**4.1.3 Apply binary notation to represent real numbers.**

*Both fixed-point and floating-point representations are required. Students should be able to calculate the range of normalized floating-point numbers given a specific representation. Issues such as the need for normalization and the loss of precision should be understood.*

JSR Teaching Notes:

This will look at 4.1.2 and 4.1.3 together. From 4.1.2 "State the mantissa and exponent" is just the first step toward 4.1.3's "Apply binary notation to represent real numbers".

Fixed Point vs. Floating Point Representations

The first thing to get straight is the difference between "fixed point" and "floating point" representations.

"Floating"...
Don't get confused and think that the "point" can somehow dynamically "float" within a certain representation - that sometimes it can be after the 5th bit, and other times after the 6th bit - within the same agreed upon system. If an agreed upon floating point representation has 8 bits for the mantissa, and 5 bits for the exponent, that does not change in that particular representation. The point only "floats", in that it moves to the MSB when converted to a "Scientific Notation" kind of form...but this "floating" is only in the programmers mind, as the number converts from one form to another - to the computer, the point is always in the same location for any given specific representation.

They really and truly are two very different ways of representing real numbers:
With fixed, what is to the left of the point is the whole number portion of the number, and what is to the right is the fractional portion of the number. With floating point, there is a part which has the normalized value and a part which stores the exponent that the normalized part must be multiplied by.

Fixed Point Representation

This is easy enough to understand once you appreciate the fact that in a binary system, each numeric place is base 2. So from the point, left, you have:

```
the 2 ^ 0s place,  the 2 ^ 1s place,  the 2 ^ 2s place,  the 2 ^ 3s place,       etc.
```
Put another way, that is:
```
the ones place,    the twos place,    the fours place,   the eights place,       etc.
```

To the right of the point, it's the:
```
the 2 ^ -1s place,         the 2 ^ -2s place,         the 2 ^ -3s place,         etc.
```
Put another way, that is:
```
the 1/2s place,            the 1/4s place,            the 1/8s place,            etc.
```

So if, for example, I look at a binary number in fixed point representation as follows:
`010101.1010`

That's:
```
0 groups of 32s
1 group of 16s
0 groups of 8s
1 group of 4s
0 groups of 2s
1 group of 1s
```
and
```
1 group of 1/2s
0 groups of 1/4s
1 group of 1/8s
0 groups of 1/16
```

or, all together: `16 + 4 + 1 + 1/2 + 1/8 = 21 5/8`

Numbers to be familiar with:
See page 218 for real numbers that can be easily represented by such a system, for example:

| | | | | |
|---|---|---|---|---|
| 0.5 | 0.1 | | | |
| 0.25 | 0.01 | and combinations of these, such as | 0.75 | 0.11 |

Fixed Point Negative Representation

Review of MSB Concept:

We use a Negative MSB (Most Significant Bit) to represent negatives.
For whole numbers, we simply made the MSB to be negative whatever it normally would have been. This balances out nicely since each additional digit in a binary system represents the total number of things that were able to be represented by all the other digits combined. If you still don't quite get that point, consider this. Here's a system with 3 bits, and all the possible combinations:
```
00
01
10
11
```

By adding a third digit, we, in effect, get all of the combinations above with 0 at the front, and all of the combinations above with a 1 at the front. So we get exactly double the possibilities.

So by adding another bit which represents the negative of what that would normally be, it perfectly off-sets the rest. In the above example, adding a (third) MSB of 0, would be adding zero groups of (in this case) -8s. So 0111 would still be 7. But adding a 1 as the (third) MSB would be adding one group of -8s. So 1111 would be a group of -8s added to the remaining total of 7, i.e. -1.

Use of MSB for Fractions in Fixed Point Systems

Note that though you could, the fixed point systems shown in the textbook do not use an MSB for the fractional part. (Though it is important to note that they do for their floating point representations which we'll get to next.)

Floating Point System

Step 1 - Decimal to Binary
For these real number conversions, we still have to do the same first step - convert both the values to the left and to the right to binary - see above.

In this system, we convert the binary representation (as done per the steps above) into a normalized form. This will allow much greater range of real numbers to be represented with the same number of bits. This range is greater both in terms of the largest positive and negative magnitude values, and also the greatest precision, in terms of number of decimal places.

