f) In our 32-bit single-precision floating point representation, we decide to convert one significand bit to an exponent bit. How many **denormalized numbers** do we have relative to before? (Circle one)

More [✓ Fewer] Half as many because lost a significand bit **(1 pt)**

Rounded to the nearest power of 2, how many denorm numbers are there in our new format?
(Answer in IEC format) **(1 pt)**

22 significand bits + sign bit but not counting ±0, so exactly 2^{23}-2 denorms __**8 Mebi #s**___

e) 0xc14c0000 interpreted as a float is 0b1100 0001 0100 1100 0000 0000 0000 0000, or separated by the IEEE 754 fields: |1|100 0001 0|100 1100 0000 0000 0000 0000|. The first 1 tells us it's negative. The second field is 130. 130 minus our bias of 127 is an exponent of 3. So now we can write this as we normally do: -1 x 1.10011 x 2^3, (not forgetting the implicit leading 1) and the 2^3 means we shift the binary point three spaces to the right, yielding the number -1100.11_2, which is -12.75. (3 pts)

f) The smallest positive normalized number has a sign bit of 0 and an exponent field of E=1 (remember that E=0 is reserved for denorms and ±0). The smallest number in magnitude will have a mantissa field of all zeros, yielding |0|0000 0001|0000 0000 0000 0000 0000 000| = 0x00800000, which we interpret as $(-1)^0$ x (1.0…0) x 2^{1-127} = 2^{-126}. (3 pts)

g) This was a hard question. We recall there were two infinities, -∞ and +∞ and that their formats were special; we'd reserved all ones in the exponent and zeros in the mantissa especially for it. So that means they look like 0bX111 1111 1000 0000 0000 0000 0000 0000 (= 0x[F7]F800000), where X is 0 for +∞ and 1 for -∞. Well the comment says to make them the same. What instruction (with an argument of simply "1") can do that? Why *shift left logical*, which would push the leftmost bit off the edge yielding 0xFF000000. Now, the second blank needs to look at \$a0 and if it's 0xFF000000 (either infinity) then \$v0 should be set to 0, otherwise set \$v0 to any non-zero value. We need something like "not-equal-to", or (in C): \$v0 = (\$a0 != 0xFF000000). The logical operation *xor* fits the bill, because xor is a "balancing" operation … when the arguments are perfectly "balanced" (i.e. equal), it is a zero. Otherwise it's not. So xor is like "not equal to", and xnor (not xor) is "equal to"). Thus the answer is: (4 pts)

```
        sll $a0 $a0 1
        xor $v0 $a0 0xFF000000
        jr $ra
```

```
IsNotInfinity:  movl   %edi, %eax
                shll  $1, %eax      # make +/- Inf look the same
                xorl  $0xFF000000, %eax
                ret
```

M3) What is that Funky Smell? Oh, it’s just Potpourri (10 pts)

a) This question asked for *non-negative* floating point numbers < 2. This did NOT include -0. Some important things to remember are that all positive denorm numbers count and the floating point representation of +2 is `0x40000000` (exponent of `0x80`). So non-negative floating point numbers less than 2 are any combination where the 2 most significant bits are 0’s. This leaves any combination of the lower 30 bits, so there are 2^{30} such numbers. (1 pt)

+0.5 pt for value, +0.5 pt for work WITH correct value.

Question 4: *Let Me Float This Idea By You* (9 Points, 16 Minutes)

(–1pt if 32 bits used)

For a very simple household appliance like a thermostat, a more minimalistic microprocessor is desired to reduce power consumption and hardware costs. We have selected a **16-bit** microprocessor that does not have a floating-point unit, so there is no native support for floating point operations (no `float/double`). However, we'd still like to represent decimals for our temperature reading so we're going to implement floating point operations in software (in C).

(also accepted: **unsigned int**)

a) Define a new variable type called `fp`: **(1 pt)**

_**typedef int fp;**_______________________

Many people were not sure what to do here. 1 pt was given mainly to those who wrote a valid statement using typedef or the #define directive, or were close. Struct definitions were also accepted.

We have decided to use a representation with a **5-bit exponent field** while following all of the representation conventions from the MIPS 32-bit floating point numbers **except denorms**.

