CS-200

Computer Organization and Assembly Language

Data Representation in Computer Systems

Chapter 2 Objectives

- Understand the fundamentals of numerical data representation and manipulation in digital computers.
- Master the skill of converting between various important radix systems.
- Understand how errors can occur in computations because of overflow and truncation.

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

Chapter 2 Objectives

- Understand the fundamental concepts of floating-point representation.
- Gain familiarity with the most popular character codes.
- Understand the concepts of error detecting and correcting codes.

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

- A *bit* is the most basic unit of information in a computer.
  - It is a state of "on" or "off" in a digital circuit.
  - Sometimes these states are "high" or "low" voltage instead of "on" or "off.."
- A *byte* is a group of eight bits.
  - A byte is the smallest possible *addressable* unit of computer storage.
  - The term, "addressable," means that a particular byte can be retrieved according to its location in memory.

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

- A *word* is a contiguous group of bytes.
  - Words can be any number of bits or bytes.
  - Word sizes of 16, 32, or 64 bits are most common.
  - In a word-addressable system, a word is the smallest addressable unit of storage.
- A group of four bits is called a *nibble*.
  - Bytes, therefore, consist of two nibbles: a "high-order nibble," and a "low-order" nibble.
  - Nibble is sometimes spelled nybble.

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

- Bytes store numbers using the position of each bit to represent a power of 2.

1	0	1	1	0	0	0	1 = 177
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

$$128 + 0 + 32 + 16 + 0 + 0 + 0 + 1 = 177$$

  - The binary system is also called the base-2 system.
  - Our decimal system is the base-10 system. It uses powers of 10 for each position in a number.
  - Any integer quantity can be represented exactly using any base (or *radix*).

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

- The decimal number 947 in powers of 10 is:

$$9 \times 10^2 + 4 \times 10^1 + 7 \times 10^0$$

- The decimal number 5836.47 in powers of 10 is:

$$5 \times 10^3 + 8 \times 10^2 + 3 \times 10^1 + 6 \times 10^0 + 4 \times 10^{-1} + 7 \times 10^{-2}$$

- To avoid confusion, the base is often denoted by a subscript:

$$11001_2 = 25_{10}$$

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

Data Representation

CONVERTING BETWEEN BASES

Converting Between Bases

- **Suppose we want to convert the decimal number 8763 to base 8.**
  - We know that $8^5 = 32768$ so our result will be less than six digits wide. The largest power of 8 that we need is therefore $8^4 = 4096$ and $4096 \times 2 = 8192$.
  - Write down the 2 and subtract 8192 from 8763, giving 571.

$$\begin{array}{r} 8763 \\ \underline{-8192} \\ 571 \end{array} = 8^4 \times 2$$

Converting Between Bases

- **Converting 8763 to base 8…**
  - The next power of 8 is $8^3 = 512$. We'll need one of these, so we subtract 512 and write down 59 in our result.
  - The next power of 8, $8^2 = 64$, is too large, but we have to assign a placeholder of zero and carry down the 59.

$$
\begin{array}{rl}
8763 & \\
-8192 & = 8^4 \times 2 \\
\hline
571 & \\
-512 & = 8^3 \times 1 \\
\hline
59 & \\
-0 & = 8^2 \times 0 \\
\hline
59 &
\end{array}
$$

Converting Between Bases

- **Converting 8763 to base 8…**
  - $8^1 = 8$ goes 7 times, leaving us with 3.
  - The last power of 8, $8^0 = 1$, is our last choice, and times 3 gives us a difference of zero.
  - Our result, reading from top to bottom is:

$8763_{10} = 21073_8$

8763
-8192 = $8^4 \times 2$
571
-512 = $8^3 \times 1$
59
-0 = $8^2 \times 0$
59
-56 = $8^1 \times 7$
3
-3 = $8^0 \times 3$
0

Converting Between Bases

- **Another method uses division instead of subtraction.**
  - First we take the number that we wish to convert and divide it by the radix in which we want to express our result.
  - In this case, 8 divides 8763 1095 times, with a remainder of 3.
  - Record the quotient and the remainder

$$8 \,|\, \underline{8763} \quad 3$$
$$1095$$

Converting Between Bases

- **Converting 8763 to base 8…**
  - Divide 1095 by 8.
  - Our remainder is 7, and the quotient is 136.

$$
\begin{array}{r|l|r}
8 & \underline{8763} & 3 \\
8 & \underline{1095} & 7 \\
 & 136 &
\end{array}
$$

Converting Between Bases

- **Converting 8763 to base 8…**
  - Continue in this way until the quotient is zero.
  - In the final calculation, we note that 8 divides 2 zero times with a remainder of 2.
  - Our result, reading from bottom to top is:

$$8763_{10} = 21073_8$$

8	8763	3
8	1095	7
8	136	0
8	17	1
8	2	2
	0	

Converting Between Bases

- Fractional values can be approximated in all base systems.
- Unlike integer values, fractions do not necessarily have exact representations under all radices.
- The quantity ½ is exactly representable in the binary and decimal systems, but is not in the ternary (base 3) numbering system.

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

Converting Between Bases

- Fractional decimal values have nonzero digits to the right of the decimal point.
- Fractional values of other radix systems have nonzero digits to the right of the *radix point*.
- Numerals to the right of a radix point represent negative powers of the radix:

$$0.47_{10} = 4 \times 10^{-1} + 7 \times 10^{-2}$$

$$0.11_2 = 1 \times 2^{-1} + 1 \times 2^{-2}$$

$$= \frac{1}{2} + \frac{1}{4}$$

$$= 0.5 + 0.25 = 0.75$$

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

Converting Between Bases

- As with whole-number conversions, you can use either of two methods: a subtraction method or an easy multiplication method.
- The subtraction method for fractions is identical to the subtraction method for whole numbers. Instead of subtracting positive powers of the target radix, we subtract negative powers of the radix.
- We always start with the largest value first, n^{-1}, where n is our radix, and work our way along using larger negative exponents.

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

Converting Between Bases

- **The calculation to the right is an example of using the subtraction method to convert the decimal 0.8125 to binary.**
  - Our result, reading from top to bottom is:

$$0.8125_{10} = 0.1101_2$$

  - Of course, this method works with any base, not just binary.

$$\begin{array}{rll} 0.8125 & & \\ -\ 0.5000 & = 2^{-1} \times & 1 \\ \hline 0.3125 & & \\ -\ 0.2500 & = 2^{-2} \times & 1 \\ \hline 0.0625 & & \\ -\ 0 & = 2^{-3} \times & 0 \\ \hline 0.0625 & & \\ -\ 0.0625 & = 2^{-4} \times & 1 \\ \hline 0 & & \end{array}$$

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

Converting Between Bases

- **Using the multiplication method to convert the decimal 0.8125 to binary, we multiply by the radix 2.**
  - The first product carries into the units place.

```
  .8125
×    2
------
1.6250
```

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

Converting Between Bases

- **Converting 0.8125 to binary . . .**
  - Ignoring the value in the units place at each step, continue multiplying each fractional part by the radix.

```
  .8125
   ×   2
------
1.6250

  .6250
   ×   2
------
1.2500

  .2500
   ×   2
------
0.5000
```

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

Converting Between Bases

- **Converting 0.8125 to binary . . .**
  - You are finished when the product is zero, or until you have reached the desired number of binary places.
  - Our result, reading from top to bottom is:

    $0.8125_{10} = 0.1101_2$
  - This method also works with any base. Just use the target radix as the multiplier.

```
  .8125
 ×   2
------
1.6250

  .6250
 ×   2
------
1.2500

  .2500
 ×   2
------
0.5000

  .5000
 ×   2
------
1.0000
```

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

Converting Between Bases

- The binary numbering system is the most important radix system for digital computers.
- However, it is difficult to read long strings of binary numbers -- and even a modestly-sized decimal number becomes a very long binary number.
  - For example: $1101010001101 1_2 = 13595_{10}$
- For compactness and ease of reading, binary values are usually expressed using the hexadecimal, or base-16, numbering system.