Step 2 - Normalizing
Normalizing (And Compared to Scientific Notation)

Normalizing moves the decimal place to be exactly the same place for every value we work with, and then multiplies that by an exponent value, to account for the movement of the decimal place. As with scientific notation, we'll move the point to be just after the first digit. But note that when using an MSB system, that digit will be often 0. So for positive numbers, you have to remember that the 0 means something, so a positive normalized value begins with a 0, for example, 0.1010 x 2 ^ 110. You would never see that in scientific notation, but in our floating point system it is correct.

Here are a couple of examples of this normalization process:

Example 1:

```
    5.5

  101.1
0.1011 x 2 ^ 3 (to account for the decimal place moved three to the left)
```

Example 2:

```
     17.25

  10001.01
0.1000101 x 2 ^ 5
```

Step 3A - Converting the Exponent to Binary

So far so good, except that there's still a part of our conversion that is not yet converted to binary - the exponent. So we convert it to binary. It is correct in the above examples to say that we moved the decimal places, respectively, 3 and 5 places, and so we need to account for this in the normalized representation. But we simply cannot store "3" and "5" - rather, we can store "11" and "101", respectively. So in our examples:

```
Example 1:
0.1011 x 2 ^ 3 becomes
0.1011 x 2 ^ 11

and Example 2:
0.1000101 x 2 ^ 11 becomes
0.1000101 x 2 ^ 101
```

Step 4 - Placing our answer in the Specific Representation

(Note that so far negatives - both of the mantissa and the exponent - have not been taken into account. More on that shortly.)

This is a crucial stage - it's not simply a matter of copying down your answer with little thought. The mantissa part is in normalized form, the exponent is not. So the mantissa is filled in from left to right. But the exponent is filled in the regular way, from right to left. Before an explanation, here are our two examples each in two different floating point systems.

Floating Point System 1: 8 bits for mantissa, 4 for exponent:

```
Example 1:
01011000.0011
and
Example 2:
01000101.0101
```

Floating Point System 2: 16 bits for mantissa, 10 for exponent:

```
Example 1:
0101100000000000.0000000011
and
Example 2:
0100010100000000.0000000101
```

It's pretty clear by comparing the two examples that the exponents are filled in the regular way, with extra 0s going to the left. That the opposite is true for the mantissa is equally clear when you are reminded that the first 0 is for the MSB. See the following section on that, but the point is that you fill in the extra 0s for the mantissa going to the right.

Step 3B - Negatives (both Mantissa and Exponent)

*Negative Mantissa for Floating Point*

Not a big deal here, just remember to do that step in-between steps 3 and 4 above. And do it the same way you did before with the 2s compliment. See full example below.

*Negative Exponent for Floating Point*

To represent very small numbers in a floating point representation, we will actually have to use a negative exponent to normalize correctly. So the MSB (the bit furthest to the left) for the exponent will be negative what it otherwise would have been in this system. For example, with 6 digits for the exponent, the MSB would be -32. So the mantissa will end up being multiplied not by a power of 2 (potentially a very big value) but 1 divided by a power of 2 (potentially a very small fractional value).

We do this because rather than normalizing by moving the decimal not to the left, you are moving it to the right. Consider the very small value 0.00001 - to normalize that number we move the decimal to the right 4 times, so the normalized value of 0.1 would be multiplied by 1/2^4 -- or put another way 2 ^ -4.

See the Convert 0.1875 example below.

Note that the use of MSB for both parts of the representation is for floating point only, the way that IB does it in the text book. (Recall that the fractions of the right hand side of fixed point are all positive in the way we do it.)

Full examples:

Here are the steps, in summary one more time:

- Convert to binary, both sides of the radix (disregard negatives considerations completely here)
- Normalize
- Convert the exponent that 2 is raised to to be binary
- If you're working with a negative, apply the 2s Compliment to the mantissa portion only.
- Be sure to place the mantissa digits in the correct places, remembering to fill in the mantissa from the left (the -1s, then the 1/2s, the 1/4s etc.) and fill in any extra zeros, if needed after that, before the radix.

