WEEK- 8

2.4 Floating Point Representation.

• The signed magnitude, one’s complement, and two’s complement representation that we have just presented deal with integer values only.

• Without modification, these formats are not useful in scientific or business applications that deal with real number values.

• Floating-point representation solves this problem.

• If we are clever programmers, we can perform floating-point calculations using any integer format.

• This is called *floating-point emulation*, because floating point values aren’t stored as such, we just create programs that make it seem as if floating point values are being used.

• Most of today’s computers are equipped with specialized hardware that performs floating-point arithmetic with no special programming required.

• Floating-point numbers allow an arbitrary number of decimal places to the right of the decimal point.

– For example: $0.5 \times 0.25 = 0.125$

• They are often expressed in scientific notation.

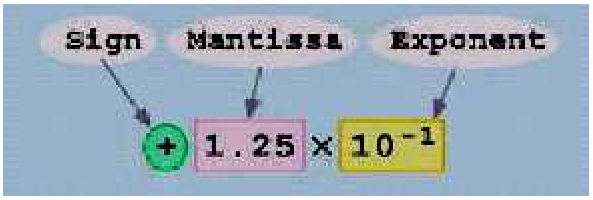
– For example:

$0.125 = 1.25 \times 10-1$

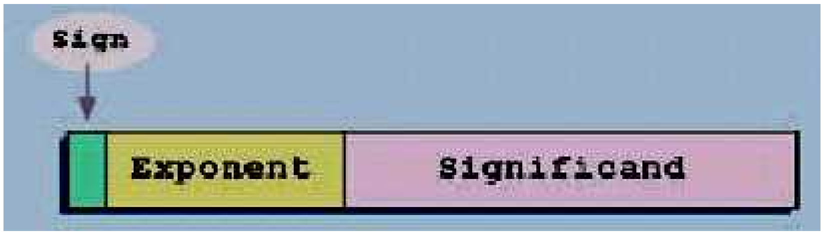
$5,000,000 = 5.0 \times 10\text{^}6$

• Computers use a form of scientific notation for floating-point representation

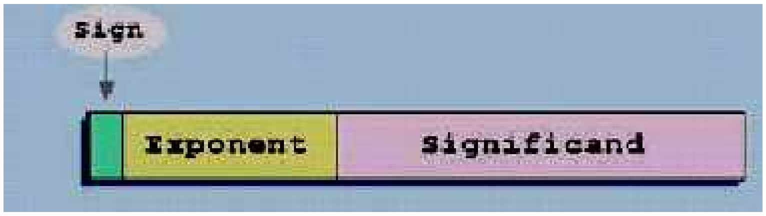
• Numbers written in scientific notation have three components:



• Computer representation of a floating-point number consists of three fixed-size fields:



• This is the standard arrangement of these fields.



• The one-bit sign field is the sign of the stored value.

•The size of the exponent field, determines the range of values that can be represented.

The size of the significant determines the precision of the representation.

• The IEEE-754 *single precision* floating point standard uses an 8-bit exponent and a 23-bit significant.

• The IEEE-754 *double precision* standard uses an 11-bit exponent and a 52-bit significant.

For illustrative purposes, we will use a 14-bit model with a 5-bit exponent and an 8-bit significant.

• The significant of a floating-point number is always preceded by an implied binary point.

• Thus, the significant always contains a fractional binary value.

• The exponent indicates the power of 2 to which the significant is raised.

• Example:

– Express 32 in the simplified 14-bit floating-point model.

• We know that 32 is 2^5. So in (binary) scientific notation

32 = 1.0 x 2^5 = 0.1 x 2^6.

• Using this information, we put 110 = $(6)_{10}$ in the exponent field and 1 in the significant as shown.

• Another problem with our system is that we have made no allowances for negative exponents. We have no way to express 0.5 = (2^ -1)!

(Notice that there is no sign in the exponent field!)

All of these problems can be fixed with no changes to our basic model.

• To resolve the problem of synonymous forms, we will establish a rule that the first digit of the significant must be 1. This results in a unique pattern for each floating-point number.

In the IEEE-754 standard, this 1 is implied meaning that a 1 is assumed after the binary point.

By using an implied 1, we increase the precision of the representation by a power of two. (Why?)**.**

In our simple instructional model, we will use no implied bits.

• To provide for negative exponents, we will use a *biased exponent*.

• A bias is a number that is approximately midway in the range of values expressible by the exponent. We subtract the bias from the value in the exponent to determine its true value.

– In our case, we have a 5-bit exponent. We will use 16 for our bias. This is called *excess-16* representation.

In our model, exponent values less than 16 are negative, representing fractional numbers.

• Example:

Express $(32)_{10}$ in the revised 14-bit floating-point model.