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

Converting Between Bases

- The hexadecimal numbering system uses the numerals 0 through 9 and the letters A through F.
  - The decimal number 12 is C_{16}.
  - The decimal number 26 is $1A_{16}$.
- It is easy to convert between base 16 and base 2, because $16 = 2^4$.
- Thus, to convert from binary to hexadecimal, all we need to do is group the binary digits into groups of four.

A group of four binary digits is called a hextet

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

Converting Between Bases

- Using groups of hextets, the binary number $1101010001101 1_2$ ($= 13595_{10}$) in hexadecimal is:

0011	0101	0001	1011
3	5	1	B

If the number of bits is not a multiple of 4, pad on the left with zeros.

- Octal (base 8) values are derived from binary by using groups of three bits ($8 = 2^3$):

011	010	100	011	011
3	2	4	3	3

Octal was very useful when computers used six-bit words.

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

Converting Between Bases

- To convert back to binary from hexidecimal or octal, simply expand each digit to its equivalent binary and then string the binary together

  $3B7A_{16}$: 3 = 0011, B = 1011, 7 = 0111, and A = 1010.
  That gives us 0011101101111010_2

- The leading 0s are often dropped unless you wish to represent all the bits in a particular memory or register location.
- The easiest way to convert from octal to hex or vice-versa is to first convert to binary.

Data Representation

BINARY MATHEMATICS

Binary Addition

- 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.
- Everything is fine as long as our sums remain less than the space we have available
  - For unsigned 8-bit math, that would be 2^9 or 256
  - For signed 8-bit math, that would be $+/-2^8$ or 128

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

Binary Addition

- Actually, digital computers ‘know’ memory as high or low voltage, not 0s or 1s, so they don’t really do math like adding or subtracting.
- Instead, they use boolean algebra to simulate math.
- For simple 1-bit adder you have 2 inputs and two outputs, the result and a carry bit.
- The outputs are derived using truth tables:

		RESULT
0	0	0
0	1	1
1	0	1
1	1	0

		CARRY
0	0	0
0	1	0
1	0	0
1	1	1

Binary Addition

- Actually, digital computers ‘know’ memory as high or low voltage, not 0s or 1s, so they don’t really do math like adding or subtracting.
- Instead, they use boolean algebra to simulate math.
- For simple 1-bit adder you have 2 inputs and two outputs, the result and a carry bit.
- The outputs are derived using truth tables:

XOR		RESULT
0	0	0
0	1	1
1	0	1
1	1	0

AND		CARRY
0	0	0
0	1	0
1	0	0
1	1	1

Binary Addition

- So our simple adder (also known as a *half-adder*) consists of an XOR circuit and an AND circuit.
- If we want more bits, the ones to the left will have an extra input: the carry from the bit to the right. But it's still the same idea: do the truth tables to get the circuits for carry and result. We call these *full-adders*.
- An adder for X bits simply strings together X of these full-adders by sending the carry from a bit to the next leftmost adder.
  - The rightmost adder always gets a 0 carry
  - The leftmost adder outputs the carry for the entire set
- The diagrams for these circuits are in Chapter 3.

Binary Multiplication

- The rules for binary multiplication are even easier:
  - Anything times 0 = 0
  - Anything times 1 = itself
- For example: 19 X 21 = 399
- Of course, each time you have to add and check for overflow
- You can create a circuit that strings together multi-bit adders that get the contents of the multiplications but it's a little complicated

$$
\begin{array}{r}
10011 \\
\times\ 10101 \\
\hline
10011 \\
00000 \\
10011 \\
00000 \\
10011 \\
\hline
110001111
\end{array}
$$

Binary Mathematics

- Subtraction and division can be constructed using logic circuits similarly.
- Boundary conditions add the most complications.
- Much research has gone into finding better algorithms for binary math and is still ongoing.
- Booth's algorithm is one such method that works for binary multiplication.
- Take away: computers do math differently than we do. The way we represent numbers might have a beneficial effect on the way we perform binary math.

Data Representation

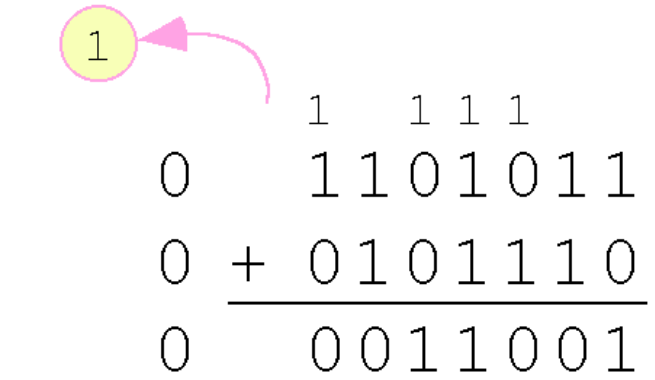
SIGNED INTEGERS

Signed Integers

- To represent a signed integer, we use the high-order bit to indicate the sign.
  - The high-order bit is the leftmost bit
  - 0 means positive, 1 means negative
- The simplest scheme has the remaining bits signifying the value of the number.
  - This is called signed magnitude
  - However, this means that there are 2 zeros: +0 (00000000_2) and -0 (10000000_2)

Signed Integers

- With unsigned addition, large sums leave us with extra digits which trigger an overflow error. Signed addition *can* also work that way.
- 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 and Bartlett Learning, LLC
www.jblearning.com

Signed Integers

- 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.

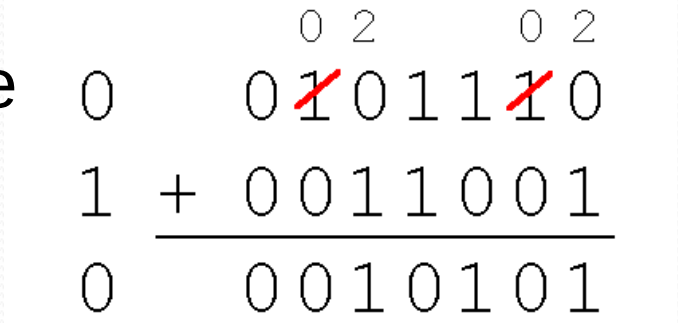
```
    1 1
1   0101110
1 + 0011001
-----------
1   1000111
```

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

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

Signed Integers

- 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 and Bartlett Learning, LLC
www.jblearning.com

Signed Integers

- Because of the special way we have to treat the sign bits for signed magnitude integers, it is awkward and costly to implement in digital logic.
- Let's not forget the two values for 0, too, which can confuse things.
- And recall the difficulties creating logical circuits to do binary math.
- Perhaps there's a better way?
- Computers often employ a *complement* system to represent signed integers.

Signed Integers

- In complement systems, negative values are represented by some difference between a number and its base.
- The *diminished radix complement* of a non-zero number *N* in base *r* with *d* digits is $(r^d - 1) - N$
- In the binary system, this gives us *one's complement*. It amounts to little more than flipping the bits of a binary number.

