Representing and Manipulating Information

Annalise Tarhan

August 13, 2020

2 Homework Problems

2.55 Compile and run show_bytes on different machines and determine the byte orderings used.

Two different MacBooks produced the same result: least significant bytes first, indicating little-endian machines.

2.56 Run show_bytes with different sample values.

Other values give predictable results.

2.57 Write procedures show_short, show_long, and show_double that print the byte representations of the corresponding C objects.

```
void show_short(short x) {
     show_bytes((byte_pointer) &x, sizeof(short));
}
void show_long(long x) {
     show_bytes((byte_pointer) &x, sizeof(long));
}
void show_double(double x) {
     show_bytes((byte_pointer) &x, sizeof(double));
}
```

2.58 Write a procedure is_little_endian that will return 1 when compiled and run on a little-endian machine and will return 0 otherwise. The program should run on any machine, regardless of its word size.

```
int is_little_endian() {
     int i = 1;
     if ((&i)[0]==1) {
          return 1;
     } else {
          return 0;
     }
}
```

2.59 Write a C expression that will yield a word consisting of the least significant byte of x and the remaining bytes of y.

```
int mash(int x, int y) {
     int maskedx=x&0xFF;
     y>>=8;
     y<<=8;
     return y|maskedx;
}
```

2.60 Complete the code for replace_byte.

```
int replace_byte(unsigned x, int y, unsigned char z) {
     int mask=0xFF << (i << 3);
     int newbyte=z << (i << 3);
     x=x|mask;
     x=x^mask;
     return x|newbyte;
}
```

2.61 Write C expressions that evaluate to 1 when the condition is true and 0 when they are false.

2.61.1 Any bit of x equals 1

```
int has_one(int x) {
     return x&&1;
}
```

2.61.2 Any bit of x equals 0.

```
int has_zero(int x) {
     return ~x&&1;
}
```

2.61.3 Any bit in the least significant byte of x equals 1.

```
int least_has_one(int x) {
     int least=x&0xff;
     return least&&1;
}
```

2.61.4 Any bit in the most significant byte of x equals 0.

```
int most_has_zero(int x) {
        /* Get most significant byte */
        int shiftby=(sizeof(int)-1)<<3;
        int shifted=x>>shiftby;
        /* If a byte has no zeros, its inverse equals 0 */
        int inverted=~shifted;
        int masked=inverted&0xff;
        /* !!masked is true if byte doesn't equal 0 */
        return !!masked;
}
```

2.62 Write a function int_shifts_are_arithmetic that yields 1 when run on a machine that uses arithmetic right shifts for data type int and yields 0 otherwise.

```
int int_shifts_are_arithmetic() {
        int i=INT_MIN;
        int shift_val=(sizeof(int)-1)<<3;
        int shifted=i>>shift_val;
        /* After shifting, only last byte will remain,
        rest will be all ones or all zeros */
        int one_masked_byte=shifted|0xff;
        int last_byte_zeroed_out=one_masked_byte^0xff;
        /* Returns 1 as long as it isn't all zeros */
        return last_byte_zeroed_out!=0;
}
```

2.63 Fill in the functions, which perform arithmetic right shifts on unsigned ints and logical shifts on signed ints.

```
unsigned srl(unsigned x, int k) {
        unsigned xsra = (int) x >> k;
        int intbits = 8*sizeof(int);
        int mask = 0;
        int i = 0;
        for (i; i<k; i++) {
                mask <<= 1;
                mask += 1;
        }
        for (i; i<intbits; i++) {
                mask <<= 1;
        }
        int xsra_masked = xsra|mask;
        int xsra_unmasked = xsra_masked^mask;
        return xsra_unmasked;
}

int sra(int x, int k) {
        int xsrl = (unsigned) x >> k;
        if (x < 0) {
                int i = 0;
                int mask = 0;
                int intbits = 8*sizeof(int);
                for (i; i<k; i++) {
                        mask <<= 1;
                        mask += 1;
                }
                for (i; i<intbits; i++) {
                        mask <<= 1;
                }
                xsrl = xsrl|mask;
        }
        return xsrl;
}
```

