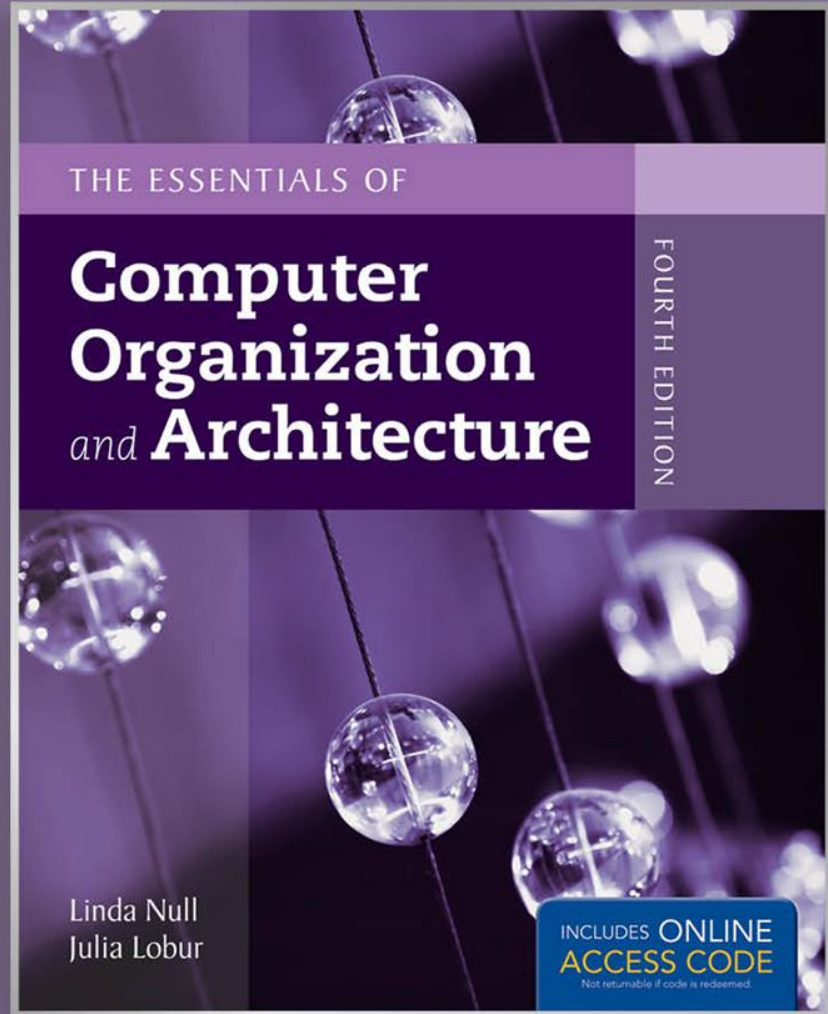


Chapter 2

Data Representation in Computer Systems



Chapter 2 Objectives (2b)

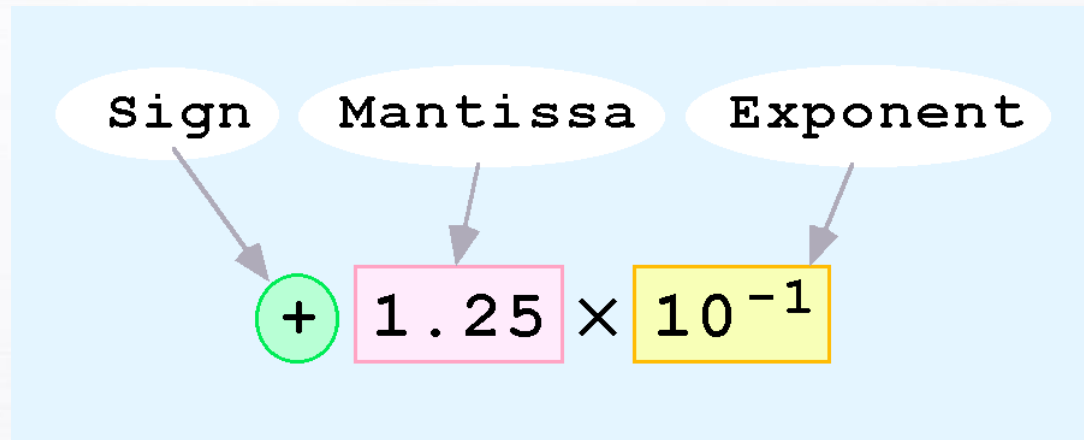
- Understand the fundamental concepts of floating-point representation.