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

Signed Integers

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

  + 3 is: `00000011`

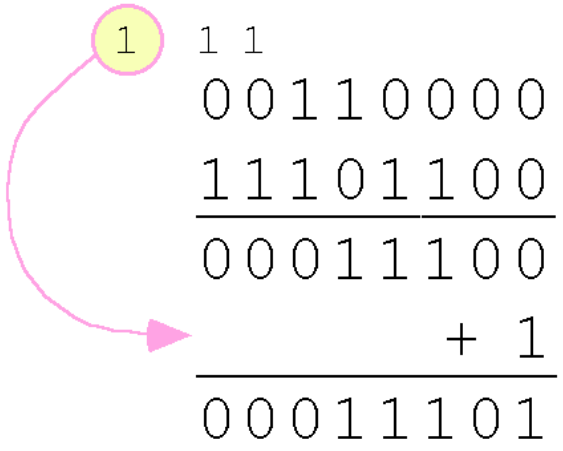
  - 3 is: `11111100`

- 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 and Bartlett Learning, LLC
www.jblearning.com

Signed Integers

- 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



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

Signed Integers

- Although the “end carry around” adds some complexity, one’s complement is simpler to implement than signed magnitude.
- But it still has the disadvantage of having two different representations for zero: positive zero and negative zero.
- Two’s complement solves this problem.
- Two’s complement is the radix complement of the binary numbering system; the *radix complement* of a non-zero number N in base r with d digits is $r^d - N$.

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

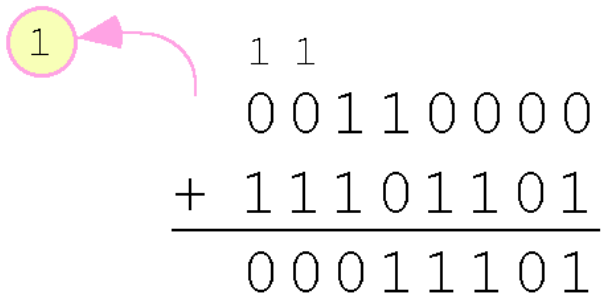
Signed Integers

- 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 and Bartlett Learning, LLC
www.jblearning.com

Signed Integers

- With two's complement arithmetic, all we do is add our two binary numbers. Just discard any carries emitting from the high order bit.
  - Example: Using one's complement binary arithmetic, find the sum of 48 and - 19.



Signed Integer Representation

- Overflow and carry are tricky ideas.
- Signed number overflow means nothing in the context of unsigned numbers, which set a carry flag instead of an overflow flag.
- If a carry out of the leftmost bit occurs with an unsigned number, overflow has occurred.
- Carry and overflow occur independently of each other.

The table on the next slide summarizes these ideas.

Signed Integer Representation

Expression	Result	Carry?	Overflow?	Correct Result?
0100 + 0010	0110	No	No	Yes
0100 + 0110	1010	No	Yes	No
1100 + 1110	1010	Yes	No	Yes
1100 + 1010	0110	Yes	Yes	No

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

Signed Integer Representation

- We can do binary multiplication and division by 2 very easily using an *arithmetic shift* operation
- A *left arithmetic shift* inserts a 0 in for the rightmost bit and shifts everything else left one bit; in effect, it multiplies by 2
- A *right arithmetic shift* shifts everything one bit to the right, but copies the sign bit; it divides by 2
- Let's look at some examples.

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

Signed Integer Representation

Example:

Multiply the value 11 (expressed using 8-bit signed two's complement representation) by 2.

We start with the binary value for 11:

00001011 (+11)

We shift left one place, resulting in:

00010110 (+22)

The sign bit has not changed, so the value is valid.

To multiply 11 by 4, we simply perform a left shift twice.

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

Signed Integer Representation

Example:

Divide the value 12 (expressed using 8-bit signed two's complement representation) by 2.

We start with the binary value for 12:

00001100 (+12)

We shift left one place, resulting in:

00000110 (+6)

(Remember, we carry the sign bit to the left as we shift.)

To divide 12 by 4, we right shift twice.

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

Signed Integer Representation

- Shifting is easier and faster to implement in logic circuits than implementing the algorithm we discussed back on slide 29.
- However, shifting only works in powers of 2.
- Booth's algorithm combines shifting with addition and subtraction where necessary to bring the speed of shifting to general multiplication.

Signed Integer Representation

In Booth's algorithm:

- If the current multiplier bit is 1 and the preceding bit was 0, subtract the multiplicand from the product
- If the current multiplier bit is 0 and the preceding bit was 1, we add the multiplicand to the product
- If we have a 00 or 11 pair, we simply shift.
- Assume a mythical "0" starting bit
- Shift after each step

```
          0011
       x 01100  <- mythical "0"
       + 0000    (shift)
       - 0011    (subtract)
      + 0000     (shift)
     + 0011      (add)
      00010010
```

We see that 3 × 6 = 18!

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

Signed Integer Representation

- Here is a larger example.

Subtract (add negative).

Ignore all bits over 2n.

```
                00110101
      x         01111110
+ 0000000000000000
+ 1111111111001011
+ 0000000000000
+ 000000000000
+ 00000000000
+ 0000000000
+ 000000000
+ 000110101
 1000110100010110
```

53 × 126 = 6678!

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

Data Representation

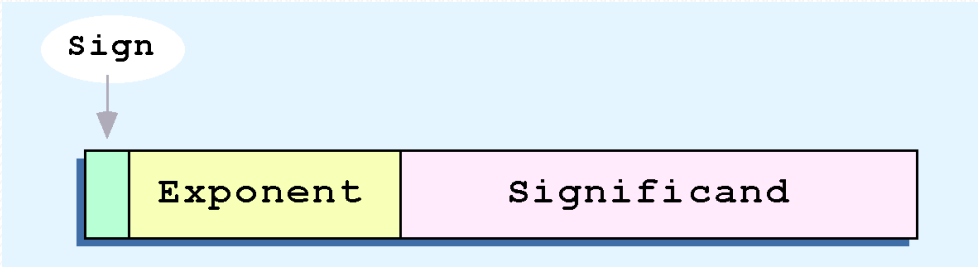
FLOATING POINT REPRESENTATION

Floating Point Representation

- The operations we've discussed work for integer representations, but imagine having to keep track of a radix point at the same time.
- Instead, we will use a representation of floating point numbers and leverage some useful mathematical properties.
- This is known as floating point emulation because the values aren't stored as such (no radix point in binary storage).
- Most computers today have hardware that manages this floating point emulation in hardware, speeding it up immensely.

Floating Point Representation

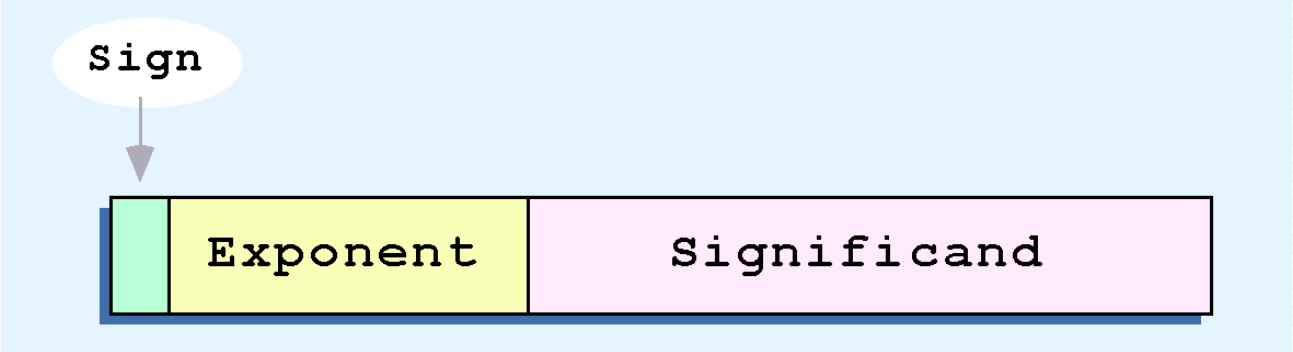
- Floating point numbers may be expressed in Scientific Notation: $0.125 = 1.25 \times 10^{-1}$
- This notation consists of three parts: sign, mantissa, and exponent.
- Computer representation of a floating point number consists of three fixed-size fields:



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 and Bartlett Learning, LLC
www.jblearning.com

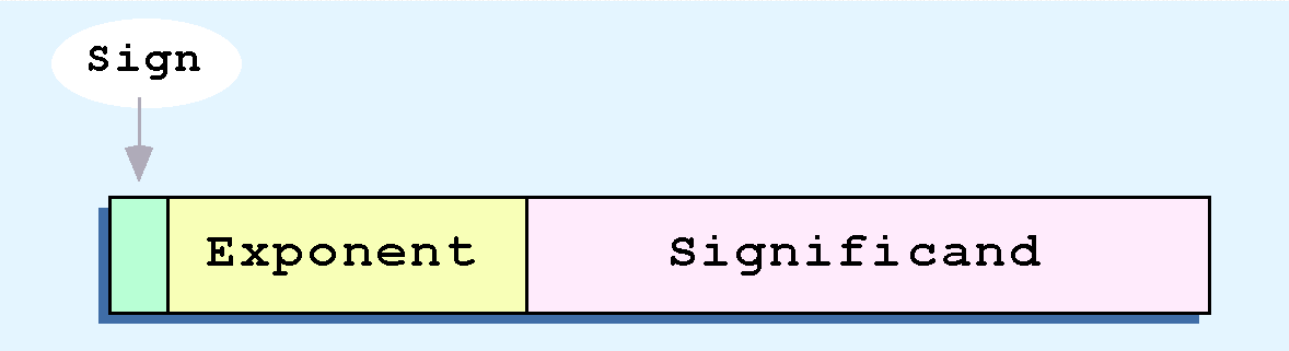
Floating Point Representation



- 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 and Bartlett Learning, LLC
www.jblearning.com

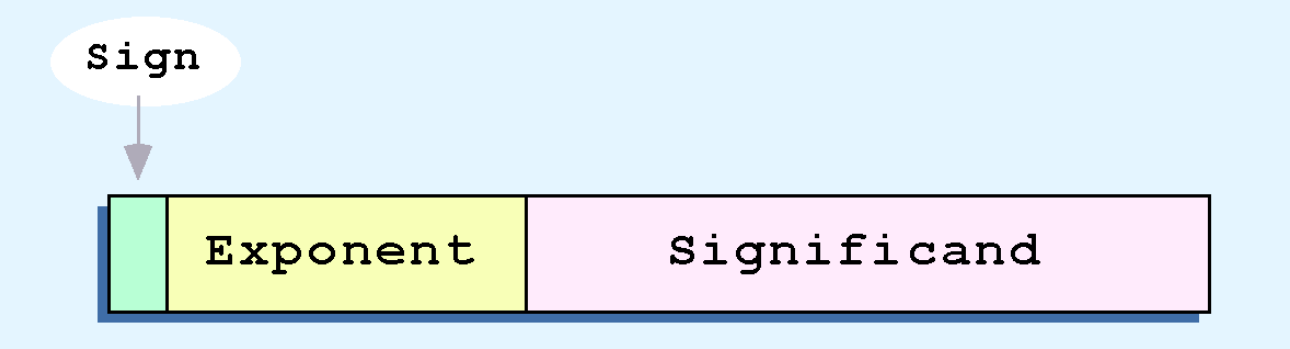
Floating Point Representation



- 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 and Bartlett Learning, LLC
www.jblearning.com

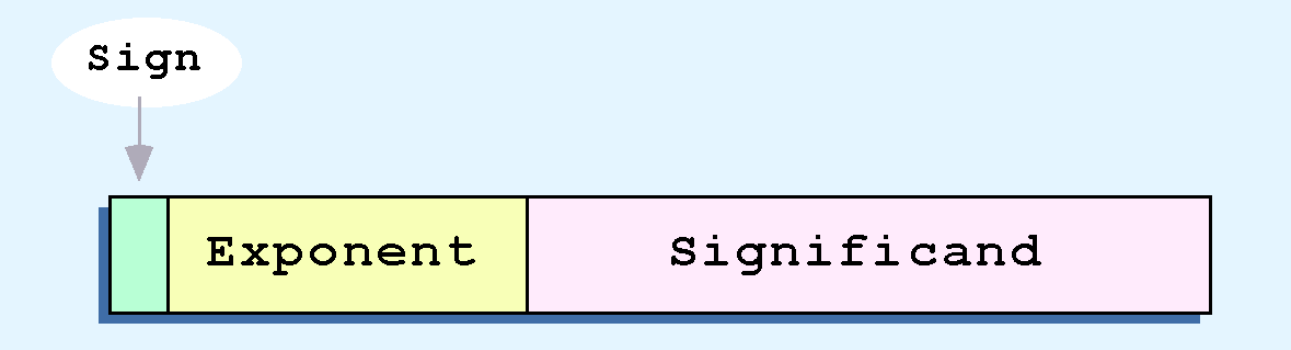
Floating Point Representation



- 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 and Bartlett Learning, LLC
www.jblearning.com

Floating Point Representation

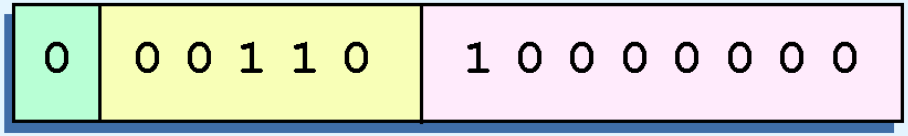


- The significand is always preceded by an implied binary point.
- Thus, the significand always contains a fractional binary value.
- The exponent indicates the power of 2 by which the significand is multiplied.

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

Floating Point Representation

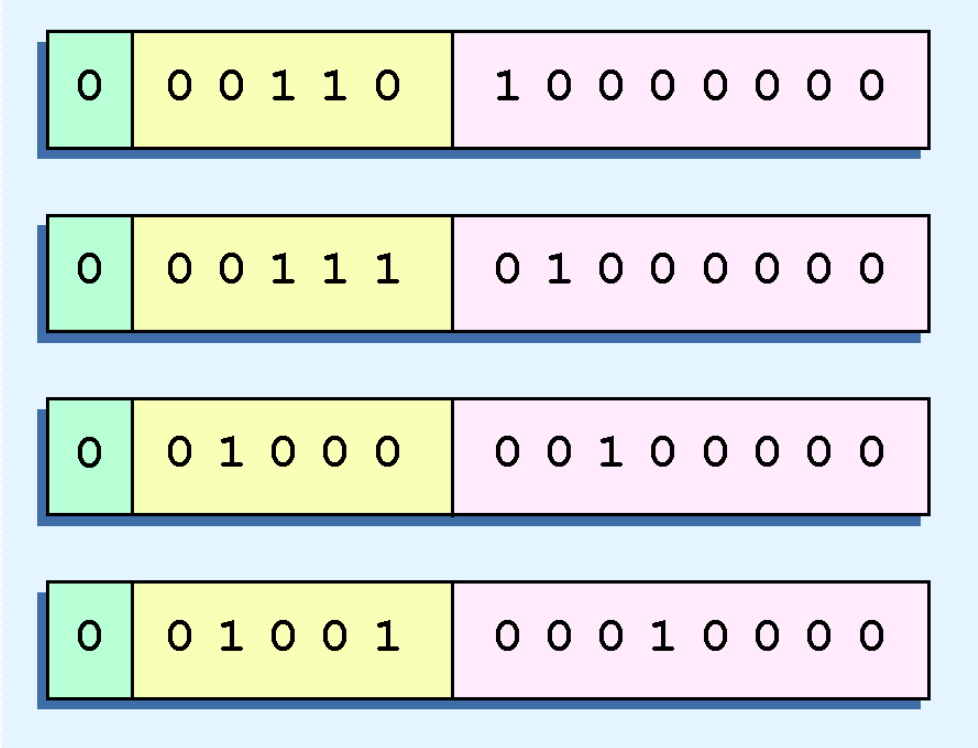
- Example:
  - Express 32_{10} in the simplified 14-bit floating-point model.
- We know that 32 is 2^5. So in (binary) scientific notation $32 = 1.0 \times 2^5 = 0.1 \times 2^6$.
  - We're using the second version arbitrarily to illustrate a point on the next slide.
- Using this information, we put 110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



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

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.
- Obviously, we need a standard.



