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:

\[
\begin{align*}
.2 \times 2 &= 0.4 \quad 0 \rightarrow \text{keep the whole portion here and carry the decimal down} \\
.4 \times 2 &= 0.8 \quad 0 \\
.8 \times 2 &= 1.6 \quad 1 \\
.6 \times 2 &= 1.2 \quad 1 \\
.2 \times 2 &= 0.4 \quad 0 \rightarrow \text{now this whole pattern (0011) repeats.} \\
.4 \times 2 &= 0.8 \quad 0 \\
.8 \times 2 &= 1.6 \quad 1 \\
.6 \times 2 &= 1.2 \quad 1
\end{align*}
\]

so a binary representation for .2 is .001100110011001100110011...

Putting the halves back together again:

64.2 is 1000000.00110011001100110011...

**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 \(2^{n-1} - 1\) where \(n = 8\) bits in this example).

\[
\begin{align*}
6 \quad (E) \\
+ 127 \quad (\text{bias}) \\
\hline
133 \quad (e)
\end{align*}
\]

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 001100110011001100110

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

\[
\begin{align*}
\text{s} & \quad \text{e} \quad \text{f} \\
0 & \quad 10000101 \quad 00000000110011001100110 \\
0b & \quad 0100 0010 1000 0000 0110 0110 0110 0110 \\
0x & \quad 4 \quad 2 \quad 8 \quad 0 \quad 6 \quad 6 \quad 6 \quad 6
\end{align*}
\]
Computers represent real values in a form similar to that of scientific notation. Consider the value $1.23 \times 10^4$

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 consistency.

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:

$\begin{array}{c|c|c}
| s | & e & f \\
\hline
\end{array}$

$s = $ one bit representing the sign of the number
$e = $ an 8-bit biased integer representing the exponent
$f = $ an unsigned integer

The decimal value represented in binary is:

$$s \ E \ (-1) \ x \ F \ x \ 2$$

where

$$E = e - \text{bias} \Rightarrow e = E + \text{bias}$$

$$F = (f/(2^n)) + 1 \Rightarrow f = (F-1) \times 2^n$$

For single precision representation, $n = 23$ bits and the 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 ($e - \text{bias}$).
- 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.
Figure 2.33 Representative values for 6-bit floating-point form where exponent bits and $n = 2$ fraction bits. The bias is 3.

<table>
<thead>
<tr>
<th>Description</th>
<th>Bit representation</th>
<th>Exponent</th>
<th>Fraction</th>
<th>$2^E \times M$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>0 0000 000</td>
<td>0  $\begin{array}{c} -6 \ \frac{1}{64} \end{array}$</td>
<td>$\begin{array}{c} 0 \ \frac{1}{8} \ \frac{1}{8} \end{array}$</td>
<td>$\begin{array}{c} 0 \ 312 \end{array}$</td>
</tr>
<tr>
<td>Smallest pos.</td>
<td>0 0000 001</td>
<td>0  $\begin{array}{c} -6 \ \frac{1}{64} \end{array}$</td>
<td>$\begin{array}{c} 2 \ \frac{2}{8} \ \frac{2}{8} \end{array}$</td>
<td>$\begin{array}{c} 3 \end{array}$</td>
</tr>
<tr>
<td></td>
<td>0 0000 010</td>
<td>0  $\begin{array}{c} -6 \ \frac{1}{64} \end{array}$</td>
<td>$\begin{array}{c} 3 \ \frac{3}{8} \ \frac{3}{8} \end{array}$</td>
<td>$\begin{array}{c} 312 \end{array}$</td>
</tr>
<tr>
<td></td>
<td>0 0000 011</td>
<td>0  $\begin{array}{c} -6 \ \frac{1}{64} \end{array}$</td>
<td>$\begin{array}{c} 7 \ \frac{7}{8} \ \frac{7}{8} \end{array}$</td>
<td>$\begin{array}{c} 312 \end{array}$</td>
</tr>
<tr>
<td>Largest denorm.</td>
<td>0 0000 111</td>
<td>0  $\begin{array}{c} -6 \ \frac{1}{64} \end{array}$</td>
<td>$\begin{array}{c} 7 \ \frac{7}{8} \ \frac{7}{8} \end{array}$</td>
<td>$\begin{array}{c} 312 \end{array}$</td>
</tr>
<tr>
<td>Smallest norm.</td>
<td>0 0001 000</td>
<td>1  $\begin{array}{c} -6 \ \frac{1}{64} \end{array}$</td>
<td>$\begin{array}{c} 0 \ \frac{1}{8} \ \frac{1}{8} \end{array}$</td>
<td>$\begin{array}{c} 8 \ 312 \end{array}$</td>
</tr>
<tr>
<td></td>
<td>0 0001 001</td>
<td>1  $\begin{array}{c} -6 \ \frac{1}{64} \end{array}$</td>
<td>$\begin{array}{c} 1 \ \frac{1}{8} \ \frac{1}{8} \end{array}$</td>
<td>$\begin{array}{c} 312 \end{array}$</td>
</tr>
<tr>
<td>One</td>
<td>0 0110 110</td>
<td>6  $\begin{array}{c} -1 \ \frac{1}{4} \end{array}$</td>
<td>$\begin{array}{c} 6 \ \frac{14}{8} \ \frac{14}{8} \end{array}$</td>
<td>$\begin{array}{c} 14 \ 16 \end{array}$</td>
</tr>
<tr>
<td></td>
<td>0 0110 111</td>
<td>6  $\begin{array}{c} -1 \ \frac{1}{4} \end{array}$</td>
<td>$\begin{array}{c} 7 \ \frac{15}{8} \ \frac{15}{8} \end{array}$</td>
<td>$\begin{array}{c} 16 \end{array}$</td>
</tr>
<tr>
<td>One</td>
<td>0 0111 000</td>
<td>7  1</td>
<td>$\begin{array}{c} 0 \ \frac{1}{8} \ \frac{1}{8} \end{array}$</td>
<td>$\begin{array}{c} 8 \end{array}$</td>
</tr>
<tr>
<td></td>
<td>0 0111 001</td>
<td>7  1</td>
<td>$\begin{array}{c} 1 \ \frac{2}{8} \ \frac{2}{8} \end{array}$</td>
<td>$\begin{array}{c} 8 \end{array}$</td>
</tr>
<tr>
<td></td>
<td>0 0111 010</td>
<td>7  1</td>
<td>$\begin{array}{c} 2 \ \frac{3}{8} \ \frac{3}{8} \end{array}$</td>
<td>$\begin{array}{c} 10 \ 8 \end{array}$</td>
</tr>
<tr>
<td>Largest norm.</td>
<td>0 1110 110</td>
<td>14 7</td>
<td>$\begin{array}{c} 128 \ \frac{14}{8} \ \frac{14}{8} \end{array}$</td>
<td>$\begin{array}{c} 1792 \end{array}$</td>
</tr>
<tr>
<td>Infinity</td>
<td>0 1111 000</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

