2.4 Signed Integer Representation

- For example, in 8-bit signed magnitude representation:

  +3 is: `00000011`

  - 3 is: `10000011`

- Computers perform arithmetic operations on signed magnitude numbers in much the same way as humans carry out pencil and paper arithmetic.
  - Humans often ignore the signs of the operands while performing a calculation, applying the appropriate sign after the calculation is complete.

© 2012 Jones & Bartlett Learning, LLC
www.jblearning.com

2.4 Signed Integer Representation

- Binary addition is as easy as it gets. You need to know only four rules:

```
0 + 0 = 0      0 + 1 = 1
1 + 0 = 1      1 + 1 = 10
```

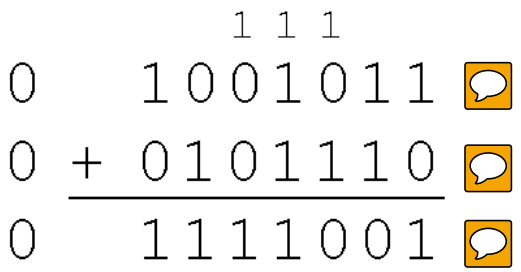
- The simplicity of this system makes it possible for digital circuits to carry out arithmetic operations.

Let's see how the addition rules work with signed magnitude numbers . . .

© 2012 Jones & Bartlett Learning, LLC
www.jblearning.com

2.4 Signed Integer Representation

- Example:
  - Using signed magnitude binary arithmetic, find the sum of 75 and 46.
- Once we have worked our way through all eight bits, we are done.

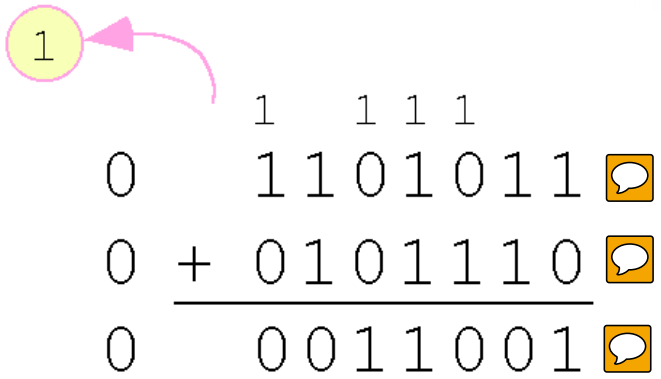


In this example, we were careful to pick two values whose sum would fit into seven bits. If that is not the case, we have a problem.

© 2012 Jones & Bartlett Learning, LLC
www.jblearning.com

2.4 Signed Integer Representation

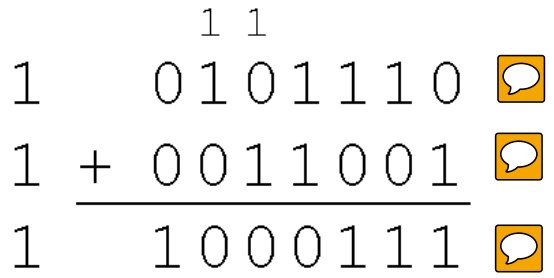
- Example:
  - Using signed magnitude binary arithmetic, find the sum of 107 and 46.
- We see that the carry from the seventh bit *overflows* and is discarded, giving us the erroneous result: 107 + 46 = 25.



© 2012 Jones & Bartlett Learning, LLC
www.jblearning.com

2.4 Signed Integer Representation

- The signs in signed magnitude representation work just like the signs in pencil and paper arithmetic.
  - Example: Using signed magnitude binary arithmetic, find the sum of - 46 and - 25.

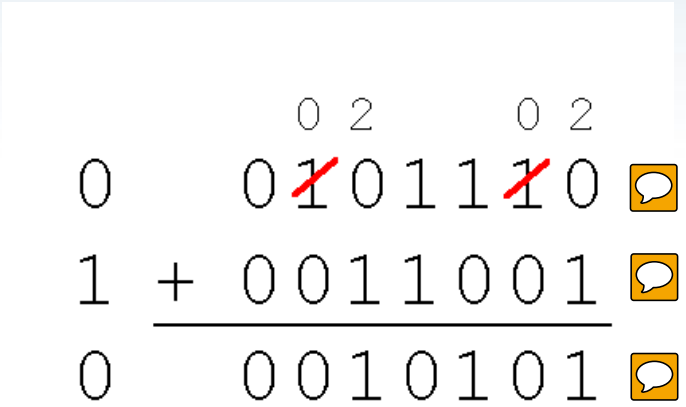


- Because the signs are the same, all we do is add the numbers and supply the negative sign when we are done.

