Data Representation in Computer Systems

Essentials of
Computer Organization and Architecture

SIXTH EDITION

Linda Null

NAVIGATE
ADVANTAGE ACCESS
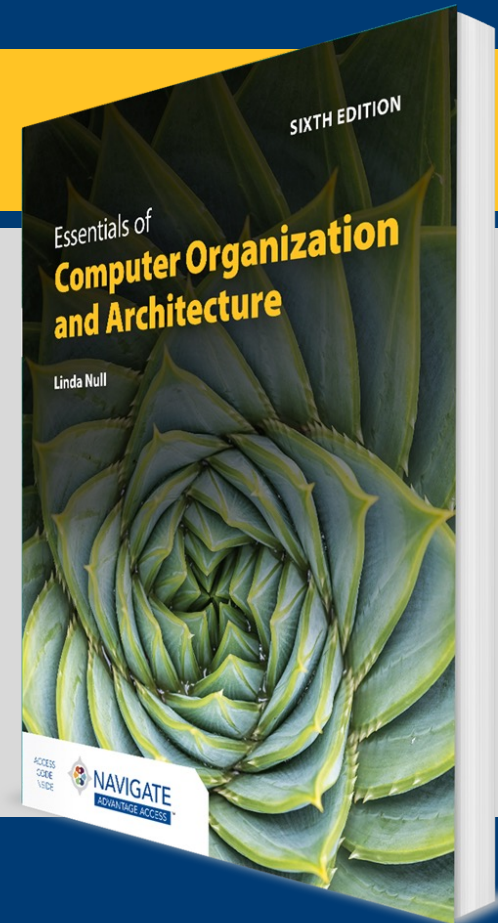
Objectives (1 of 2)

- Understand the fundamentals of numerical data representation and manipulation in digital computers.

- Master the skill of converting between various radix systems.

- Understand how errors can occur in computations because of overflow and truncation.

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.

- 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.

- 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.

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

- 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:

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

- The binary number 11001 in powers of 2 is:

$$1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$
$$= \quad 16 \quad + \quad 8 \quad + \quad 0 \quad + \quad 0 \quad + \quad 1 \quad = \quad 25$$

- When the radix of a number is something other than 10, the base is denoted by a subscript.

  - Sometimes, the subscript 10 is added for emphasis:

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

- Because binary numbers are the basis for all data representation in digital computer systems, it is important that you become proficient with this radix system.

- Your knowledge of the binary numbering system will enable you to understand the operation of all computer components as well as the design of instruction set architectures.

- In an earlier slide, we said that every integer value can be represented exactly using any radix system.

- There are two methods for radix conversion: the subtraction method and the division remainder method.

- The subtraction method is more intuitive, but cumbersome. It does, however, reinforce the ideas behind radix mathematics.

- Suppose we want to convert the decimal number 538_{10} to base 8.
  - The largest power of 8 that we need is $8^3 = 512$, and $512 \times 1 = 512$.
  - Write down the 1 and subtract 512 from 538, giving 26.