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

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.

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

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.)
- Don't be confused by the sign bit for the entire number.

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

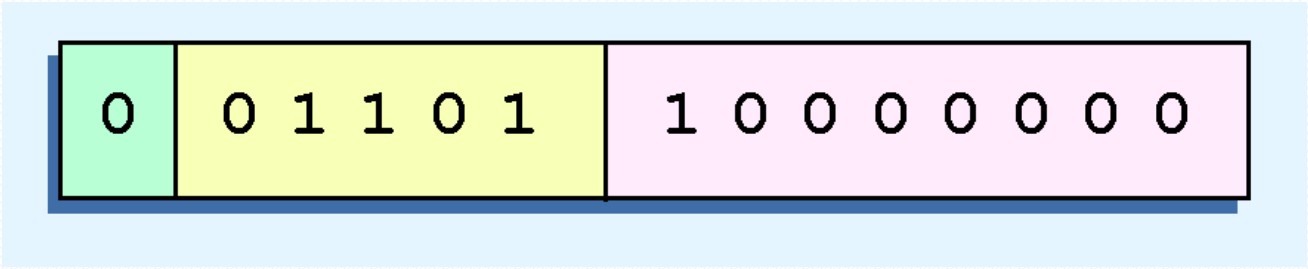
Floating Point Representation

- 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 and Bartlett Learning, LLC
www.jblearning.com

Floating Point Representation

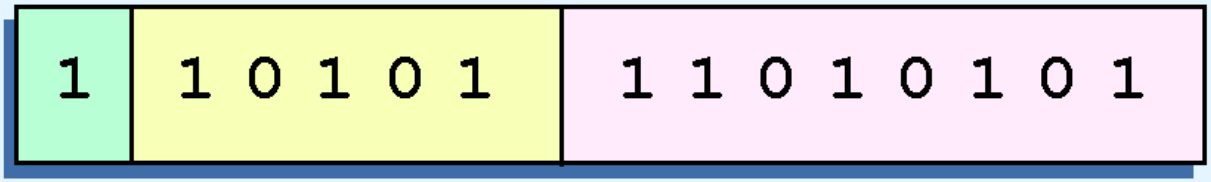
- Example:
  - Express 0.0625_{10} in the revised 14-bit floating-point model.
- We know that 0.0625 is 2^{-4}. So in (binary) scientific notation $0.0625 = 1.0 \times 2^{-4} = 0.1 \times 2^{-3}$.
- To use our excess 16 biased exponent, we add 16 to -3, giving 13_{10} ($=01101_2$).



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

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 and Bartlett Learning, LLC
www.jblearning.com

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.

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

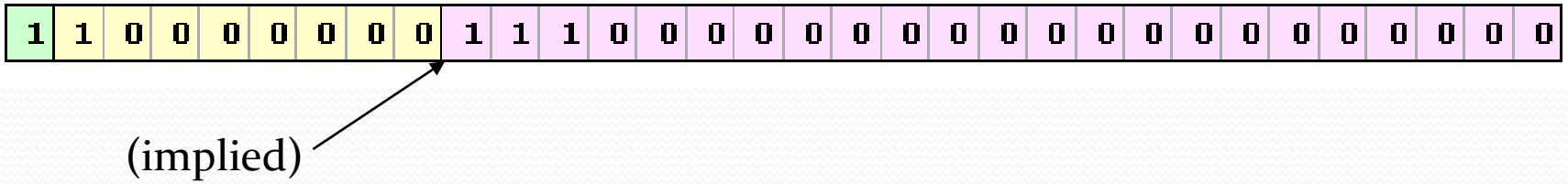
Floating Point Representation

- 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 is does not need to be listed in the significand (the significand would include only 001).

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

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)



  - Since we have an implied 1 in the significand, this equates to

    $-(1).111_2 \times 2^{(128-127)} = -1.111_2 \times 2^1 = -11.11_2 = -3.75.$

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

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 ± 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.

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

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.

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

Data Representation

FLOATING POINT REPRESENTATION

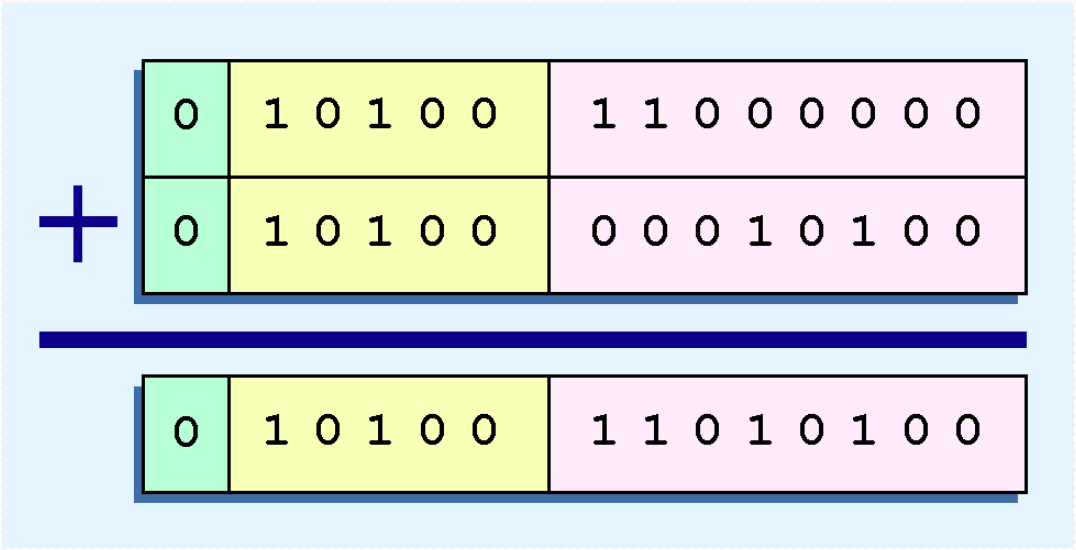
Floating Point Calculation

- 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.

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

Floating Point Calculation

- 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 x 2 4.