© 2012 Jones & Bartlett Learning, LLC
www.jblearning.com

2.4 Signed Integer Representation

- Mixed sign addition (or subtraction) is done the same way.
  - Example: Using signed magnitude binary arithmetic, find the sum of 46 and - 25.



- The sign of the result gets the sign of the number that is larger.
  - Note the "borrows" from the second and sixth bits.

© 2012 Jones & Bartlett Learning, LLC
www.jblearning.com

2.4 Signed Integer Representation

- For example, using 8-bit one's complement representation:

  + 3 is: `00000011`

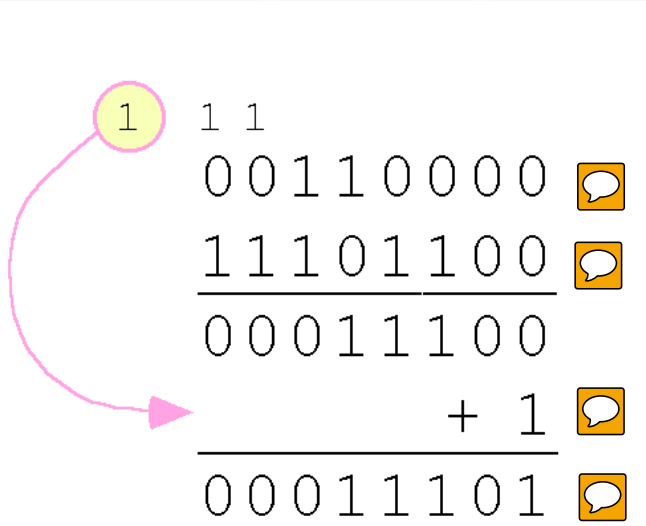
  - 3 is: `11111100`

- In one's complement representation, as with signed magnitude, negative values are indicated by a 1 in the high order bit.
- Complement systems are useful because they eliminate the need for subtraction. The difference of two values is found by adding the minuend to the complement of the subtrahend.

© 2012 Jones & Bartlett Learning, LLC
www.jblearning.com

2.4 Signed Integer Representation

- With one's complement addition, the carry bit is "carried around" and added to the sum.
  - Example: Using one's complement binary arithmetic, find the sum of 48 and - 19



We note that 19 in binary is `00010011`,
so -19 in one's complement is: `11101100`.

© 2012 Jones & Bartlett Learning, LLC
www.jblearning.com

2.4 Signed Integer Representation

- To express a value in two's complement representation:
  - If the number is positive, just convert it to binary and you're done.
  - If the number is negative, find the one's complement of the number and then add 1.
- Example:
  - In 8-bit binary, 3 is:
    `00000011`
  - -3 using one's complement representation is:
    `11111100`
  - Adding 1 gives us -3 in two's complement form:
    `11111101`.

© 2012 Jones & Bartlett Learning, LLC
www.jblearning.com

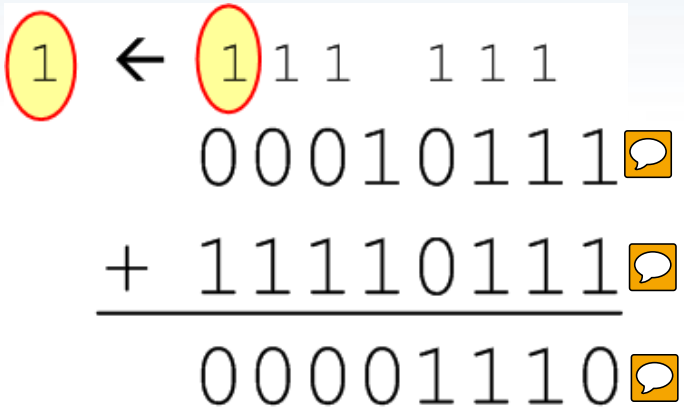
2.4 Signed Integer Representation

- When we use any finite number of bits to represent a number, we always run the risk of the result of our calculations becoming too large or too small to be stored in the computer.
- While we can't always prevent overflow, we can always *detect* overflow.
- In complement arithmetic, an overflow condition is easy to detect.