• We know that 32 = 1.0 x 2^5 = 0.1 x 2^6.

• To use our excess 16 biased exponent, we add 16 to 6, giving

$$(22)_{10} = (10110)_2.$$

• Graphically:

• Example:

– Express 0.062510 in the revised 14-bit floating-point model.

• We know that 0.0625 is 2^-4. So in (binary) scientific notation

0.0625 = 1.0 x 2^-4 = 0.1 x 2^ -3.

• To use our excess 16 biased exponent, we add 16 to -3, giving

$(13)_{10} = (01101)_2$.

• Example:

– Express -26.62510 in the revised 14-bit floating-point model.

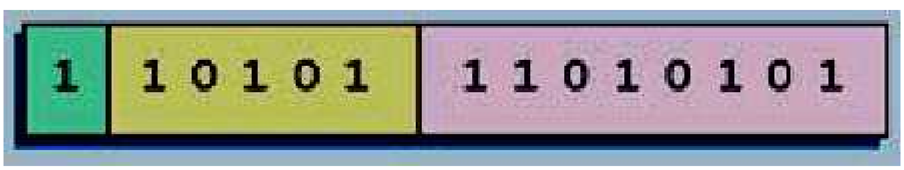
• We find 26.62510 = 11010.101 x 2^0. Normalizing, we have:

$(26.625)_{10} = 0.11010101 \times 2^{5}$.

• To use our excess 16 biased exponent, we add 16 to 5, giving

$(21)_{10} = (10101)_2$.

We also need a 1 in the sign bit.



• Floating-point addition and subtraction are done using methods analogous to how we perform calculations using pencil and paper.

• The first thing that we do to express both operands in the same exponential power, then add the numbers, preserving the exponent in the sum.

• If the exponent requires adjustment, we do so at the end of the calculation.

• Example:

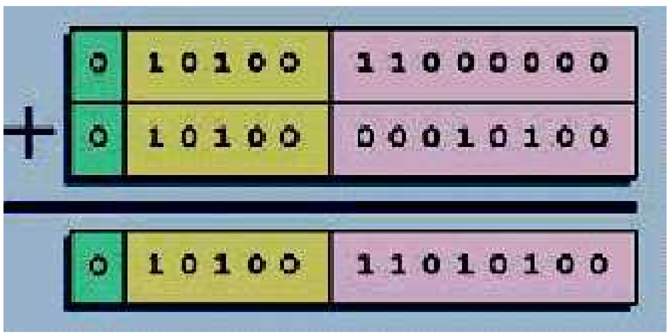
– Find the sum of $(12)_{10}$ and $(1.25)_{10}$ using the 14-bit floating point model.

• We find

$(12)_{10}$ = 0.1100 x 2^ 4. And $(1.25)_{10}$ = 0.101 x 2 ^1

=0.000101 x 2^ 4.**2.5**

• Thus, our sum is 0.110101 x 2 ^4.



• Floating-point multiplication is also carried out in a manner a kind to how we perform multiplication using pencil and paper.

• We multiply the two operands and add their exponents.

• If the exponent requires adjustment, we do so at the end of the calculation.

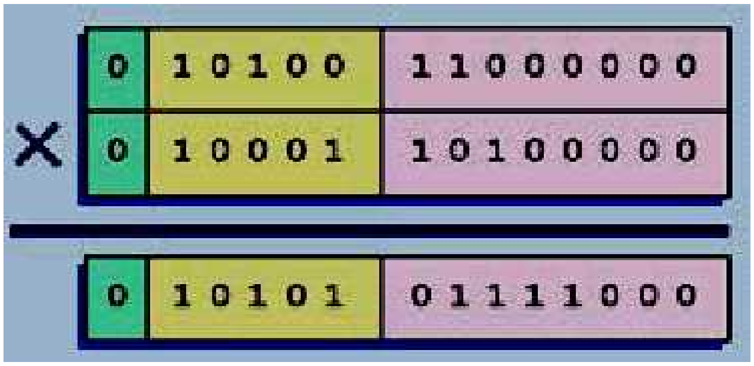
• Example:

– Find the product of $(12)_{10}$ and $(1.25)_{10}$ using the 14-bit floating point model.

• We find $(12)_{10}$ = 0.1100 x 2^ 4 and $(1.25)_{10}$ = 0.101 x 2^1.

• Thus, our product is

0.0111100 x 2^5 = 0.1111 x 2^4.



• The normalized product requires an exponent of

$$(20)_{10} = (10110)_2.$$

• No matter how many bits we use in a floating-point representation, our model must be finite.

• The real number system is, of course, infinite, so our models can give nothing more than an approximation of a real value.

• At some point, every model breaks down, introducing errors into our calculations.

• By using a greater number of bits in our model, we can reduce these errors, but we can never totally eliminate them.