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

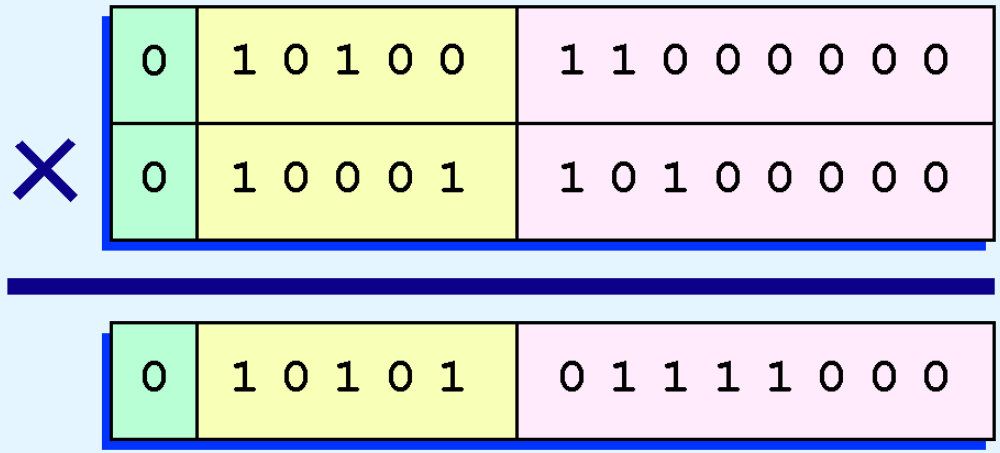
Floating Point Calculation

- 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.

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

Floating Point Calculation

- Example:
  - Find the product of 12_{10} and 1.25_{10} using the 14-bit floating-point model.
- We find $12_{10} = 0.1100 \times 2^{4}$. 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 $22_{10} = 10110_2$.



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

Floating Point Calculation

- 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.

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

Floating Point Calculation

- 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:

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

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

Floating Point Calculation

- 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.

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

Floating Point Calculation

- 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.

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

Floating Point Calculation

- 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.

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

Floating Point Calculation

- 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.

Pat's note: really experienced programmers know it's better not to crash, either.

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

Floating Point Calculation

- 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

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

Floating Point Calculation

- 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.

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

Data Representation

CHARACTER CODES

Character Codes

- If computer's don't really store numbers, how can we tell devices what we want printed or displayed?
- Beyond numbers, how about the alphabet or other characters?
- Basically, we need a code.
  - Computer 'knows' bit patterns
  - We must match patterns to characters in I/O context.

Character Codes

- As computers have evolved, character codes have evolved.
- Larger computer memories and storage devices permit richer character codes.
- The earliest computer coding systems used six bits.
- Binary-coded decimal (BCD) was one of these early codes. It was used by IBM mainframes in the 1950s and 1960s.

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

Character Codes

- In 1964, BCD was extended to an 8-bit code, Extended Binary-Coded Decimal Interchange Code (EBCDIC).
- EBCDIC was one of the first widely-used computer codes that supported upper *and* lowercase alphabetic characters, in addition to special characters, such as punctuation and control characters.
- EBCDIC and BCD are still in use by IBM mainframes today.

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

Character Codes

- Other computer manufacturers chose the 7-bit ASCII (American Standard Code for Information Interchange) as a replacement for 6-bit codes.
- While BCD and EBCDIC were based upon punched card codes, ASCII was based upon telecommunications (Telex) codes.
- Until recently, ASCII was the dominant character code outside the IBM mainframe world.

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

Character Codes

- Many of today's systems embrace Unicode, a 16-bit system that can encode the characters of every language in the world.
  - The Java programming language, and some operating systems now use Unicode as their default character code.
- The Unicode codespace is divided into six parts. The first part is for Western alphabet codes, including English, Greek, and Russian.

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

Character Codes

- The Unicode codes-pace allocation is shown at the right.
- The lowest-numbered Unicode characters comprise the ASCII code.
- The highest provide for user-defined codes.

Character Types	Language	Number of Characters	Hexadecimal Values
Alphabets	Latin, Greek, Cyrillic, etc.	8192	0000 to 1FFF
Symbols	Dingbats, Mathematical, etc.	4096	2000 to 2FFF
CJK	Chinese, Japanese, and Korean phonetic symbols and punctuation.	4096	3000 to 3FFF
Han	Unified Chinese, Japanese, and Korean	40,960	4000 to DFFF
	Han Expansion	4096	E000 to EFFF
User Defined		4095	F000 to FFFE

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

Data Representation

ERROR DETECTION AND CORRECTION

Error Detection and Correction

- It is physically impossible for any data recording or transmission medium to be 100% perfect 100% of the time over its entire expected useful life.
- As more bits are packed onto a square centimeter of disk storage, as communications transmission speeds increase, the likelihood of error increases--sometimes geometrically.
- Thus, error detection and correction is critical to accurate data transmission, storage and retrieval.

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

Error Detection and Correction

- Checksums and CRCs are examples of *systematic error detection*.
- In *systematic error detection* a group of error control bits is appended to the end of the block of transmitted data.
  - This group of bits is called a *syndrome*.
- CRCs are polynomials over the modulo 2 arithmetic field.

The mathematical theory behind modulo 2 polynomials is beyond our scope. However, we can easily work with it without knowing its theoretical underpinnings.

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

Error Detection and Correction

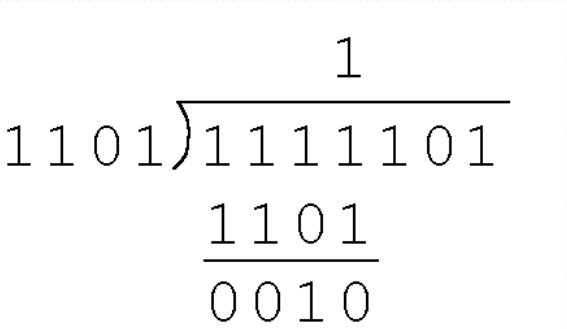
- **Modulo 2 subtraction is easy.**
- Ignore carry or borrow.
  - 0 - 0 = 0
  - 1 – 1 = 0
  - 1 – 0 = 1
  - 0 – 1 = 1
- So

$$\begin{array}{r} 1010 \\ -1100 \\ \hline 0110 \end{array}$$

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

Error Detection and Correction

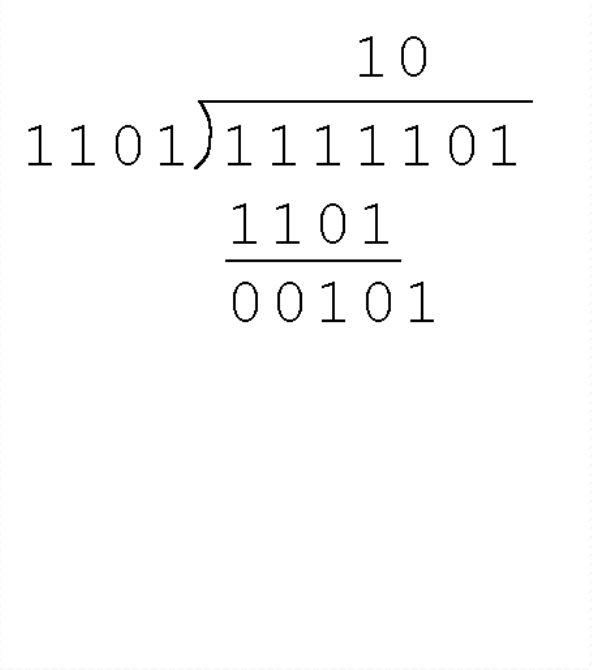
- **Find the quotient and remainder when 1111101 is divided by 1101 in modulo 2 arithmetic.**
  - As with traditional division, we note that the dividend is divisible once by the divisor.
  - We place the divisor under the dividend and perform modulo 2 subtraction.