2.64 Write a function any_odd_one that returns 1 when any odd bit of x equals 1; 0 otherwise.

```
int any_odd_one(unsigned x) {
        unsigned all_even_ones = 0x55555555; //0b0101==0x5
        unsigned masked = x|all_even_ones;
        unsigned unmasked = masked^all_even_ones;
        return unmasked > 0;
}
```

2.65 Write a function odd_ones that returns 1 when x contains an odd number of 1s; 0 otherwise.

This code takes advantage of the fact that xor efficiently tracks even/odd. First, it compares the first sixteen bits of the given number with the second sixteen bits. (We are only interested in the second half of the result.) Each of those sixteen bits is zero if both compared bits are zero or if both bits are one. That effectively cancels out pairs of ones. Then, those sixteen bits are split and compared, ignoring the leading sixteen bits. Again, pairs of ones are cancelled out, and we are left with eight bits with the same parity as the original 32. Eventually, the least significant bit will have the same parity as x and can be returned without modification.

(Many thanks to TypeIA - https://stackoverflow.com/questions/21617970/how-to-check-if-value-has-even-parity-of-bits-or-odd/21618038#21618038)

```
int odd_ones(unsigned x) {
        unsigned shifted16 = x >> 16;
        unsigned xor16 = x ^ shifted16;

        unsigned shifted8 = xor16 >> 8;
        unsigned xor8 = xor16 ^ shifted8;

        unsigned shifted4 = xor8 >> 4;
        unsigned xor4 = xor8 ^ shifted4;

        unsigned shifted2 = xor4 >> 2;
        unsigned xor2 = xor4 ^ shifted2;

        unsigned shifted1 = xor2 >> 1;
        unsigned xor1 = xor2 ^ shifted1;

    return x == 0 || xor1 & 0x00000001;
}
```

2.66 Write a function that generates a mask indicating the leftmost 1 in x.

This function right shifts the number right and ORs itself with the result. It repeats the process with powers of two, changing all the bits to the right of the leading bit to ones. The final number is then shifted one to the right and XORed with itself, giving a mask.

(Thanks again to stackoverflow, this time erickson on
https://stackoverflow.com/questions/53161/find-the-highest-order-bit-in-c)

```
int leftmost_one(unsigned x) {
        unsigned shift_one = x >> 1;
        x = x|shift_one;
        unsigned shift_two = x >> 2;
        x = x|shift_two;
        unsigned shift_four = x >> 4;
        x = x|shift_four;
        unsigned shift_eight = x >> 8;
        x = x|shift_eight;
        unsigned shift_sixteen = x >> 16;
        x = x|shift_sixteen;
        unsigned shifted = x >> 1;
        return x^shifted;
}
```

2.67 Consider the following procedure, which should determine whether the machine's int size is 32 bits.

2.67.1 In what way does the code fail to comply with the C standard?

The C standard does not define behavior for shifting by large values, so beyond_msb would not necessarily be zero.

2.67.2 Modify the code to run properly on any machine for which data type int is at least 32 bits.

```
int int_size_is_32() {
        int set_msb = 1 << 31;
        int beyond_msb = set_msb << 1;
        return set_msb > beyond_msb;
}
```

2.67.3 Modify the code to run properly on any machine for which data type int is at least 16 bits.

```
int int_size_is_32() {
        int set_16_msb = 1 << 15;
        int beyond_16_msb = set_16_msb << 1;
        if (beyond_16_msb < set_16_msb) return 0;
        int set_32_msb = 1 << 31;
        int beyond_32_msb = set_32_msb << 1;
        return set_32_msb > beyond_32_msb;
}
```

2.68 Write a function, lower_one_mask(int n), which returns a mask with the least significant n bits set to 1.

```
        int one_one = 1 << (n-1);
        int mask = one_one - 1;
        return mask | one_one;
}
```

2.69 Write a function, rotate_left, which does a rotating left shift of n bits given an int x.

```
unsigned rotate_left(unsigned x, int n) {
        int word_size = sizeof(int) << 3;
        unsigned left_shifted = x << n;
        /* If the right shift was performed all at once,
        behavior would be unpredictable when n==0. */
        unsigned almost_right_shifted =
                x >> (word_size - n - 1);
        unsigned fully_right_shifted =
                almost_right_shifted >> 1;
        return left_shifted | fully_right_shifted;
}
```