2.5 Floating-Point Representation

- 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:
 $0.125 = 1.25 \times 10^{-1}$
 $5,000,000 = 5.0 \times 10^6$

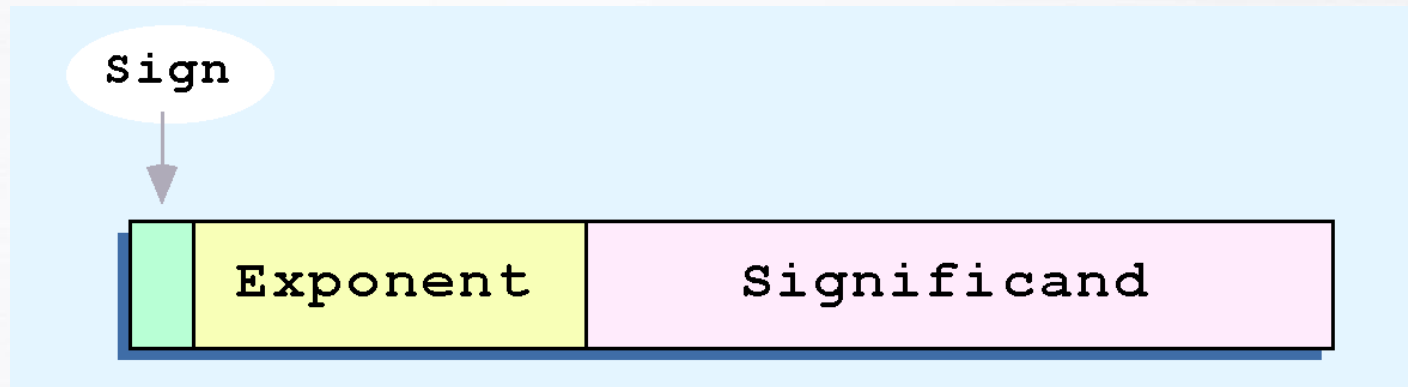
2.5 Floating-Point Representation

- Computers use a form of scientific notation for floating-point representation
- Numbers written in scientific notation have three components:



2.5 Floating-Point Representation

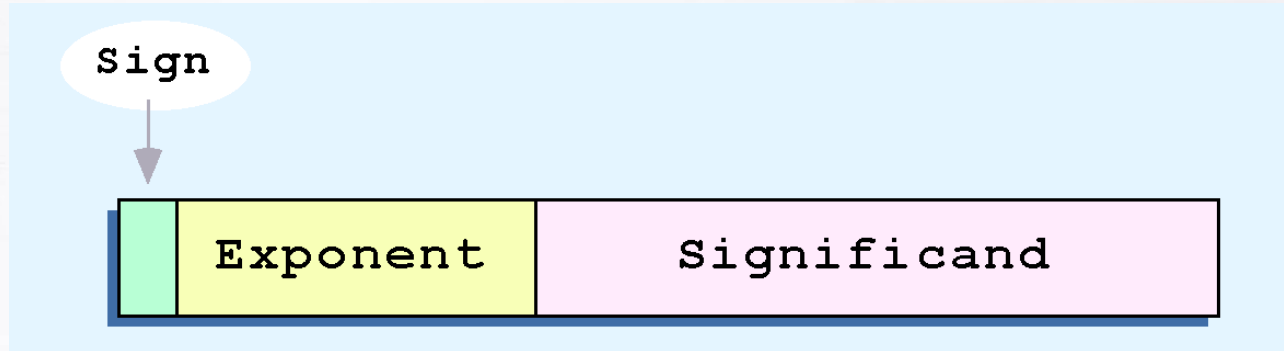
- Computer representation of a floating-point number consists of three fixed-size fields:



- This is the standard arrangement of these fields.