$$538$$
$$\underline{- \ 512} = 8^3 \times 1$$
$$26$$

not using subtraction method
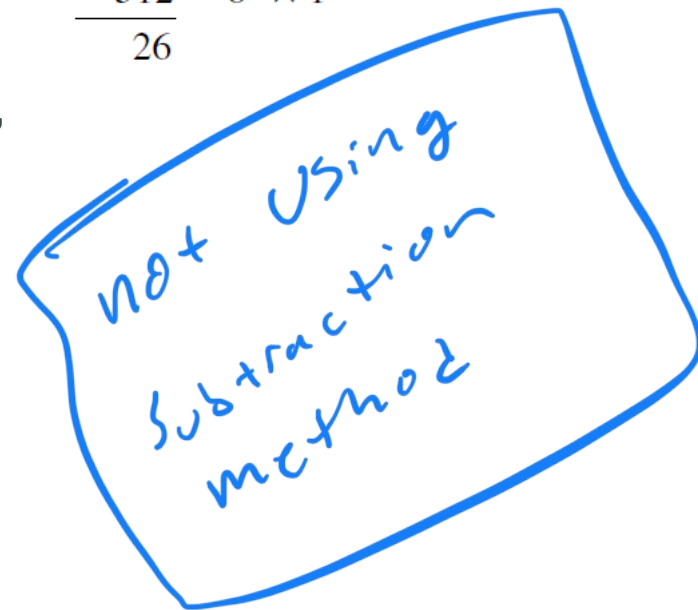
- Converting 538 to base 8
  - The next power of 8 is $8^2 = 64$. However, that is too large, so we write down 0.
  - The next power of 8, $8^1 = 8$, goes into our remainder three times, so we write down 3 and subtract $8 \times 3 = 24$. This leaves us 2.

$$538$$
$$-512 = 8^3 \times 1$$
$$\overline{26}$$
$$-0 = 8^2 \times 0$$
$$\overline{26}$$
$$-24 = 8^1 \times 3$$
$$\overline{2}$$

- Converting 538 to base 8

  - $8^0 = 1$, so we use $8^0 \times 2$. This is our last choice, and it gives us a difference of zero.

  - Our result, reading from top to bottom is:

    $$538_{10} = 1032_8$$

$$
\begin{array}{r}
538 \\
- 512 = 8^3 \times 1 \\
\hline
26 \\
- 0 = 8^2 \times 0 \\
\hline
26 \\
- 24 = 8^1 \times 3 \\
\hline
2 \\
- 2 = 8^0 \times 2 \\
\hline
0
\end{array}
\qquad 538_{10} = 1032_8
$$

- Another method of converting integers from decimal to some other radix uses division.

- This method is mechanical and easy.

- It employs the idea that successive division by a base is equivalent to successive subtraction by powers of the base.

- Let's use the division remainder method to again convert 190 in decimal to base 3.

- Converting 538 to base 8
  - 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 538 67 times, with a remainder of 2.
  - Record the quotient and the remainder.

$$8\overline{)538} \quad 2$$
$$\underline{67}$$

- Converting 538 to base 8
  - First divide 538 by 8, which gives us 67 with a remainder of 2.
  - Then divide 67 by 8, which gives us 8 with a remainder of 3.

$$
\begin{array}{rcl}
8\underline{|538} & 2 & \text{8 divides 538 67 times with a remainder of 2.} \\
8\underline{|67} & 3 & \text{8 divides 67 8 times with a remainder of 3.} \\
8\underline{|8} & 0 & \text{8 divides 8 1 time with a remainder of 0.} \\
8\underline{|1} & 1 & \text{8 divides 1 0 times with a remainder of 1.} \\
0 & &
\end{array}
$$

§ Converting 538 to base 8
  § Continue in this way until the quotient is zero.
  § In the final calculation, we note that 8 divides 1 zero times with a remainder of 1.
  § Our result, reading from bottom to top is:

$$538_{10} = 1032_8$$

8|538   2   8 divides 538 67 times with a remainder of 2.
8|67   3   8 divides 67 8 times with a remainder of 3.
8|8   0   8 divides 8 1 time with a remainder of 0.
8|1   1   8 divides 1 0 times with a remainder of 1.
0

Handwritten notes (left):

133

133 / 2 = 66 r 1
66 / 2 = 33 r 0
33 / 2 = 16 r 1
16 / 2 = 8 r 0
8 / 2 = 4 r 0
4 / 2 = 2 r 0
2 / 2 = 1 r 0
1 / 2 = 0 r 1

1 0 0 0 0 1 0 1

Handwritten notes (right):

Ex   567

567 / 2 = 283 r 1
283 / 2 = 141 r 1
141 / 2 = 70 r 1
70 / 2 = 55 r 0
35 / 2 = 17 r 1
17 / 2 = 8 r 1

8/2 = 4 r0

4/2 = 2 r0

2/2 = 1 r0

1/2 = 0 r1

- 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.