2.70 Write a function, fits_bits, that returns 1 when an int x can be represented as an n-bit, 2's complement number, 0 otherwise.

```
int fits_bits(int x, int n) {
        int word_size = sizeof(int) << 3;
        int left_shift = x << (word_size - n);
        int right_shift = left_shift >> (word_size - n);
        return right_shift == x;
}
```

2.71 Consider the xbyte function, which is supposed to extract a bit from the given word indicated by bytenum.

2.71.1 What is wrong with the code?

The code returns the correct byte, but it doesn't take its sign into account. The int will always have three bytes worth of leading zeros, including the sign bit, so it will always be positive.

2.71.2 Give a correct implementation.

The problem indicates the answer should use subtraction, but this code seems to work without it. (Maybe the masking counts as subtraction?)

```
int xbyte(packed_t word, int bytenum) {
        int byte = (word >> (bytenum << 3)) & 0xFF;
        byte <<= 24;
        byte >>= 24;
        return byte;
}
```

2.72 Consider the copy_int function, which is supposed to check if enough space is available in the buffer before copying an int into it.

2.72.1 Explain why the conditional test always succeeds.

Because sizeof returns an unsigned value, maxbytes is cast to an unsigned before the subtraction. The result of any arithmetic function on two unsigned numbers will always be nonnegative, so the conditional is always true.

2.72.2 Show how you can rewrite the conditional test to make it work properly.

Instead of subtracting them, simply compare them: $if(maxbytes \geq sizeof(val))$

2.73 Write a function, saturating_add, that performs addition, but in the case of positive or negative overflow returns TMax or TMin respectively.

```
int saturating_add(int x, int y) {
        int sum = x+y;
        /* Shifting by 31 leaves only sign bit */
        int pos_x = (x >> 31 == 0);
        int pos_y = (y >> 31 == 0);
        int pos_sum = (sum >> 31 == 0);
        /* Overflow occurs when x and y have same sign bit
        but their sum has the opposite one */
        int pos_overflow = pos_x && pos_y && !pos_sum;
        int neg_overflow = !pos_x && !pos_y && pos_sum;
        /* Using && this way mimics an if statement */
        pos_overflow && (sum = INT_MAX);
        neg_overflow && (sum = INT_MIN);
        return sum;
}
```

2.74 Write a function tsub_ok that determines whether arguments can be subtracted without overflow.

```
int tsub_ok(int x, int y) {
        int diff = x-y;
        int pos_x = (x >> 31 == 0);
        int pos_y = (y >> 31 == 0);
        int pos_diff = (diff >> 31 == 0);
        int pos_overflow = pos_x && !pos_y && !pos_diff;
        int neg_overflow = !pos_x && pos_y && pos_diff;
        return !pos_overflow && !neg_overflow;
}
```

2.75 Write a function unsigned_high_prod that computes the high-order half of the product of two unsigned integers, using signed_high_prod, which computes the high-order bits of the product of two signed integers. Justify the correctness of the solution.

Equation 2.18 shows that for two signed integers x and y, where x' and y' are the unsigned integers with the same bit-level representations, $(x' * y')mod2^w = (x*y)mod2^w$. Since we are interested in the 2^w bit result, we double the modulo to 2^{2w} and calculate that $(x' * y')mod2^{2w} = (x * y + (x_{w-1} * y + y_{w-1} * x) *$