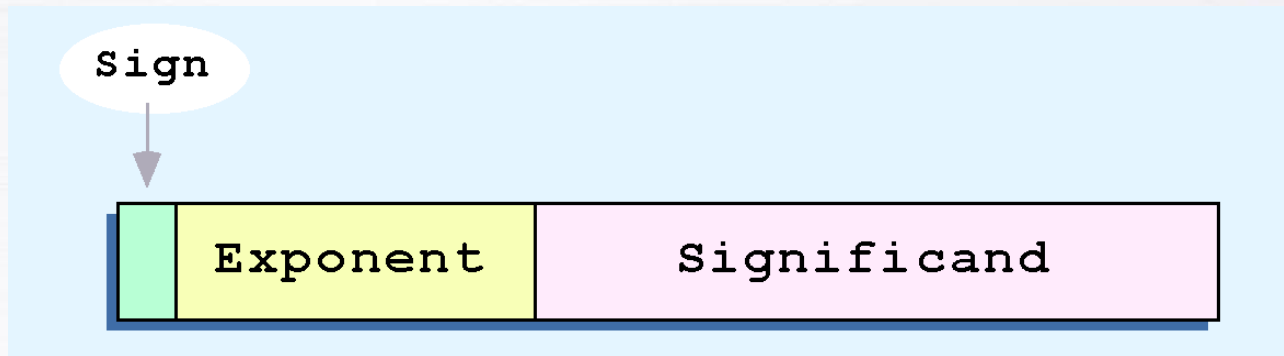
Note: Although “significand” and “mantissa” do not technically mean the same thing, many people use these terms interchangeably. We use the term “significand” to refer to the fractional part of a floating point number.

2.5 Floating-Point Representation



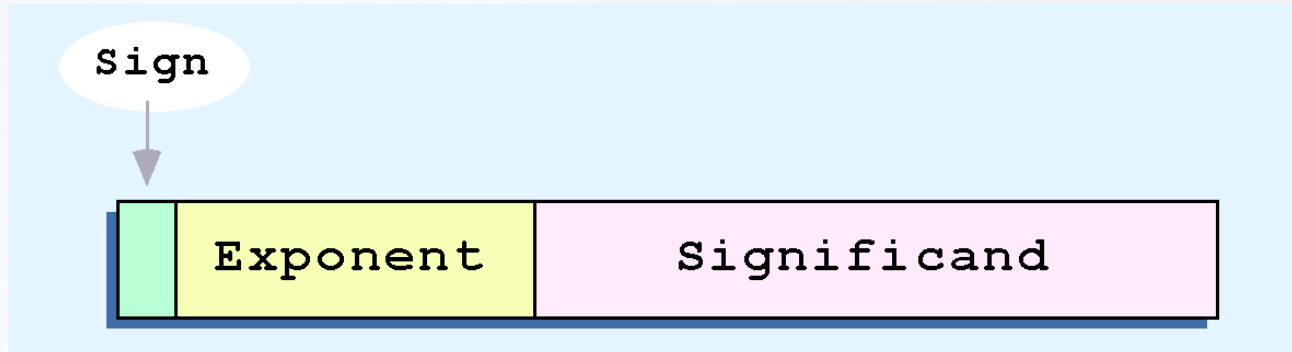
- The one-bit sign field is the sign of the stored value.
- The size of the exponent field determines the range of values that can be represented.
- The size of the significand determines the precision of the representation.

2.5 Floating-Point Representation



- We introduce a hypothetical “Simple Model” to explain the concepts
- In this model:
 - A floating-point number is 14 bits in length
 - The exponent field is 5 bits
 - The significand field is 8 bits

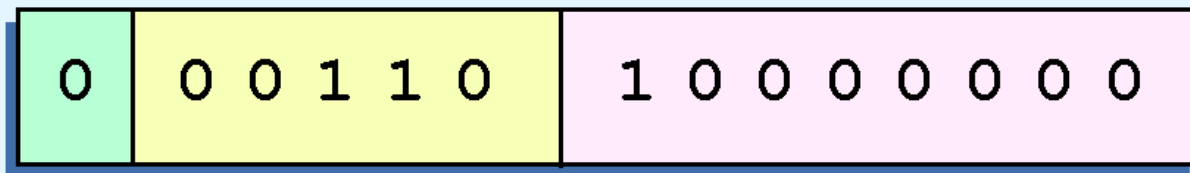
2.5 Floating-Point Representation



- The significand is always preceded by an implied binary point.
- Thus, the significand always contains a fractional binary value.
- The exponent indicates the power of 2 by which the significand is multiplied.

