Problem 2.64 Solution:

This problem is very simple, but it reinforces the idea of using different bit patterns as masks.

code/data/bits.c

```
/* Return 1 when any even bit of x equals 1; 0 otherwise.  Assume w=32 */
int any_even_one(unsigned x) {
    /* Use mask to select even bits */
    return (x&0x55555555) != 0;
}
```

code/data/bits.c

Problem 2.73 Solution:

Here is the solution.

code/data/bits.c

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

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 2.81 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)<<5) + x-y == 31*y+33*x`. Yes, from the ring properties of two's complement arithmetic.

C. `~x+~y == ~(x+y)`. No, let `x` = 0, `y` = 0.

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

E. `((x >> 1) << 1) <= 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 $y = 001$, we have $Y = 1$, $k = 3$, $V = \frac{1}{7}$.
   (b) For $y = 1001$, we have $Y = 9$, $k = 4$, $V = \frac{9}{15} = \frac{3}{5}$.
   (c) For $y = 000111$, we have $Y = 7$, $k = 6$, $V = \frac{7}{63} = \frac{1}{9}$.

Problem 2.85 Solution:

This exercise is of practical value, since Intel-compatible processors perform all of their arithmetic in extended precision. It is interesting to see how adding a few more bits to the exponent greatly increases the range of values that can be represented.

Description	Extended precision	
	Value	Decimal
Smallest pos. denorm.	$2^{-63} \times 2^{-16382}$	3.64×10^{-4951}
Smallest pos. norm.	2^{-16382}	3.36×10^{-4932}
Largest norm.	$(2-\epsilon) \times 2^{16383}$	1.19×10^{4932}

Problem 2.87 Solution:

This problem tests a lot of concepts about floating-point representations, including the encoding of normalized and denormalized values, as well as rounding.

Format A		Format B		Comments
Bits	Value	Bits	Value	
`1 01110 001`	$\frac{-9}{16}$	`1 0110 0010`	$\frac{-9}{16}$	
`0 10110 101`	208	`0 1110 1010`	208	
`1 00111 110`	$\frac{-7}{1024}$	`1 0000 0111`	$\frac{-7}{1024}$	Norm $\rightarrow$ denorm
`0 00000 101`	$\frac{5}{131072}$	`0 0000 0001`	$\frac{1}{1024}$	Smallest positive denorm
`1 11011 000`	-4096	`1 1110 1111`	-248	Smallest number $> -\infty$
`0 11000 100`	768	`0 1111 0000`	$+\infty$	Round to ∞.