****But, don't do this for the exponent part – you fill this in the normal way**.***

Convert 35.5 to binary using a floating point representation which has a mantissa of 8 bits, including an MSB, and 5 bits for the exponent, including an MSB.

```
     35.5
  100011.1
0.1000111 x 2 ^ 6
0.1000111 x 2 ^ 110

 01000111.00110
 --------.-----
```

Convert -5.125 to binary using a floating-point representation which has a mantissa of 8 bits, including an MSB, and 5 bits for the exponent, including an MSB.

```
  -5.125
(Do it as positive first - remember, disregard negatives and MSBs completely at this point)
   101.001
0.101001 x 2 ^ 3
0.101001 x 2 ^ 11

1.010110 (Now do the 2s compliment)
+_____1
1.010111 x 2 ^ 11

10101110.00011
--------.-----
```

Convert 0.1875 to binary using a floating-point representation which has a mantissa of 10 bits, including an MSB, and 7 bits for the exponent, including an MSB.

```
0.1875
0.0011        (You figure out 0.1875 must be 0.125 + 0.0625 from looking on page 218.
               and so 0.001 + 0.0001 = 0.0011 with no MSB yet)
0.11 x 2 ^ -2     (Normalizing, only this time with the decimal to the right...)
0.11 x 2 ^ -1111101     (Exponent now in binary - 2s compliment method not shown)

0110000000.1111101
----------.-------
```

Checking Your Work.

This is actually quite straight-forward, though tedious and prone to careless errors. You first simply translate each 0 and 1 given its place, remembering that (in systems with an MSB) the mantissa is, FROM THE LEFT: -1s, 1/2s, 1/4s etc., and that the exponent is, FROM THE RIGHT: 1s, 2s, 4s, all the way up to next to the decimal point, a -MSB. And having translated them, you calculate out the result by multiplying the mantissa by the exponent.

The examples above:

```
35.5 is
01000111.00110

 0    1     0     0     0      1      1      1           0   0   1   1   0
-1s  1/2s  1/4s  1/8s  1/16s  1/32s  1/64s  1/128s  .  -16s  8s  4s  2s  1s
     64/128         +         4/128   +     1/128   x           2 ^ 6
     71/128 x 64
     35.5
```

```
-5.125 is
10101110.00011


 1    0     1     0     1      1      1      0           0   0   0   1   1
-1s  1/2s  1/4s  1/8s  1/16s  1/32s  1/64s  1/128s  .  -16s  8s  4s  2s  1s
-64/64  + 16/64     +   4/64 + 2/64 + 1/64          x              2 ^ 3
     -41/64 x 8
     -5.125


0.1875 is
0110000000.1000010

 0    1     1     0   etc.       1    1    1   1   1   1   0
-1s  1/2s  1/4s  1/8s etc.  .  -64s  32s  16s  8s  4s  2s  1s
     2/4 + 1/4              x                        2 ^ - 2 (-64 + 62)
       3/4 x 1/2^2
     0.1875
```

Finding the Range of Numbers Possible

You can be asked for the "range" of numbers possible to be represented by a given system, i.e. fixed point, or floating point, and a certain number of bits for (fixed) whole and factional portions, or (floating point) mantissa and exponent.

Fixed Point, MSB, 6 & 4 system Example

Largest Positive and Negative Magnitude:

```
011111.0111
 31 + 1/4 + 1/8 + 1/16
 = 31 15/16
 = 31.9375

100000.0111
 -32
 = -32 15/16
 = -32.9375
```

Largest Precision (Or lowest magnitude -we'll do closest to 0):

```
000000.0001
1/16

000000.1111
-1/16
```

BUT YOU DON'T HAVE TO BE ABLE TO DO THE FIXED POINT RANGE CALCULATIONS, ONLY FLOATING POINT:

Floating Point, MSB, 6 & 4 system Example

Largest Positive and Negative Magnitude:

```
011111.0111
 = 31/32 x 2 ^ 7
 = 0.96875 x 128
 = 124

100000.0111
 = -1 x 2 ^ 7
 = -1 x 128
 = -128
```

Largest Precision (Or lowest magnitude - we'll do closest to 0):

See the bottom of 225 - I can't offer much more explanation to this one. But there are two keys to understanding this:

First, remember that you want to multiply by the largest negative exponent, which will always be .100... however many 0s there are for the exponent portion.

Secondly, remember that you want to multiply that exponent by the smallest possible number. But you can't use 0, and you must have a normalized number. So 00001 won't count. For positives, 010000 is the best you can do.

One Additional Important Note

I'll always write the normalized values as 01111 and 10000, but the text often writes them as 0.1111 and 1.0000. Also, sometimes that decimal is shown, and others, just a space. So, when doing exam review, we'll have to note that way that has been used most often in the past.