$2^w + (x_{w-1} * y_{w-1}) * 2^{2w}) mod 2^{2w}$. This time, only the final addend drops out and we are left with $(x' * y') mod 2^{2w} = x * y + (x_{w-1} * y + y_{w-1} * x) * 2^w$. This will equal the final 2^{2w} bit result, and our function needs to calculate the top w bits. To do that, we mod the result by 2^{w-1}. The final result is $(x * y) mod 2^{w-1} + (x_{w-1} * y + y_{w-1} * x)$, which is calculated below.

```
unsigned unsigned_high_prod(unsigned x, unsigned y) {
        /* Casting unsigned to int doesn't change bits */
        int int_x = (int) x;
        int int_y = (int) y;

        /* Compute high order bits of x and y */
        int sg_prod= signed_high_prod(int_x, int_y);

        /* Cast back to unsigned, bits still correct */
        unsigned unsg_prod = (unsigned) sg_prod;

        /* Determine sign bits of x and y */
        unsigned sgb_x = x >> 31;
        unsigned sgb_y = y >> 31;

        /* Calculate result based on equation 2.18 */
        unsigned result =
                unsg_prod + (x*sgb_y) + (y*sgb_x);
        return result;
}
```

2.76 Write an implementation of calloc that uses malloc and memset and has no vulnerabilities due to arithmetic overflow.

```
void *calloc(size_t nmemb, size_t size) {
        size_t required_size = nmemb * size;
        if (required_size < 1) {
                return NULL;
        }
        char *ptr = malloc(required_size);
        memset(ptr, 0, required_size);
        return 0;
}
```

2.77 For each of the following values of K, write a C expression to perform the multiplication of K and an integer variable x using at most three operations, using only +, -, and ¡¡.

2.77.1 K=17

```
(x << 4) + x;
```

2.77.2 K=-7

```
x - (x << 3)
```

2.77.3 K=60

```
(x << 6) - (x << 2);
```

2.77.4 K=-112

```
(x << 4) - (x << 7);
```

2.78 Write a function divide_power2 that computes x divided by two to the kth power with correct rounding.

```
int divide_power2(int x, int k) {
        int bias = 0;
        int bias_for_negatives = (1 << k) - 1;
        /* If x's sign bit isn't 0, calculate with bias */
        (x >> 31) && (bias = bias_for_negatives);
        return (x + bias) >> k;
}
```

2.79 Write a function mul3div4 that computes 3 * x/4 for any integer x.

```
int mul3div4(int x) {
        int bias = 0;
        int bias_for_negatives = 3;
        (x >> 31) && (bias = bias_for_negatives);
        int multiplied = (x << 1) + x;
        int divided = (multiplied + bias) >> 2;
        return divided;
}
```

2.80 Write a function threefourths that computes the value of 3/4*x, rounded toward zero, that doesn't overflow for any integer x.

```
int threefourths(int x) {
        int remainder = 0;
        int low_bits = x & 0x3;
        int is_neg = (x >> 31);
        !is_neg && (low_bits == 3) && (remainder = 1);
        is_neg && (low_bits == 1) && (remainder = 1);
        is_neg && (low_bits == 2) && (remainder = 1);
        is_neg && (low_bits == 3) && (remainder = 2);
        return (x >> 1) + (x >> 2) + remainder;
}
```

2.81 Write C expressions to generate the following bit patterns.

2.81.1 $1^{w-k}0^k$

```
-1 << k
```

2.81.2 $0^{w-k-j}1^k0^j$

```
((1 << k) - 1) << j
```

2.82 For each C expression, indicate whether or not the expression always yields 1. Explain why or give an argument that makes it yield 0.

2.82.1 $(x < y) == (-x > -y)$

False. When x equals TMin, any $y \neq x$ will be greater than x, but $-x = x$, so any $-y > -x$.

2.82.2 $((x + y) << 4) + y - x == 17 * y + 15 * x$

True. Left shifting is equivalent to multiplying by powers of two and shifting distributes over addition. If there is overflow, it occurs in the same way on both sides.

2.82.3 $\sim x + \sim y + 1 == \sim (x + y)$

True. This reflects the fact that negating a bit vector representing a number n results in a bit vector representing -(n+1).

2.82.4 $(ux - uy) == -(unsigned)(y - x)$

True. Casting between signed and unsigned ints does not change their bit representation, only how they are interpreted. Subtraction behaves the same way for signed and unsigned numbers, so the casts don't change the math either.

2.82.5 $((x >> 2) << 2) <= x$

True. When there is no overflow, the number doesn't change. When it does, the effect is to set the two least significant bits to zero. For both positive and negative numbers, changing those bits to zero decreases the value of the number.

