CSCI-UA.0201 Fall 2019
Assignment 2 Hints

1. Defining function-like macros using the preprocessor

Use #define to define a macro that results in a textual substitution before the code is compiled. For example,

```
#define TIMES24(x) ((x) * 24)
```

will cause the statements

```
y = TIMES24(z);
a = TIMES24(b+c);
```

to be transformed, before compilation, into

```
y = (z) * 24;
a = (b+c) * 24;
```

The second example shows why it is always a good idea, when defining a macro, to put parentheses around the macro parameter, in this case x.

2. Extracting the value of a multiple-bit field from a variable

Note: Remember that bit 0 is the rightmost (least significant) bit and bit 31 is the leftmost (most significant) bit of a 32-bit variable.

If you want to extract *m* bits from a variable v, starting at bit position *n* (i.e. the value of bits *n* through *n*+*m*-1 of v), then:

(1) shift v to the right by *n*, and

(2) bitwise-and (&) the result of the shift with a mask whose rightmost *m* bits are 1.

For example,

```
#define MASK5 0x1F   // a mask whose rightmost 5 bits are 1.
y = ((x >> 12) & MASK5);   // sets y to the value of bits 12 through 16 of x
```

3. Setting the value of a multiple-bit field in a variable

If you want to write an *m* bit value *k* into a variable v, starting at bit position *n* (i.e. setting the value of bits *n* through *n*+*m*-1 of v to *k*), then:

(1) bitwise-and (&) *k* with a mask whose rightmost *m* bits are 1

(2) bitwise-or (|) x with the result of step (1) shifted left by *n*.

For example,

```
#define MASK12 0xFFF   // a mask whose rightmost 12 bits are 1
x = x | ((k & MASK12) << 6);  // sets bits 6 through 17 of x to k.
```

(continued on next page)

4. Treating a floating point variable as an unsigned 32-bit value (without conversion)

If you want to operate on the individual bits of a floating point variable f (as in the assignment), you cannot perform & or | on f directly, nor can you simply cast f to unsigned int. For example, if you were trying to extract the rightmost 16 bits of f,

```
unsigned int x = f & 0xFFFF;   // This will be rejected by the compiler, & cannot be used on floats

unsigned int x = ((unsigned int) f) & 0xFFFF;   // This will convert f to an unsigned integer,
                                                //  changing the bits. No good.
```

To get around this issue, first convert the address of f to an unsigned int * (i.e. an unsigned int pointer) and then dereference the pointer to get the value. For example,

```
// In two steps
unsigned int *pf = (unsigned int *) &f;     //  pf points to f.
unsigned int x = (*pf) & 0xFFFF;            // x gets the value of the rightmost 16 bits of f

// In one step
unsigned int x = (*((unsigned int *) &f)) & 0xFFFF; // x gets the value of the rightmost 16 bits of f
```

5. Treating an unsigned int as a float (without conversion)

Similar to above, if you've assembled the bits of an unsigned int variable x and want to treat it as a float (as in the assignment), you convert the address of x to a float * (i.e. a float pointer) and then dereference the pointer. For example,

```
// In two steps
float *px = (float *) &x;
float f = *px;

// In one step
float f = *((float *) &x);
```

6. Sign-extending 32-bit signed integer to a 64-bit signed integer

Converting a 32-bit signed integer to a 64-bit signed integer is done automatically by the assignment statement. For example,

```
int x = 54;
long int y = x;   // y gets the 64-bit representation of x's value.
```

Note that sign extension is automatically performed. For example, in the following code,

```
int x = -25;
long int y = x;
```

the leftmost bit of x is 1 (since it is a two's complement negative number), so each of the leftmost 32 bits of y have been filled in with a 1. That is, sign extension when going from a 32-bit number to a 64-bit number writes the value of the leftmost bit of the 32-bit number into the each of the leftmost 32 bits of the 64-bit number.