary representations.

A. Letting V denote the value of the string, we can see that shifting the binary point k positions to the right gives a string $y.yyyyyy\cdots$, which has numeric value $Y+V$, and also value $V\times 2^k$. Equating these gives $V = \frac{Y}{2^k-1}$.

B. (a) For $y = 101$, we have $Y = 5$, $k = 3$, $V = \frac{5}{7}$.
   (b) For $y = 0110$, we have $Y = 6$, $k = 4$, $V = \frac{6}{15} = \frac{2}{5}$.
   (c) For $y = 010011$, we have $Y = 19$, $k = 6$, $V = \frac{19}{63}$.

Problem 2.83 Solution:

This problem helps students appreciate the property of IEEE floating point that the relative magnitude of two numbers can be determined by viewing the combination of exponent and fraction as an unsigned integer. Only the signs and the handling of ± 0 requires special consideration.

code/data/floatcomp-ans.c

```
1  int float_le(float x, float y) {
2      unsigned ux = f2u(x);
3      unsigned uy = f2u(y);
4      unsigned sx = ux >> 31;
5      unsigned sy = uy >> 31;
6
7      return
8          (ux<<1 == 0 && uy<<1 == 0) ||  /* Both are zero */
9          (sx && !sy) ||                 /* x < 0, y >=  0 */
10         (!sx && !sy && ux <= uy) ||    /* x >= 0, y >= 0 */
11         (sx && sy && ux >= uy);        /* x <  0, y <  0 */
12 }
```