© 2012 Jones & Bartlett Learning, LLC
www.jblearning.com

2.4 Signed Integer Representation

- Example:
  - Using two's complement binary arithmetic, find the sum of 23 and -9.
  - We see that there is carry into the sign bit and carry out. The final result is correct: 23 + (-9) = 14.



Rule for detecting signed two's complement overflow: When the "carry in" and the "carry out" of the sign bit differ, overflow has occurred. If the carry into the sign bit equals the carry out of the sign bit, no overflow has occurred.

© 2012 Jones & Bartlett Learning, LLC
www.jblearning.com

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:

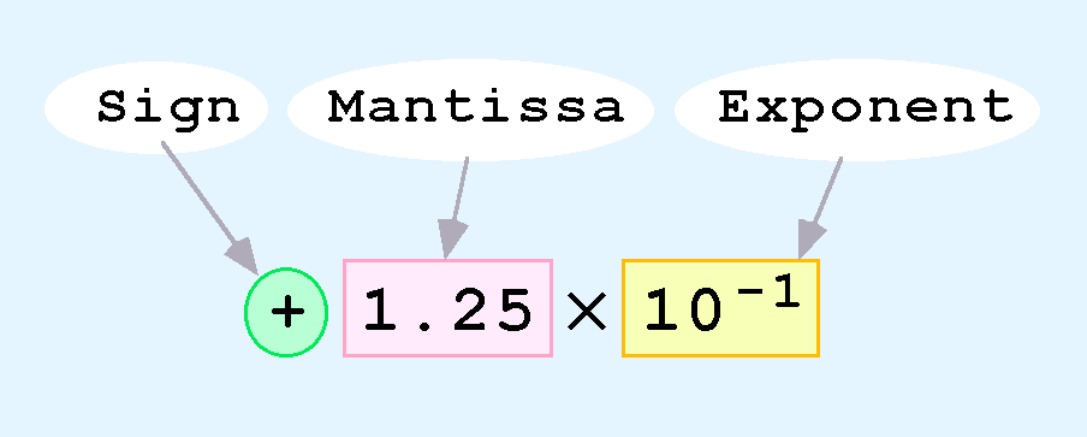
    $0.125 = 1.25 \times 10^{-1}$

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

© 2012 Jones & Bartlett Learning, LLC
www.jblearning.com

2.5 Floating-Point Representation

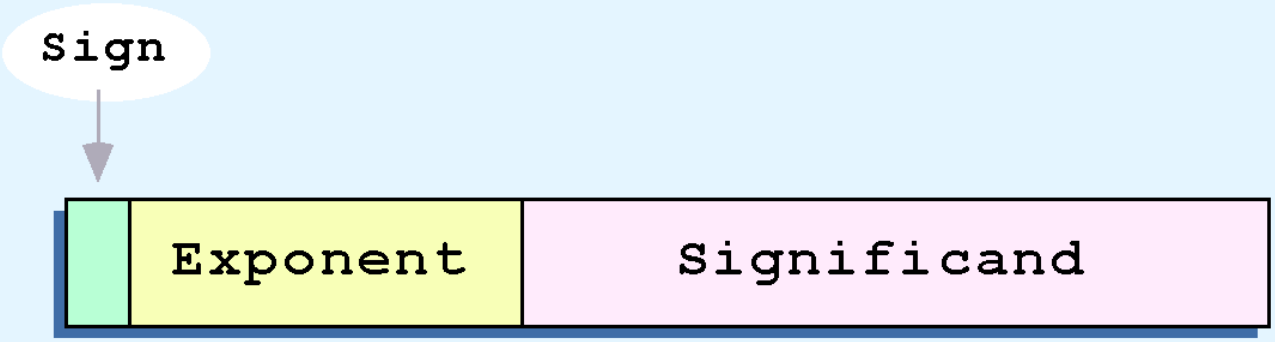
- Computers use a form of scientific notation for floating-point representation
- Numbers written in scientific notation have three components:



© 2012 Jones & Bartlett Learning, LLC
www.jblearning.com

2.5 Floating-Point Representation

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

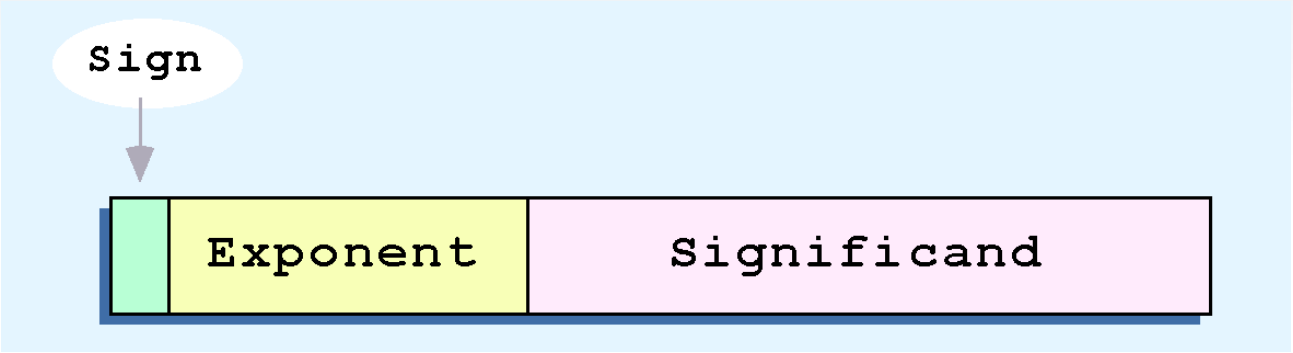


- This is the standard arrangement of these fields.

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.

© 2012 Jones & Bartlett Learning, LLC
www.jblearning.com

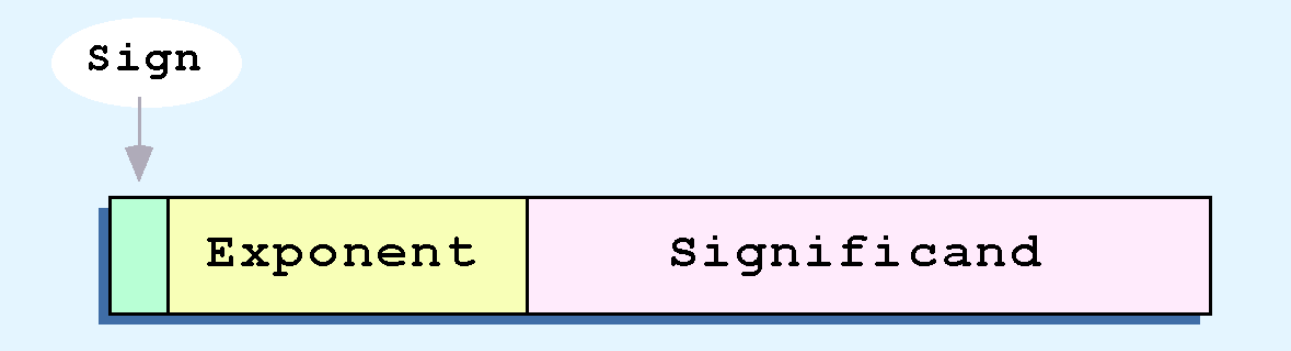
2.5 Floating-Point Representation



- 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.