2.83 Consider numbers having a binary representation consisting of an infinite string of the form 0.yyyyy... where y is a k-bit sequence.

2.83.1 Let $Y = B2U_k(y)$. Give a formula in terms of Y and k for the value represented by the infinite string.

$Value = \frac{Y}{2^k - 1}$

2.83.2 What is the numeric value of the following values of y?

101 $\frac{5}{7}$

0110 $\frac{2}{5}$

010011 $\frac{19}{31}$

2.84 Fill in the return value for the following procedure, which tests whether its first argument is less than or equal to its second.

```
(sx > sy) || (ux <= uy)
```

2.85 Write formulas for the exponent E, the significand M, the fraction f, and the value V for the given quantities, as well as a description of the bit representation.

2.85.1 The number 7.0

$E = 2 \quad M = \frac{7}{4} \quad f = \frac{3}{4} \quad V = 7.0$
Bit representation: sign bit is 0, exponent is bit representation of 2 plus the bias, or $1 + 2^{k-1}$, fraction is two ones at the beginning (for $\frac{1}{2}$ and $\frac{1}{4}$) and zeros to fill.

2.85.2 The largest odd integer that can be represented exactly.

$E = 0 \quad M = 2^{n+1} - 1 \quad V = 2^{n+1} - 1$
Bit representation: sign bit is 0, exponent is bit representation of the bias, $2^{k-1} - 1$, or one zero followed by ones to fill, fraction is all ones.

2.85.3 The reciprocal of the smallest positive normalized value.

$E = 2^{k-1} - 2 \quad M = 1 \quad V = 2^{2^{k-1}-2}$
Bit representation: sign bit is 0, exponent is all ones, fraction is all zeros. Hang on, that's infinity.

2.86 Fill in the values and decimals for each number in the extended-precision floating-point format.

2.86.1 Smallest positive denormalized

Value: 2^{-16382}
Decimal: Zero-ish

2.86.2 Smallest positive normalized

Value: 2^0
Decimal: 1.0

2.86.3 Largest normalized

Value: $(2^{63} - 1) * (2^{16} - 2)$
Decimal: Infinity-ish

2.87 Fill in the values for each number in the half-precision floating-point format.

2.87.1 -0

Hex: 0x8000 M: 0 E: -14 $V : -0$ D: -0.0

2.87.2 Smallest value ¿ 2

Hex: 0x4001 M: $\frac{1025}{1024}$ E: 1 $V : 1025 * 2^{-9}$ D: 2.00195312

2.87.3 512

Hex: 0x6000 M: 1 E: 9 V: 512 D: 512.0

2.87.4 Largest denormalized

Hex: 0x03FF M: $\frac{1023}{1024}$ E: -14 V: $1023 * 2^{-24}$ D: 0.00006098

2.87.5 $-\infty$

Hex: 0xFC00 – – V: $-2047 * 2^6$ D: -131008.0

2.87.6 Number with hex representation 3BB0

Hex: 3BB0 M: $\frac{123}{64}$ E: -1 V: $123 * 2^{-7}$ D: 0.9609375

2.88 Fill in the values and floating point representations for formats A and B.

2.88.1 0 10110 011

Value: 176 Bits: 0 1110 0110 Value: 176

2.88.2 1 00111 010

Value: $-\frac{5}{2^{10}}$ Bits: 1 0000 0101 Value: $-\frac{5}{2^{10}}$

2.88.3 0 00000 111

Value: $\frac{7}{2^{17}}$ Bits: 0 0000 0001 Value: $\frac{1}{2^{10}}$

2.88.4 1 11100 000

Value: -8192 Bits: 1 1110 1111 Value: -248

2.88.5 0 10111 100

Value: 384 Bits: 0 1111 0000 Value: ∞

2.89 For each of the C expressions, indicate whether or not the expression always yields 1. If so, explain why. Otherwise give a counterexample.

2.89.1 (float) x == (float) dx

True. Casting ints or doubles to floats can result in a rounded number, but the int and double representations of the same int will be rounded the same way.

2.89.2 dx-dy == (double) (x-y)

True. No loss of precision casting ints to doubles.

2.89.3 (dx+dy) + dz == dx + (dy + dz)

True. Two's complement addition is associative.