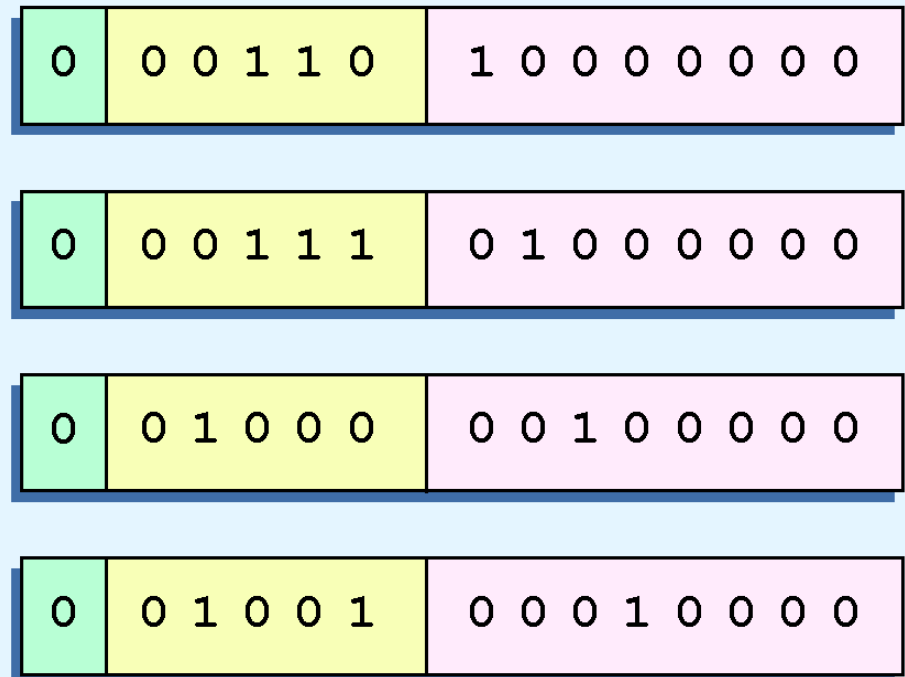
2.5 Floating-Point Representation

- Example:
 - Express 32_{10} in the simplified 14-bit floating-point model.
- We know that 32 is 2^5 . So in (binary) scientific notation $32 = 1.0 \times 2^5 = 0.1 \times 2^6$.
- Using this information, we put 110 ($= 6_{10}$) in the exponent field and 1 in the significand as shown.

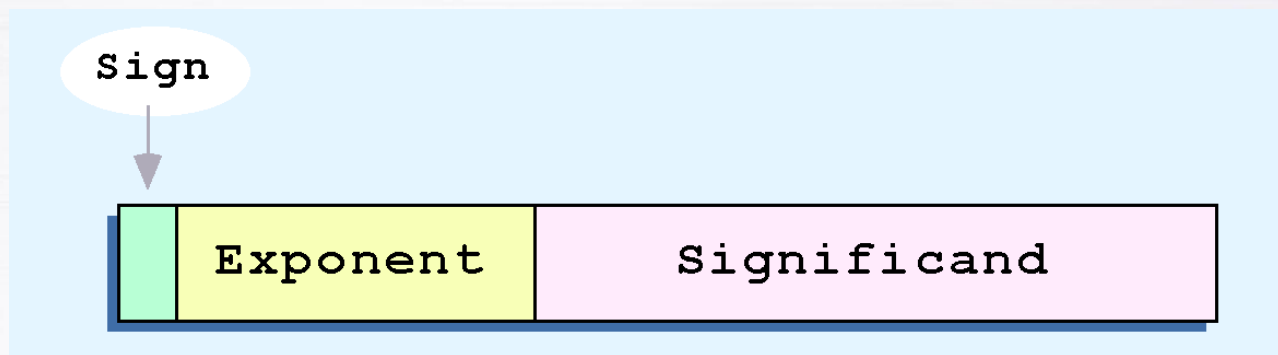


2.5 Floating-Point Representation

- The illustrations shown at the right are *all* equivalent representations for 32 using our simplified model.
- Not only do these synonymous representations waste space, but they can also cause confusion.