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

Error Detection and Correction

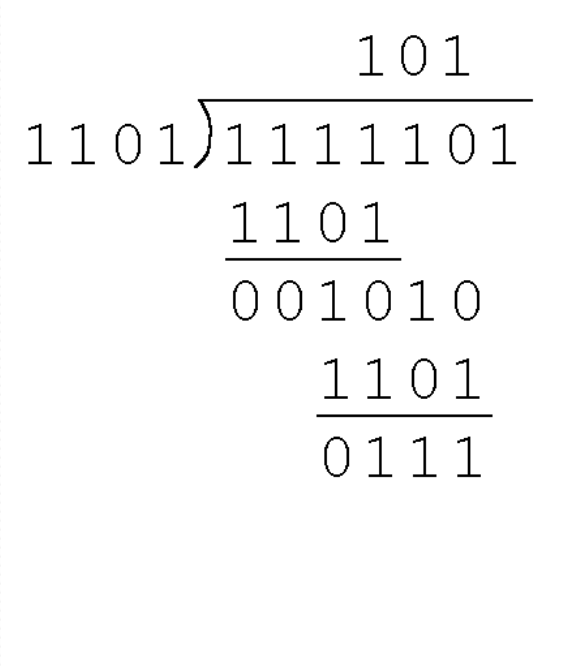
- **Find the quotient and remainder when 1111101 is divided by 1101 in modulo 2 arithmetic…**
  - Now we bring down the next bit of the dividend.
  - We see that 00101 is not divisible by 1101. So we place a zero in the quotient.



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

Error Detection and Correction

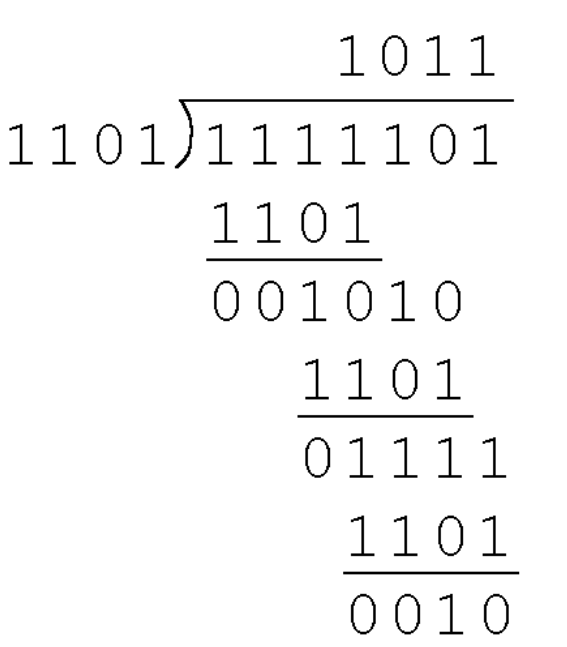
- **Find the quotient and remainder when 1111101 is divided by 1101 in modulo 2 arithmetic…**
  - 1010 is divisible by 1101 in modulo 2.
  - We perform the modulo 2 subtraction.



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

Error Detection and Correction

- **Find the quotient and remainder when 1111101 is divided by 1101 in modulo 2 arithmetic…**
  - We find the quotient is 1011, and the remainder is 0010.
- **This procedure is very useful to us in calculating CRC syndromes.**



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

Error Detection and Correction

- **Suppose we want to transmit the information string: 1111101.**
- **The receiver and sender decide to use the (arbitrary) polynomial pattern, 1101.**
- **The information string is shifted left by one position less than the number of positions in the divisor.**
- **The remainder is found through modulo 2 division (at right) and added to the information string: 1111101000 + 111 = 1111101111.**

```
          1011011
    ------------
1101)1111101000
     1101
     ----
     001010
        1101
        ----
        01111
         1101
         ----
         001000
            1101
            ----
            01010
             1101
             ----
             0111
```

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

Error Detection and Correction

- **If no bits are lost or corrupted, dividing the received information string by the agreed upon pattern will give a remainder of zero.**
- **We see this is so in the calculation at the right.**
- **Real applications use longer polynomials to cover larger information strings.**
- **If remainder = 0, receiver truncates to get original msg.**

```
          1011011
     ┌──────────
1101 ) 1111101111
       1101
       ────
       001010
          1101
          ────
          01111
            1101
            ────
            001011
               1101
               ────
               01101
                 1101
                 ────
                 0000
```

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

Error Detection and Correction

- Data transmission errors are easy to fix once an error is detected.
  - Just ask the sender to transmit the data again.
- In computer memory and data storage, however, this cannot be done.
  - Too often the only copy of something important is in memory or on disk.
- Thus, to provide data integrity over the long term, error *correcting* codes are required.

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

Error Detection and Correction

- Hamming codes and Reed-Solomon codes are two important error correcting codes.
- Reed-Solomon codes are particularly useful in correcting *burst errors* that occur when a series of adjacent bits are damaged.
  - Because CD-ROMs are easily scratched, they employ a type of Reed-Solomon error correction.
- Because the mathematics of Hamming codes is much simpler than Reed-Solomon, we discuss Hamming codes in detail.

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

Error Detection and Correction

- Let's suppose we have 2-bit data and we add a parity bit to make up 3-bit code words: 00 + 1 = 001, 01 + 0 = 010, 10 + 0 = 100, and 11 + 1 = 111.
- We only have 4 code words but there are 8 possible combinations of 3 bits: 000, 001, 010, 011, 100, 101, 110, and 111 (our code words are in red).
- The Hamming Distance between words is how many bits must be changed to get from one word to another: 001 to 010 takes two bit changes so the distance is two.

Error Detection and Correction

000, 001, 010, 011, 100, 101, 110, and 111

- The minimum Hamming Distance D(min) is the smallest distance between any two code words (the non-used codes don't count).
- In this example, D(min) = 2
- Two single-bit errors in a code word in our example could give us another code word but one error will always give us a non-used word, so we can always detect a single error.
- More generally, to detect *k* errors, D(min) = *k* + 1

Error Detection and Correction

000, 001, 010, 011, 100, 101, 110, and 111

- Hamming Codes have *n*-bit code words which have *m* data bits and *r* (redundant) parity bits. Our example is 3-bit words with 2 data bits and 1 parity bit.
- To detect and correct errors, each code word must have n invalid words at a Hamming Distance of 1.

  001 is associated with 000, 011, and 101.
- However, correction is only possible if the invalid words aren't shared: 011 –> 001 or 010?

Error Detection and Correction

- Hamming codes can *detect* D(min) - 1 errors and *correct* $\left\lfloor \frac{D(min) - 1}{2} \right\rfloor$ errors
- Thus, a Hamming distance of $2k + 1$ is required to be able to correct k errors in any data word.
- Hamming distance is provided by adding a suitable number of parity bits to a data word.

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

Error Detection and Correction

- Using n bits, we have 2^n possible bit patterns. We have 2^m valid code words with r check bits (where $n = m + r$).
- For each valid codeword, we have (n+1) bit patterns (1 legal and N illegal).
- This gives us the inequality:

$$(n + 1) \times 2^m \leq 2^n$$

- Because $n = m + r$, we can rewrite the inequality as:

$$(m + r + 1) \times 2^m \leq 2^{m+r} \quad \text{or} \quad (m + r + 1) \leq 2^r$$

  - This inequality gives us a lower limit on the number of check bits that we need in our code words.