2.89.4 (dx*dy) * dz == dx * (dy * dz)

True. Two's complement multiplication is associative.

2.89.5 dx/dx == dz/dz

False. Division by zero is undefined.

2.90 Fill in the blank portions of the code to compute a floating point representation of 2^x.

```
float fpwr2(int x) {
        unsigned exp, frac;
        unsigned u;
        if (x < -149) {
                exp = 0;
                frac = 0;
        } else if (x < -126) {
                exp = 0;
                frac = 126 - x;
        } else if (x < 128) {
                exp = x + 127;
                frac = 0;
        } else {
                exp = UINT_MAX;
                frac = 0;
        }
        u = exp << 23 | frac;
        return u2f(u);
}
```

2.91 Consider the following approximations of π: $\frac{223}{71} < \pi < \frac{22}{7}$ and 0x40490FDB

2.91.1 What is the fractional binary number denoted by this floating-point value?

$2 + 1 + \frac{1}{8} + \frac{1}{64} + \frac{1}{2048} + \frac{1}{4096} + \frac{1}{8192} + \frac{1}{16348} + \frac{1}{32696} + \frac{1}{65392} + \frac{1}{261568} + \frac{1}{523136} + \frac{1}{2092544} + \frac{1}{4185088}$

2.91.2 What is the fractional binary representation of $\frac{22}{7}$?

$2 + 1 + \frac{1}{8} + \frac{1}{64} + \frac{1}{512} + \frac{1}{4096} + \frac{1}{32768} + \frac{1}{262144} + \frac{1}{2097152} + \frac{1}{16777216} + ...$

2.91.3 At what bit position do these two approximations to π diverge?

They diverge at the 9th digit to the right of the binary point, or at 2^{-9}.

2.92 Write a function to compute -f for any given float_bits f.

```
float_bits float_negate(float_bits f) {
        unsigned sign = f>>31;
        unsigned exp = f>>23 & 0xFF;
        unsigned frac = f & 0x7FFFFF;
        if (exp != 0xFF || frac == 0) {
                sign = ~sign;
        }
        return (sign << 31) | (exp << 23) | frac;
}
```

2.93 Write a function to compute the absolute value of any given float_bits f.

```
float_bits float_absval(float_bits f) {
        unsigned sign = f>>31;
        unsigned exp = f>>23 & 0xFF;
        unsigned frac = f & 0x7FFFFF;
        if (exp != 0xFF || frac == 0) {
                sign = 0;
        }
        return (sign << 31) | (exp << 23) | frac;
}
```

2.94 Write a function to compute 2.0 * f for any given float_bits f.

```
float_bits float_twice(float_bits f) {
        unsigned sign = f>>31;
        unsigned exp = f>>23 & 0xFF;
        unsigned frac = f & 0x7FFFFF;
        if (exp == 0) {
                if (frac >> 22 == 1) {
                        exp = 1;
                }
                frac = ((frac << 1) & 0x7FFFFF);
        } else if (exp < 0xFE) {
                exp += 1;
        } else if (exp >= 0xFE) {
                exp = 0xFF;
                frac = 0;
        }
        return (sign << 31) | (exp << 23) | frac;
}
```

2.95 Write a function to compute 0.5 * f for any given float_bits f.

```
float_bits float_half(float_bits f) {
        unsigned sign = f>>31;
        unsigned exp = f>>23 & 0xFF;
        unsigned frac = f & 0x7FFFFF;
        if (exp == 0) {
                if ((frac & 3) == 3) {frac += 1;}
                frac >>= 1;
        } else if (exp == 1) {
                exp = 0;
                unsigned last_bits = frac & 3;
                frac = (frac >> 1) | (1 << 22);
                if (last_bits == 3) { frac += 1; }
        } else if (exp > 1 && exp != 0xFF) {
                exp -= 1;
        }
        return (sign << 31) | (exp << 23) | frac;
}
```