2.5 Floating-Point Representation



- 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.

2.5 Floating-Point Representation

- To resolve the problem of synonymous forms, we establish a rule that the first digit of the significand must be 1, with no ones to the left of the radix point.
- This process, called *normalization*, results in a unique pattern for each floating-point number.
 - In our simple model, all significands must have the form 0.1xxxxxxxx
 - For example, $4.5 = 100.1 \times 2^0 = 1.001 \times 2^2 = 0.1001 \times 2^3$. The last expression is correctly normalized.

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

2.5 Floating-Point Representation

- To provide for negative exponents, we will use a *biased exponent*.
 - In our case, we have a 5-bit exponent.
 - $2^{5-1} - 1 = 2^4 - 1 = 15$
 - Thus will use 15 for our bias: our exponent will use *excess-15* representation.
- In our model, exponent values less than 15 are negative, representing fractional numbers.

Excess-M Representation

- Excess-M representation (also called offset binary representation) is another way for unsigned binary values to represent signed integers.
 - Excess-M representation is intuitive because the binary string with all 0s represents the smallest number, whereas the binary string with all 1s represents the largest value.
- An unsigned binary integer M (called the *bias*) represents the value 0, whereas all zeroes in the bit pattern represents the integer $2M$.
- The integer is interpreted as positive or negative depending on where it falls in the range.

Excess-M Representation

- If n bits are used for the binary representation, we select the bias in such a manner that we split the range equally.
- Typically we choose a bias of $2^{n-1} - 1$.
 - For example, if we were using 4-bit representation, the bias should be $2^{4-1} - 1 = 7$.
- Just as with signed magnitude, one's complement, and two's complement, there is a specific range of values that can be expressed in n bits.

Excess-M Representation

- The unsigned binary value for a signed integer using excess-M representation is determined simply by adding M to that integer.
 - For example, assuming that we are using excess-7 representation, the integer 0_{10} is represented as $0 + 7 = 7_{10} = 0111_2$.
 - The integer 3_{10} is represented as $3 + 7 = 10_{10} = 1010_2$.
 - The integer -7 is represented as $-7 + 7 = 0_{10} = 0000_2$.
 - To find the decimal value of the excess-7 binary number 1111_2 subtract 7: $1111_2 = 15_{10}$ and $15 - 7 = 8$; thus 1111_2 , in excess-7 is $+8_{10}$.

Signed Integer Representation

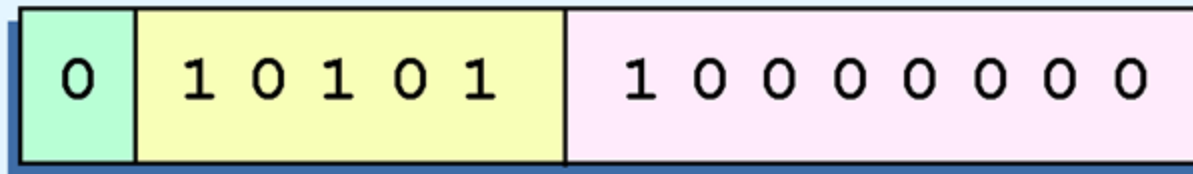
- Lets compare our representations:

Decimal	Binary (for absolute value)	Signed Magnitude	One's Complement
2	00000010	00000010	00000010
-2	00000010	10000010	11111101
100	01100100	01100100	01100100
-100	01100100	11100100	10011011

Decimal	Binary (for absolute value)	Two's Complement	Excess-127
2	00000010	00000010	10000001
-2	00000010	11111110	01111101
100	01100100	01100100	11100011
-100	01100100	10011100	00011011