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

Error Detection and Correction

- Suppose we have data words of length $m = 8$. Then:

  $$(8 + r + 1) \leq 2^r$$

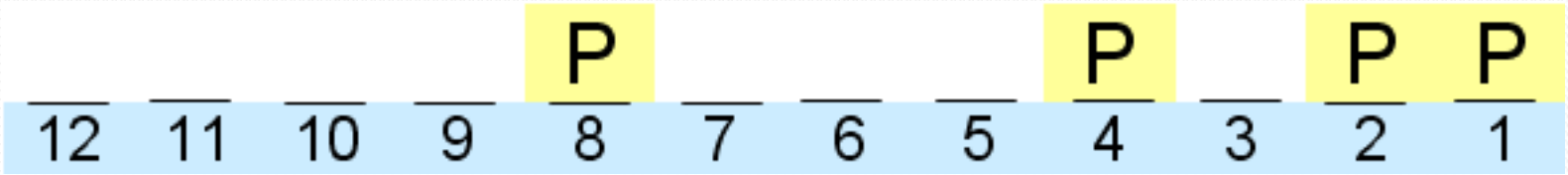
  implies that r must be greater than or equal to 4.

  - *We should always use the smallest whole value of r that makes the inequality true.*
- This means to build a code with 8-bit data words that will correct single-bit errors, we must add 4 check bits, creating code words of length 12.
- Finding the number of check bits is the hard part. The rest is easy.

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

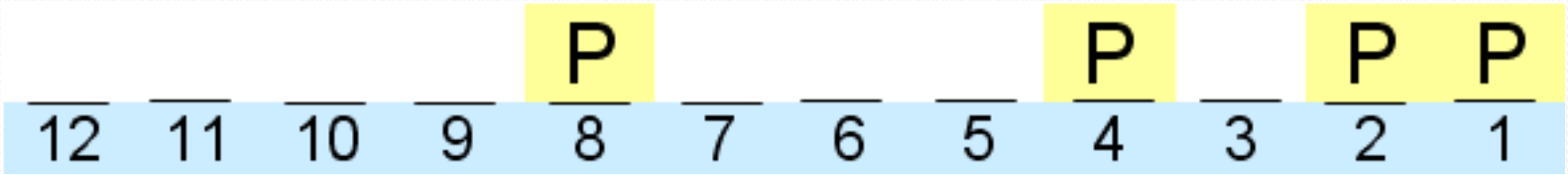
Error Detection and Correction

- Using our code words of length 12, number each bit position starting with 1 in the low-order bit.
- Each bit position corresponding to a power of 2 will be occupied by a check bit.
- The remaining bits (our data) can all be expressed as sums of powers of 2.
- Bit 10, for example, is $2^8 + 2^2$.



Error Detection and Correction

- Each data bit contributes to the parity of its corresponding powers of 2. Bit 10 contributes to bit 8 and 2.
- Put the other way around, each parity bit takes parity from it's corresponding data bits. Bit 4 takes parity from 12, 7, 6, and 5.



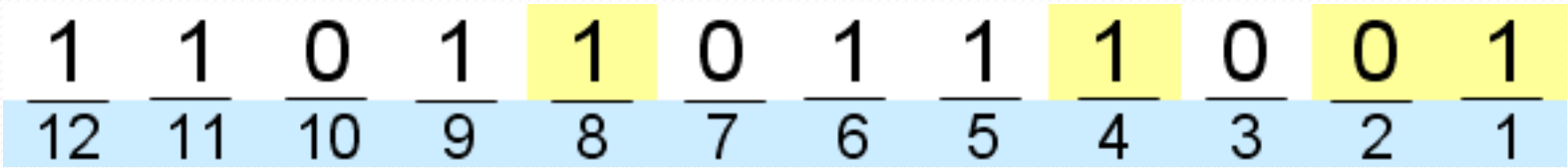
Error Detection and Correction

- Lets insert the data word 11010110 and use even parity for our check bits.
- Since 1 (= 2^0) contributes to the values 3, 5, 7, 9, and 11, bit 1 will check parity over these bits.
- Since 2 (= 2^1) contributes to the values 3, 6, 7, 10, and 11, bit 2 will check parity over these bits.

1	1	0	1		0	1	1		0	0	1
12	11	10	9	8	7	6	5	4	3	2	1

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

Error Detection and Correction



- The completed code word is shown above.
  - Bit 4 checks the bits 5, 6, 7, and 12, so its value is 1.
  - Bit 8 checks the bits 9, 10, 11, and 12, so its value is also 1.
- Using the Hamming algorithm, we can not only detect single bit errors in this code word, but also correct them!

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

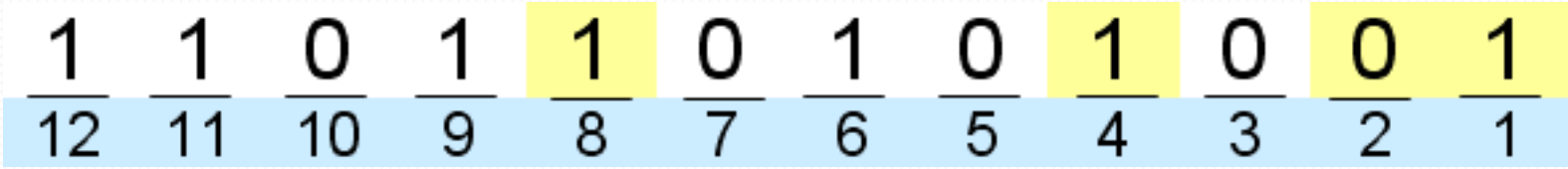
Error Detection and Correction

1	1	0	1	1	0	1	0	1	0	0	1
12	11	10	9	8	7	6	5	4	3	2	1

- Suppose an error occurs in bit 5, as shown above. Our parity bit values are:
  - Bit 1 checks 3, 5, 7, 9, and 11. *This is incorrect as we have a total of 2 ones (which is not odd parity).*
  - Bit 2 checks bits 3, 6, 7, 10, and 11. The parity is correct.
  - Bit 4 checks bits 5, 6, 7, and 12. *This parity is incorrect, as we have 2 ones.*
  - Bit 8 checks bit 9, 10, 11, and 12. This parity is correct.

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

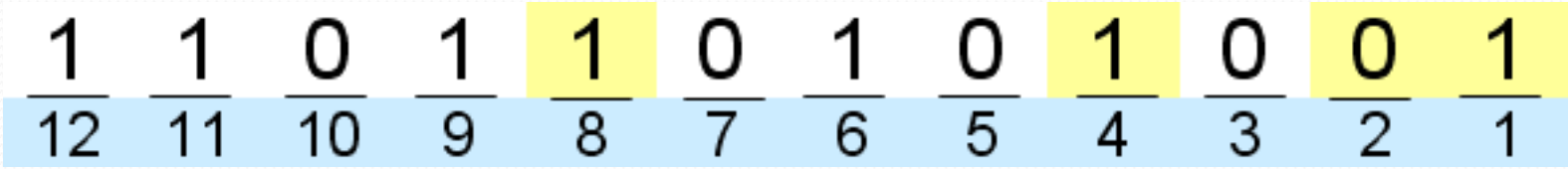
Error Detection and Correction



- We have erroneous parity for check bits 1 and 4.
- With *two* parity bits that don't check, we know that the error is in the data, and not in a parity bit.
  - Each data bit position is the sum of two or more powers of 2, so at least 2 parity bits would be affected if it was wrong.
  - If only one parity bit doesn't check, it must be the parity bit that is wrong.

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

Error Detection and Correction



- Which data bits are in error? Bit 1 points to 3, 5, 7, 9, and 11. Bit 4 points to 5, 6, 7, and 12. Together they point to 5 and 7. But bit 2, which also points to 7, shows no problem so the error must be bit 5.
- More simply, we find out by adding the bit positions of the erroneous bits.
- 1 + 4 = 5. This tells us that the error is in bit 5. Flip bit 5 to a 1 and our data is restored.

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

Data Representation

END OF CHAPTER