- 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:

$$0.47_{10} = 4 \times 10^{-1} + 7 \times 10^{-2}$$
$$0.11_{2} = 1 \times 2^{-1} + 1 \times 2^{-2}$$
$$= \tfrac{1}{2} + \tfrac{1}{4}$$
$$= 0.5 + 0.25 = 0.75$$

- 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.

- 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.

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

$.76 \; (10) = x \times (2)$

$.76 \cdot 2 = 1.52$

$.52 \cdot 2 = 1.04$

$.04 \cdot 2 = 0.08$

$.08 \cdot 2 = 0.16$

$0.16 \cdot 2 = 0.32$

$0.32 \cdot 2 = 0.64$

$0.64 \cdot 2 = 1.28$

$.11 0001$

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

```
  .8125
×     2
1.6250
```

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

```
  .8125
×    2
1.6250

  .6250
×    2
1.2500

  .2500
×    2
0.5000
```

- 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.

```
  . 8 1 2 5
× _____2
1 . 6 2 5 0

  . 6 2 5 0
× _____2
1 . 2 5 0 0

  . 2 5 0 0
× _____2
0 . 5 0 0 0

  . 5 0 0 0
× _____2
1 . 0 0 0 0
```

- 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: $11010100011011_2 = 13595_{10}$

- For compactness and ease of reading, binary values are usually expressed using the hexadecimal, or base-16, numbering system.

- 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.

- Using groups of hextets, the binary number 11010100011011_2 $(= 13595_{10})$ in hexadecimal is:

$$\boxed{0011}\ \boxed{0101}\ \boxed{0001}\ \boxed{1011}$$
$$3 \qquad 5 \qquad 1 \qquad B$$

> 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 ($8 = 2^3$):

$$\boxed{011}\ \boxed{010}\ \boxed{100}\ \boxed{011}\ \boxed{011}$$
$$3 \quad 2 \quad 4 \quad 3 \quad 3$$

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

not doing

- The conversions we have so far presented have involved only unsigned numbers.

- To represent signed integers, computer systems allocate the high-order bit to indicate the sign of a number.
  - The high-order bit is the leftmost bit. It is also called the most significant bit.
  - 0 is used to indicate a positive number; 1 indicates a negative number.

- The remaining bits contain the value of the number (but this can be interpreted different ways).

- There are three ways in which signed binary integers may be expressed:
  - Signed magnitude
  - One's complement
  - Two's complement

- In an 8-bit word, *signed magnitude* representation places the absolute value of the number in the 7 bits to the right of the sign bit.

- 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.

- 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.
  - We will describe these circuits in Chapter 3.

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

- Example:
  - Using signed magnitude binary arithmetic, find the sum of 75 and 46.

- First, convert 75 and 46 to binary, and arrange as a sum, but separate the (positive) sign bits from the magnitude bits.

```
0   1001011
0 + 0101110
  _____
```

- Example:
  - Using signed magnitude binary arithmetic, find the sum of 75 and 46.

- Just as in decimal arithmetic, we find the sum starting with the rightmost bit and work left.

```
0   1 0 0 1 0 1 1
0 + 0 1 0 1 1 1 0
              1
```

- Example:
  - Using signed magnitude binary arithmetic, find the sum of 75 and 46.

```
                1
  0   1 0 0 1 0 1 1
  0 + 0 1 0 1 1 1 0
  _____
              0 1
```

- In the second bit, we have a carry, so we note it above the third bit.

- Example:
  - Using signed magnitude binary arithmetic, find the sum of 75 and 46.

- The third and fourth bits also give us carries.

```
        1 1 1
0    1 0 0 1 0 1 1
0  + 0 1 0 1 1 1 0
   ─────────────────
           1 0 0 1
```

- 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.

```
            1 1 1
0      1 0 0 1 0 1 1
0  +   0 1 0 1 1 1 0
0      1 1 1 1 0 0 1
```

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.