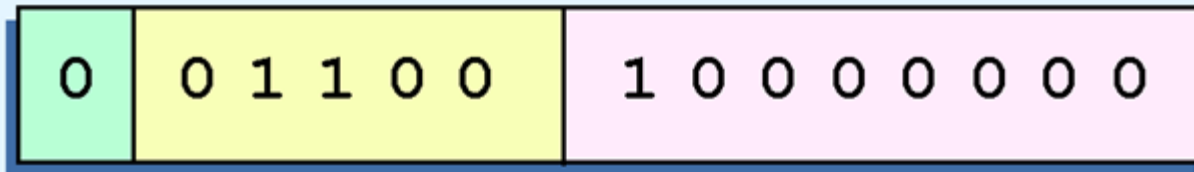
2.5 Floating-Point Representation

- Example:
 - Express 32_{10} in the revised 14-bit floating-point model.
- We know that $32 = 1.0 \times 2^5 = 0.1 \times 2^6$.
- To use our excess 15 biased exponent, we add 15 to 6, giving 21_{10} ($=10101_2$).
- So we have:



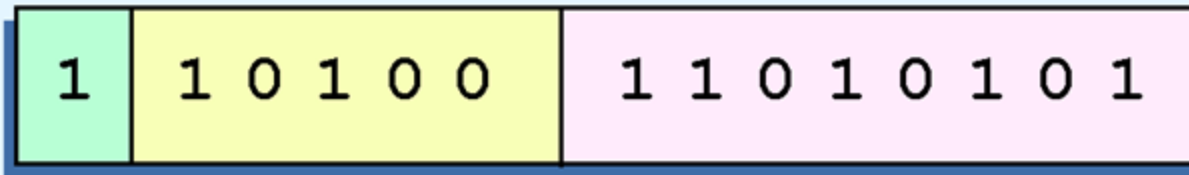
2.5 Floating-Point Representation

- Example:
 - Express 0.0625_{10} in the revised 14-bit floating-point model.
- We know that 0.0625 is 2^{-4} . So in (binary) scientific notation $0.0625 = 1.0 \times 2^{-4} = 0.1 \times 2^{-3}$.
- To use our excess 15 biased exponent, we add 15 to -3 , giving 12_{10} ($=01100_2$).



2.5 Floating-Point Representation

- Example:
 - Express -26.625_{10} in the revised 14-bit floating-point model.
- We find $26.625_{10} = 11010.101_2$. Normalizing, we have: $26.625_{10} = 0.11010101 \times 2^5$.
- To use our excess 15 biased exponent, we add 15 to 5, giving 20_{10} ($=10100_2$). We also need a 1 in the sign bit.



2.5 Floating-Point Representation

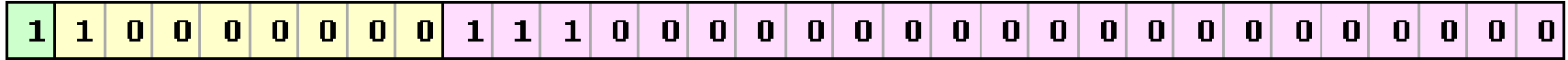
- The IEEE has established a standard for floating-point numbers
- The IEEE-754 *single precision* floating point standard uses an 8-bit exponent (with a bias of 127) and a 23-bit significand.
- The IEEE-754 *double precision* standard uses an 11-bit exponent (with a bias of 1023) and a 52-bit significand.

2.5 Floating-Point Representation

- In both the IEEE single-precision and double-precision floating-point standard, the significant has an implied 1 to the LEFT of the radix point.
 - The format for a significand using the IEEE format is:
 $1.xxx...$
 - For example, $4.5 = .1001 \times 2^3$ in IEEE format is $4.5 = 1.001 \times 2^2$. The 1 is implied, which means it does not need to be listed in the significand (the significand would include only 001).