Fill in the following functions. Not all blanks need to be used. You can call these functions and assume proper behavior regardless of your implementation. Assume our hardware implements the C operator ">>" as *shift right arithmetic*.

b) **(1 pt)**

```
/* returns -num */
fp negateFP(fp num) {
      return _num ^ 0x8000___________________________________________;
}
```

If you assumed 32-bit type, then using 0x80000000 was okay.

c) **(1 pt mask/shift, 1 pt bias)**

```
/* returns the signed value of the exponent */
int getExp(fp num) {
      ______________________________________________________________

      return _((num & 0x7C00) >> 10) - 15_________________________;
}
```

0x7c00 to zero out everything but the exponent field, shift right by 10 to get the unsigned value, then subtract bias of $2^4 – 1 = 15$ to get the actual signed value.

d) **(1 pt per line)**

```
/* multiplies floating point num by 2^n, while detecting over/underflow */
/* remember, there are no denorms */
fp multPow2(fp num,int n) {
      _int exp = getExp(num) + n; /* get exponent or exponent + n */_______
      if(_exp > 15___________________________________) exit(1);  #overflow
      if(_exp < -15__________________________________) exit(-1); #underflow
      _num &= 0x83FF; /* zero old exponent */______________________________
      return _num | ((exp + 15) << 10); /* set new exponent */_____________
}
```

5 pts total:

First line: 1pt for trying to get the exponent by means of getExp(num) or manually retrieving it.

Second and third line: **–0.5 pt each line** if the numeric value on the right was close, but not correct.

Fourth and fifth: needed to correctly zero out the exponent field of num, and OR or add the modified exponent back into that field. 1pt for not forgetting to re-add the bias, and 1pt for getting the masking/shifting right.

Other:

–1 pt for left shifting the exponent by n instead of adding.

If you didn't add the 15 bias because in getExp() you didn't subtract the 15 bias, then I didn't mark you off for that.

Question 5: Floating Point (10 pts)

Assume integers and IEEE 754 single precision floating point are **32 bits wide**.

a) Convert from IEEE 754 to decimal: **0xC0900000** [3 pts]

S = 1, E = 0b1000 0001, M = 0010…0; $-1.001_2 \times 2^2 = -100.1_2$

$\boxed{-4.5}$

b) What is the smallest positive integer that is a power of 2 that can be represented in IEEE 754 but not as a signed int? You may leave your answer as a power of 2. [2 pts]

Largest 32-bit signed int is $2^{31} - 1$.

$\boxed{2^{31}}$

c) What is the *smallest positive* integer x such that `x + 0.25` can't be represented? You may leave your answer as a power of 2. [3 pts]

Need 2^{-2} digit to run off end of mantissa, so

$10000000000000000000000.01_2 = 1.00000000000000000000000001 \times 2^{22}$

$\boxed{2^{22}}$

d) We have the following word of data: **0xFFC00000**. Circle the number representation below that results in the *most negative number*. [1 pt]

Unsigned Integer (positive number)	[✓ Two's Complement] (negative number)	Floating Point (NaN)

e) If we decide to stray away from IEEE 754 format by making our Exponent field 10 bits wide and our Mantissa field 21 bits wide. This gives us (circle one): [1 pt]

MORE PRECISION // [✓ LESS PRECISION]

Fewer mantissa bits means less precision.

Question 1: Number Representation (8 pts)

a) Convert `0x1A` into base 6. Don't forget to indicate what base your answer is in! [1 pt]

0x1A = 0b1 1010 = 16 + 8 + 2 = 26 = $4 \times 6^1 + 2 \times 6^0$

$\boxed{42_6}$

b) In IEEE 754 floating point, how many numbers can we represent in the interval [10,16)? You may leave your answer in powers of 2. [3 pts]

$\boxed{2^{22} + 2^{21} = 3 \times 2^{21}}$

10 = $0\text{b}1010 = 1.01 \times 2^3$ and 16 = $0\text{b}10000 = 1.0 \times 2^4$

Count all numbers with Exponent of 2^3 and Mantissa bits of the form { 1b'0, 1b'1, 21{1b'X} } and { 1b'1, 22{1b'X} }, for a total of $2^{21} + 2^{22}$ numbers.

c) If we use 7 Exponent bits, a denorm exponent of -62, and 24 Mantissa bits in floating point, what is the largest positive power of 2 that we can multiply with 1 to get *underflow*? [2 pts]

$\boxed{2^{-87}}$

Smallest denorm is $2^{-62} \times 0.0000\ 0000\ 0000\ 0000\ 0000\ 0001 = 2^{-86}$, which is representable. So the next smaller power of 2 is unrepresentable and causes underflow.