© 2012 Jones & Bartlett Learning, LLC
www.jblearning.com

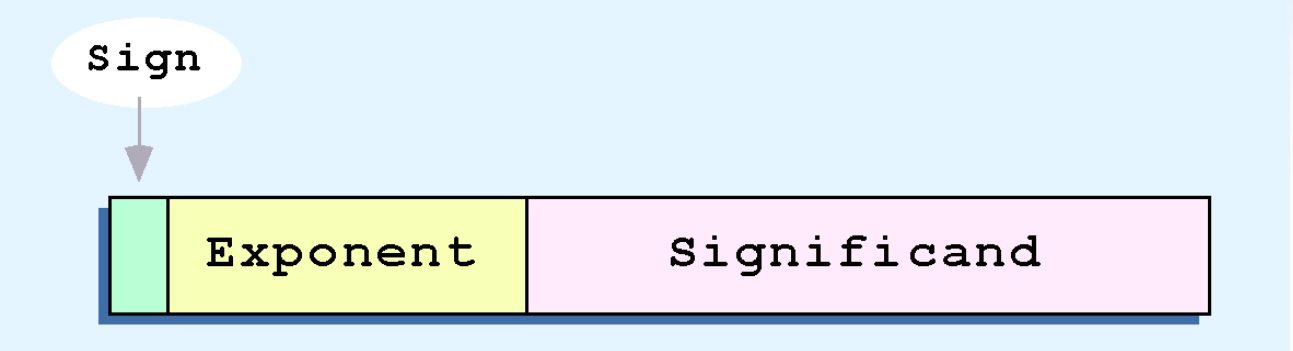
2.5 Floating-Point Representation



- 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

© 2012 Jones & Bartlett Learning, LLC
www.jblearning.com

2.5 Floating-Point Representation

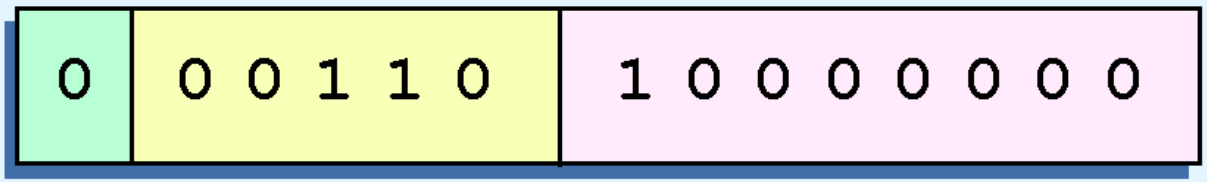


- The significand is always preceded by an implied binary point.
- Thus, the significand always contains a fractional binary value.
- The exponent indicates the power of 2 by which the significand is multiplied.

© 2012 Jones & Bartlett Learning, LLC
www.jblearning.com

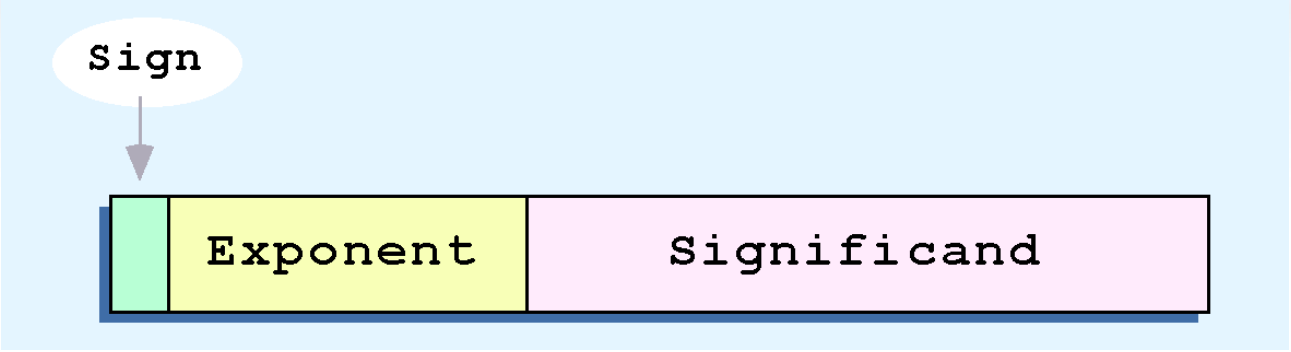
2.5 Floating-Point Representation

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



© 2012 Jones & Bartlett Learning, LLC
www.jblearning.com

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.

© 2012 Jones & Bartlett Learning, LLC
www.jblearning.com

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.