2.5 Floating-Point Representation

- Example: Express -3.75 as a floating point number using IEEE single precision.
- First, let's normalize according to IEEE rules:
 - $3.75 = -11.11_2 = -1.111 \times 2^1$
 - The bias is 127, so we add $127 + 1 = 128$ (this is our exponent)



(implied)

- Since we have an implied 1 in the significand, this equates to
$$-(1).111_2 \times 2^{(128 - 127)} = -1.111_2 \times 2^1 = -11.11_2 = -3.75.$$

2.5 Floating-Point Representation

- Using the IEEE-754 single precision floating point standard:
 - An exponent of 255 indicates a special value.
 - If the significand is zero, the value is \pm infinity.
 - If the significand is nonzero, the value is NaN, “not a number,” often used to flag an error condition.
- Using the double precision standard:
 - The “special” exponent value for a double precision number is 2047, instead of the 255 used by the single precision standard.

2.5 Floating-Point Representation

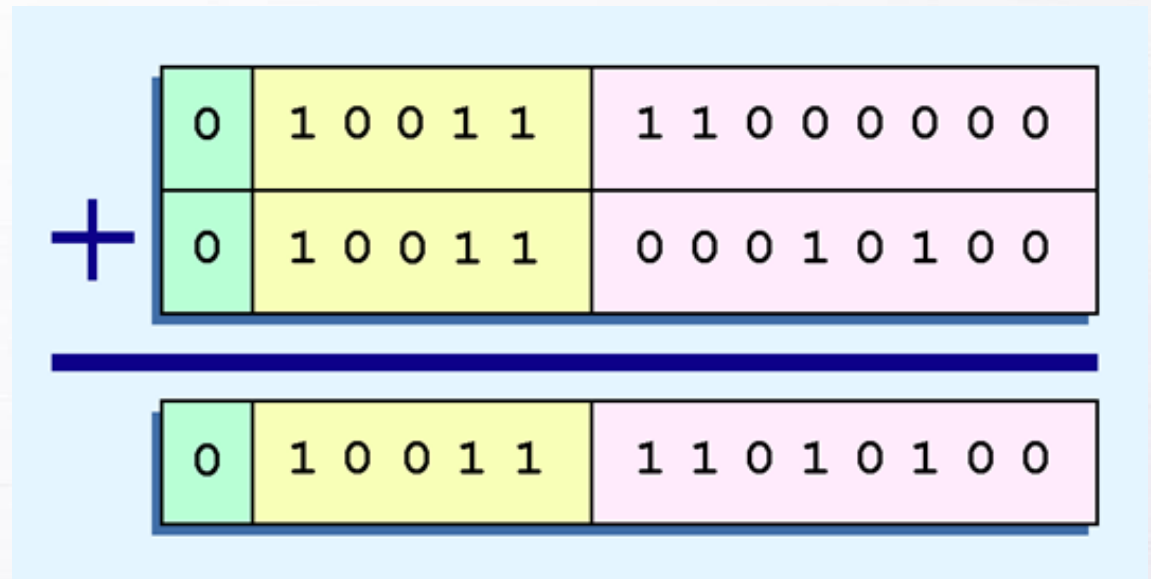
- Both the 14-bit model that we have presented and the IEEE-754 floating point standard allow two representations for zero.
 - Zero is indicated by all zeros in the exponent and the significand, but the sign bit can be either 0 or 1.
- This is why programmers should avoid testing a floating-point value for equality to zero.
 - Negative zero does not equal positive zero.

2.5 Floating-Point Representation

- 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 is 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.

2.5 Floating-Point Representation

- Example:
 - Find the sum of 12_{10} and 1.25_{10} using the 14-bit “simple” floating-point model.
- We find $12_{10} = 0.1100 \times 2^4$. And $1.25_{10} = 0.101 \times 2^1 = 0.000101 \times 2^4$.
- Thus, our sum is 0.110101×2^4 .

