Put the decimal number 64.2 into the IEEE standard single precision floating point representation.

**Step 1:**
Get a binary representation for 64.2. To do this, get unsigned binary representations for the stuff to the left and right of the decimal point separately.

64 is 1000000

.2 can be gotten using the algorithm:

\[
\begin{align*}
.2 \times 2 &= 0.4 & 0 \\
.4 \times 2 &= 0.8 & 0 \\
.8 \times 2 &= 1.6 & 1 \\
.6 \times 2 &= 1.2 & 1
\end{align*}
\]

.2 \times 2 = 0.4 0 now this whole pattern (0011) repeats.

\[
\begin{align*}
.4 \times 2 &= 0.8 & 0 \\
.8 \times 2 &= 1.6 & 1 \\
.6 \times 2 &= 1.2 & 1
\end{align*}
\]

so a binary representation for .2 is .001100110011... .

Putting the halves back together again:

64.2 is 1000000.0011001100110011...

**Step 2:**
Normalize the binary representation. (make it look like scientific notation)

1.000000 00110011... x 2^6

**Step 3:**
6 is the true exponent. For the standard form, it needs to be in 8-bit, biased-127 representation.

\[
\begin{align*}
6 \\
+ 127 \\
----- \\
133
\end{align*}
\]

133 in 8-bit, unsigned representation is 1000 0101

This is the bit pattern used for E in the standard form.

**Step 4:**
The mantissa/significand stored (F) is the stuff to the right of the radix point in the normalized form. We need 23 bits of it.

000000 00110011001100110

**Put it all together** (and include the correct sign bit):

\[
\begin{array}{ccc}
S & E & F \\
0 & 1000101 & 0000000011001100110110
\end{array}
\]

0b 0100 0010 1000 0000 0110 0110 0110 0110
0x 4 2 8 0 6 6 6 6
Computers represent real values in a form similar to that of scientific notation. Consider the value 1.23 x 10^4

The number has a sign (+ in this case)
The significand (1.23) is written with one non-zero digit to the left of the decimal point.
The base (radix) is 10.
The exponent (an integer value) is 4. It too must have a sign.

There are standards which define what the representation means, so that across computers there will be consistancy.

Note that this is not the only way to represent floating point numbers, it is just the IEEE standard way of doing it. Here is what we do… The representation has three fields:

----------------------------
| S |   E     |     F      |
----------------------------

S is one bit representing the sign of the number  
E is an 8-bit biased integer representing the exponent  
F is an unsigned integer

The decimal value represented is:

\[ S \times (-1)^{e} \times f \times 2^{E - bias} \]

where

\[ e = E - bias \rightarrow E = e + bias \]

\[ f = \frac{F}{2^{n}} + 1 \rightarrow F = (f-1)\times2^{n} \]

For single precision representation, \( n = 23 \) and \( bias = 127 \)

For double precision representation (a 64-bit representation), \( n = 52 \) (there are 52 bits for the mantissa field) \( bias = 1023 \) (there are 11 bits for the exponent field)

Now, what does all this mean?

- S, E, F all represent fields within a representation. Each is just a bunch of bits.
- S is just a sign bit. 0 for positive, 1 for negative. This is the sign of the number.
- E is an exponent field. The E field is a biased-127 integer representation. So, the true exponent represented is (E - bias).
- The radix for the number is ALWAYS 2.  
  Note: Computers that did not use this representation, like those built before the standard, did not always use a radix of 2. For example, some IBM machines had radix of 16.
- F is the mantissa (significand). It is in a somewhat modified form. There are 23 bits available for the mantissa. It turns out that if floating point numbers are always stored in their normalized form, then the leading bit (the one on the left, or MSB) is ALWAYS a 1. So, why store it at all? It gets put back into the number (giving 24 bits of precision for the mantissa) for any calculation, but we only have to store 23 bits. This MSB is called the hidden bit.