2.96 Write a function to compute (int) f for any given float_bits f. If the result is out of range or NaN, the function should return 0x80000000.

```
int float_f2i(float_bits f) {
        unsigned sign = f>>31;
        unsigned exp = f>>23 & 0xFF;
        unsigned frac = f & 0x7FFFFF;
        int result = -1;
        int invalid = 0;
        int biased_exp = exp - 127;
        /* Floats with negative exponents round to zero */
        if (biased_exp < 0) {
                result = 0;
        } else if (exp < 0xFF){
                if (biased_exp < 23) {
                        result = frac >> (23-biased_exp);
                } else {
                        result = frac << (biased_exp-23);
                }
                /* Normalized floats add one to fraction,
                then multiply by exponent */
                result += (1 << biased_exp);
        }
        /* At this point, a negative result
           indicates overflow */
        if (result < 0) {
                invalid = 1;
        }
        if (sign == 1) {
                result *= -1;
        }
        if (invalid) {
                result = 0x80000000;
        }
        return result;
}
```

2.97 Write a function to compute the bit level representation of (float) i for any given int.

```
float_bits float_i2f(int i) {
        unsigned sign = (i < 0);
        unsigned exp = 0;
        unsigned frac;

        /* For negative ints, flip bits and add one */
        unsigned coerced_positive = i;
        if (sign) {
                coerced_positive = ~coerced_positive+1;
        }

        /* temp holds same value as coerced_positive */
        unsigned temp = coerced_positive;

        /* Find exp value by repeatedly right shifting */
        while (temp > 0) {
                temp >>= 1;
                exp++;
        }

        /* Reset temp to original value */
        temp = coerced_positive;

        /* Find frac value. First, drop leading one */
        frac = temp << (33-exp);

        /* Shift frac left, making room for sign and exp.
        Round carefully. */
        unsigned last_nine = frac & 0x1ff;
        unsigned last_ten = frac & 0x3ff;
        frac >>= 9;
        if (last_nine >= 260 || last_ten >= 768) {
                frac += 1;
                /* In case rounding added frac digit */
                if ((frac ^ 0xffffff) == 0x7fffff) {
                        exp += 1;
                        frac = 0;
                }
        }

        /* Construct float by shifting bits */
        unsigned sign_bits = sign << 31;
```

```
        unsigned exp_bits = (exp+126) << 23;

        /* 0 is a special case, denormalized */
        if (i == 0) {
                exp_bits = 0;
        }

        return sign_bits | exp_bits | frac;
}
```

Testing Functions

```
float two_to_the[256];

void setUpTwoPowers() {
        for (int i=0; i<256; i++) {
                two_to_the[i] = pow(2.0, i-128);
        }
}

float getPowerOfTwo(int power) {
        return two_to_the[power+128];
}

double getMantissaValue(unsigned frac) {
        double result = 0.0;
        unsigned bit;
        for (int i=23; i>0; i--) {
                bit = frac & 0x1;
                if (bit) {
                        result += getPowerOfTwo(-i);
                }
                frac >>= 1;
        }
        return result;
}

int getMultiplier(unsigned sign) {
        if (sign == 0) {
                return 1;
        } else {
                return -1;
        }
}

float bits_to_float(float_bits f) {
        unsigned sign = f>>31;
        unsigned exp = f>>23 & 0xFF;
        unsigned frac = f & 0x7FFFFF;
        double actual_float;

        /* When exponent is all ones, special value */
        if (exp == 0xFF) {
                if (frac == 0) {
```

```
                        /* When exp is all ones and frac
                        is all zeros, positive or negative
                        infinity */
                        if (sign == 0) {
                                actual_float = 1.0/0.0;
                        } else {
                                actual_float = -1.0/0.0;
                        }

                        /* Otherwise, Nan, so return
                        original */
                } else {
                        actual_float =
                        getMultiplier(sign) *
                        getMantissaValue(frac) *
                        getPowerOfTwo(-127);
                }

                /* If exponent is all zeros,
                denormalized */
        } else if (exp == 0) {
                actual_float = getPowerOfTwo(-126) *
                getMultiplier(sign) *
                getMantissaValue(frac);

                /* Else, normalized */
        } else {
                actual_float = getPowerOfTwo(exp-127) *
                 getMultiplier(sign) *
                 (1+getMantissaValue(frac));
        }
        return actual_float;
}
```