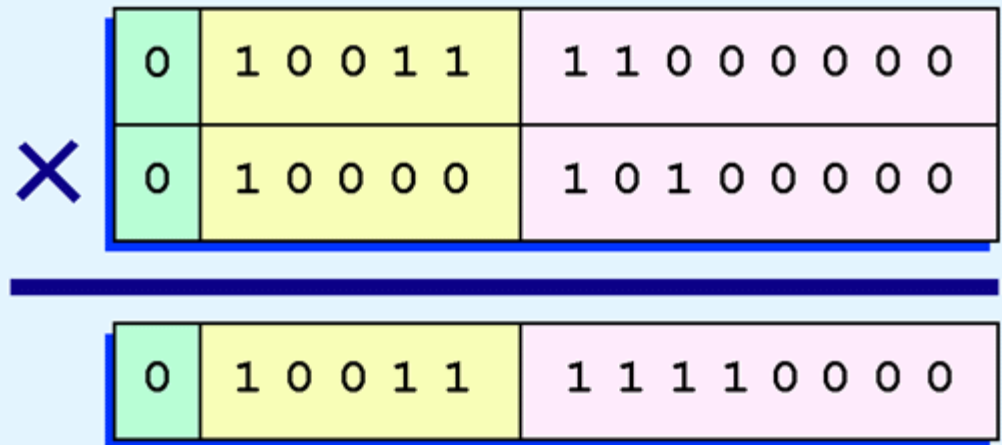


2.5 Floating-Point Representation

- Floating-point multiplication is also carried out in a manner akin 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.

2.5 Floating-Point Representation

- Example:
 - Find the product of 12_{10} and 1.25_{10} using the 14-bit floating-point model.
- We find $12_{10} = 0.1100 \times 2^4$. And $1.25_{10} = 0.101 \times 2^1$.
- Thus, our product is $0.0111100 \times 2^5 = 0.1111 \times 2^4$.
- The normalized product requires an exponent of $19_{10} = 10011_2$.



2.5 Floating-Point Representation

- 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.

2.5 Floating-Point Representation

- Our job becomes one of reducing error, or at least being aware of the possible magnitude of error in our calculations.
- We must also be aware that errors can compound through repetitive arithmetic operations.
- For example, our 14-bit model cannot exactly represent the decimal value 128.5. In binary, it is 9 bits wide:

$$10000000.1_2 = 128.5_{10}$$

2.5 Floating-Point Representation

- When we try to express 128.5_{10} in our 14-bit model, we lose the low-order bit, giving a relative error of:

$$\frac{128.5 - 128}{128.5} \approx 0.39\%$$

- If we had a procedure that repetitively added 0.5 to 128.5, we would have an error of nearly 2% after only four iterations.

2.5 Floating-Point Representation

- Floating-point errors can be reduced when we use operands that are similar in magnitude.
- If we were repetitively adding 0.5 to 128.5, it would have been better to iteratively add 0.5 to itself and then add 128.5 to this sum.
- In this example, the error was caused by loss of the low-order bit.
- Loss of the high-order bit is more problematic.

2.5 Floating-Point Representation

- Floating-point overflow and underflow can cause programs to crash.
- Overflow occurs when there is no room to store the high-order bits resulting from a calculation.
- Underflow occurs when a value is too small to store, possibly resulting in division by zero.

Experienced programmers know that it's better for a program to crash than to have it produce incorrect, but plausible, results.

2.5 Floating-Point Representation

- When discussing floating-point numbers, it is important to understand the terms *range*, *precision*, and *accuracy*.
- The range of a numeric integer format is the difference between the largest and smallest values that can be expressed.
- Accuracy refers to how closely a numeric representation approximates a true value.
- The precision of a number indicates how much information we have about a value

2.5 Floating-Point Representation

- Most of the time, greater precision leads to better accuracy, but this is not always true.
 - For example, 3.1333 is a value of pi that is accurate to two digits, but has 5 digits of precision.
- There are other problems with floating point numbers.
- Because of truncated bits, you cannot always assume that a particular floating point operation is commutative or distributive.

2.5 Floating-Point Representation

- This means that we cannot assume:

$$(a + b) + c = a + (b + c) \text{ or}$$

$$a*(b + c) = ab + ac$$

- Moreover, to test a floating point value for equality to some other number, it is best to declare a “nearness to x” epsilon value. For example, instead of checking to see if floating point x is equal to 2 as follows:

if $x = 2$ then ...

it is better to use:

if $(\text{abs}(x - 2) < \text{epsilon})$ then ...

(assuming we have epsilon defined correctly!)