Figure 2.34 Example nonnegative values for 8-bit floating-point form where $k = 4$ exponent bits and $n = 3$ fraction bits. The bias is 3.
To sum up, the following are the corresponding values for a given representation:

<table>
<thead>
<tr>
<th>Sign</th>
<th>Exponent (e)</th>
<th>Fraction (f)</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00..00</td>
<td>00..00</td>
<td>+0</td>
</tr>
<tr>
<td>0</td>
<td>00..00</td>
<td>00..01 : 11..11</td>
<td>Positive Denormalized Real 0.f × 2^{(e+b+1)}</td>
</tr>
<tr>
<td>0</td>
<td>00..01 : 11..10</td>
<td>XX..XX</td>
<td>Positive Normalized Real 1.f × 2^{(e-b)}</td>
</tr>
<tr>
<td>0</td>
<td>11..11</td>
<td>00..00</td>
<td>+Infinity</td>
</tr>
<tr>
<td>0</td>
<td>11..11</td>
<td>00..01 : 01..11</td>
<td>SNaN</td>
</tr>
<tr>
<td>0</td>
<td>11..11</td>
<td>10..00 : 11..11</td>
<td>QNaN</td>
</tr>
<tr>
<td>1</td>
<td>00..00</td>
<td>00..00</td>
<td>-0</td>
</tr>
<tr>
<td>1</td>
<td>00..00</td>
<td>00..01 : 11..11</td>
<td>Negative Denormalized Real -0.f × 2^{(e+b+1)}</td>
</tr>
<tr>
<td>1</td>
<td>00..01 : 11..10</td>
<td>XX..XX</td>
<td>Negative Normalized Real -1.f × 2^{(e-b)}</td>
</tr>
<tr>
<td>1</td>
<td>11..11</td>
<td>00..00</td>
<td>-Infinity</td>
</tr>
<tr>
<td>1</td>
<td>11..11</td>
<td>00..01 : 01..11</td>
<td>SNaN</td>
</tr>
<tr>
<td>1</td>
<td>11..11</td>
<td>10..00 : 11..11</td>
<td>QNaN</td>
</tr>
</tbody>
</table>

### Special Values

IEEE reserves exponent field values of all 0s and all 1s to denote special values in the floating-point scheme.

#### Zero

As mentioned above, zero is not directly representable in the straight format, due to the assumption of a leading 1 (we'd need to specify a true zero mantissa to yield a value of zero). Zero is a special value denoted with an exponent field of zero and a fraction field of zero. Note that -0 and +0 are distinct values, though they both compare as equal.

#### Denormalized

If the exponent is all 0s, but the fraction is non-zero (else it would be interpreted as zero), then the value is a denormalized number, which does not have an assumed leading 1 before the binary point. Thus, this represents a number (-1)^s × 0.f × 2^{126}, where s is the sign bit and f is the fraction. For double precision, denormalized numbers are of the form (-1)^s × 0.f × 2^{1022}. From this you can interpret zero as a special type of denormalized number.

#### Infinity

The values +infinity and -infinity are denoted with an exponent of all 1s and a fraction of all 0s. The sign bit distinguishes between negative infinity and positive infinity. Being able to denote infinity as a specific value is useful because it allows operations to continue past overflow situations. Operations with infinite values are well defined in IEEE floating point.

#### Not A Number

The value NaN (Not a Number) is used to represent a value that does not represent a real number. NaNs are represented by a bit pattern with an exponent of all 1s and a non-zero fraction. There are two categories of NaN: QNaN (Quiet NaN) and SNaN (Signalling NaN). A QNaN is a NaN with the most significant fraction bit set. QNaNs propagate freely through most arithmetic operations. These values pop out of an operation when the result is not mathematically defined.

An SNaN is a NaN with the most significant fraction bit clear. It is used to signal an exception when used in operations. SNaNs can be handy to assign to uninitialized variables to trap premature usage.

Semantically, QNaN's denote indeterminate operations, while SNaN's denote invalid operations.