© 2012 Jones & Bartlett Learning, LLC
www.jblearning.com

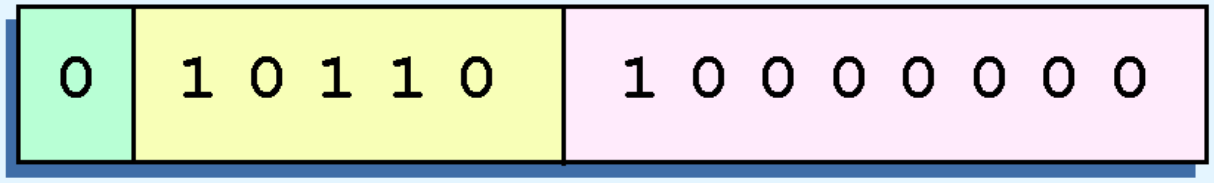
2.5 Floating-Point Representation

- 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.

© 2012 Jones & Bartlett Learning, LLC
www.jblearning.com

2.5 Floating-Point Representation

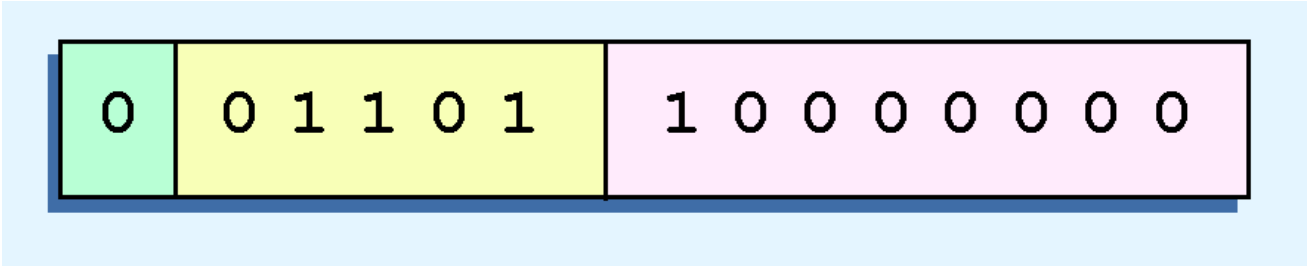
- Example:
  - Express 32_{10} in the revised 14-bit floating-point model.
- We know that 32 = 1.0 x 2^5 = 0.1 x 2^6.
- To use our excess 16 biased exponent, we add 16 to 6, giving 22_{10} (=10110_2).
- So we have:



© 2012 Jones & Bartlett Learning, LLC
www.jblearning.com

2.5 Floating-Point Representation

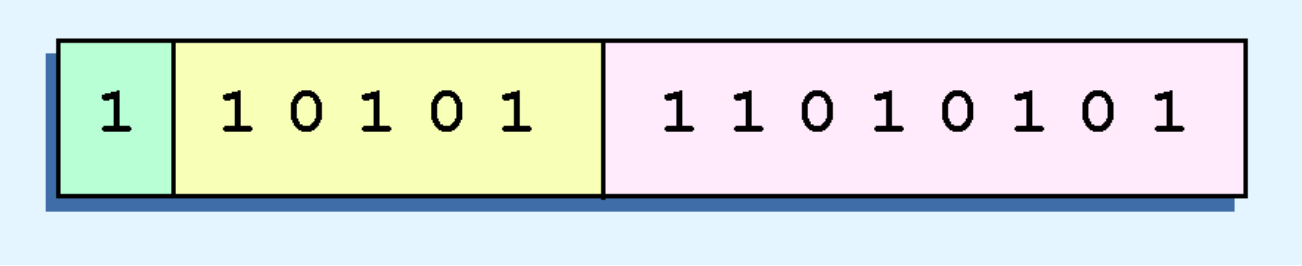
- Example:
  - Express 0.0625_{10} in the revised 14-bit floating-point model.
- We know that 0.0625 is 2^{-4}. So in (binary) scientific notation 0.0625 = 1.0 x 2^{-4} = 0.1 x 2^{-3}.
- To use our excess 16 biased exponent, we add 16 to -3, giving 13_{10} (=01101_2).



© 2012 Jones & Bartlett Learning, LLC
www.jblearning.com

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 16 biased exponent, we add 16 to 5, giving 21_{10} (=10101_2). We also need a 1 in the sign bit.



© 2012 Jones & Bartlett Learning, LLC
www.jblearning.com