Part I -  Number Conversions

**1169.00**  
Convert **1169.00** into a binary number with three places to the right of the binary point:

*(0b will denote binary number - I don't know how to access subscript on here)*

1024  512  256  128  64  32  16  8  4  2  1

**1         0       0       1      0     0     1    0  0  0  1**

**1169.00 = 0b10010010001.000**

2) Convert the binary number into a hexadecimal number. *(0x will denote hexadecimal number)*

0b10010010001.000 to hexadecimal:

* Break into fours starting at binary point: (pad 0s for leftmost and rightmost relative to binary point ->) **0100 1001 0001 . 0000**
* 0100 = **0x4**
* 1001 = **0x9**
* 0001 = **0x1**
* .
* 0000 = **0x0**

0b10010010001.000 = **0x4910.0**

3) Convert the binary number to a floating-point number using the IEEE -754 Floating-Point Standard

0b10010010001.000:

* 32-bit number
* Sign-bit, 8 exponent bits, 23 significand bits
  + The exponent bits use excess-127 bias
* The sign-bit will be 0 as this is a positive number: **0**
* We will have to move binary point 10 spaces - therefore, exponent will be 10
  + Exponent will have to be 127 + 10 = 137 = 0b10001001
  + **0 1000 1001**
* We moved the binary point 10 spots to the right of the most significant bit since it is a 1:  *1.0010010001*
  + We drop the '1.' and append the '0010010001' after the exponent bits
  + Fill the remaining bits to the right with 0s so that the total is 32 bits
    - 1 - sign
    - 8 - exponent
    - 23 - significand/mantissa
* **0  10001001  00100100010000000000000**

Part II - Adding Binary Numbers

Select two decimal numbers between -60 and +60. One should be positive and the other should be negative.

Convert the numbers into 8-bit unsigned numbers with negative numbers in the 2’s complement form. (Remember that positive numbers have the leftmost bit =0 and negative numbers have the leftmost bit =1).  
Add the two numbers together to generate an eight-bit binary result. Show your work.

+37 = **0010 0101**

-23 = -(0001 0111) = 1110 1000 + 1 = **1110 1001**

1 1 1              1           <--- Carries (the last carry to the left is left off)

   0 0 1 0 0 1 0 1        <--- +37

   1 1 1 0 1 0 0 1        <--- -23

**0 0 0 0 1 1 1 0         <--- +14**= 37 - 23

Part III

If a computer is capable only of manipulating and storing integers, what difficulties present themselves? How are these difficulties overcome:

If a computer can only store and manipulate integers, then how do we store/manipulate fractions? What integers can the computer store? Is it a binary system? Is it a decimal system? How can we keep track of which integers belong together and in what order? The binary system overcomes a few of these obstacles as it is simply 1/0 which translates nicely to on/off or high/low voltage. Memory addresses/registers, the ability to keep track of where things are stored allows for the 0s and 1s to translate into encodings so that we, the users, do not have to look at a wall of 1s and 0s which would be meaningless and impractical. So, taking for granted that a computer could only possibly store something like an on/off the real solution that helped overcome these obstacles and bridge the gap between person and machine were encodings like ASCII, UNICODE, and BCD/EBCDIC before them.