- 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.
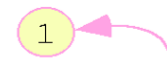
```
              1   1 1 1
   0      1 1 0 1 0 1 1
   0  +   0 1 0 1 1 1 0
   0      0 0 1 1 0 0 1
```

- 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.

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

- 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.

$$
\begin{array}{ccccccccc}
 & & 0\,2 & & & 0\,2 & & \\
0 & & 0\,\cancel{1}\,0\,1\,1\,\cancel{1}\,0 \\
1 & + & 0\,0\,1\,1\,0\,0\,1 \\
\hline
0 & & 0\,0\,1\,0\,1\,0\,1 \\
\end{array}
$$

- Signed magnitude representation is easy for people to understand, but it requires complicated computer hardware.

- Another disadvantage of signed magnitude is that it allows two different representations for zero: positive zero and negative zero.

- For these reasons (among others), computer systems employ *complement systems* for numeric value representation.

- 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.

- 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 between two values is found by adding the minuend to the complement of the subtrahend.

- 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`.

```
  1    1 1
       00110000
       11101100
       00011100
            + 1
       00011101
```
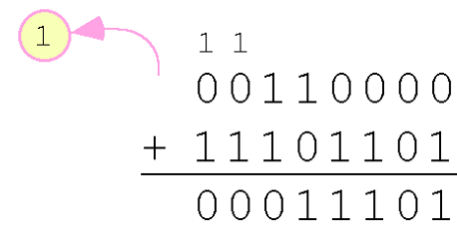
- Although the "end carry around" adds some complexity, one's complement is simpler to implement than signed magnitude.

- 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 $r^d - N$.

- 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.`

- 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.
- We note that 19 in binary is: `00010011`,
  - so −19 using one's complement is: `11101100`,
  - and −19 using two's complement is: `11101101`.

```
       1 1
     00110000
  +  11101101
  _____
     00011101
```

- 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 represent the integer 2M.

- The integer is interpreted as positive or negative depending on where it falls in the range.

- 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.

- 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 $+ 8_{10}$.

- Let's 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

- 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.

- Example:
    - Using two's complement binary arithmetic, find the sum of 107 and 46.

$$\begin{array}{r} {\scriptstyle 1\ 1\ \ \ 1\ 1\ 1} \\ 0\ 1\ 1\ 0\ 1\ 0\ 1\ 1 \\ +\ 0\ 0\ 1\ 0\ 1\ 1\ 1\ 0 \\ \hline 1\ 0\ 0\ 1\ 1\ 0\ 0\ 1 \end{array}$$

- We see that the nonzero carry from the seventh bit *overflows* into the sign bit, giving us the erroneous result: $107 + 46 = -103$.

- But overflow into the sign bit does not always mean that we have an error.

- 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.

$$
\begin{array}{r}
\boxed{1} \leftarrow \boxed{1}\,1\,1 \quad 1\,1\,1 \\
0\,0\,0\,1\,0\,1\,1\,1 \\
+\ 1\,1\,1\,1\,0\,1\,1\,1 \\
\hline
0\,0\,0\,0\,1\,1\,1\,0
\end{array}
$$

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.

- Signed and unsigned numbers are both useful.
  - For example, memory addresses are always unsigned.

- Using the same number of bits, unsigned integers can express twice as many "positive" values as signed numbers.

- Trouble arises if an unsigned value "wraps around."
  - In four bits: 1111 + 1 = 0000.

- Good programmers stay alert for this kind of problem.

- Research into finding better arithmetic algorithms has continued for over 50 years.

- One of the many interesting products of this work is Booth's algorithm.

- In most cases, Booth's algorithm carries out multiplication faster and more accurately than pencil-and-paper methods.

- The general idea is to replace arithmetic operations with bit shifting to the extent possible.

- 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   0110
    +   0000      (shift)
    -   0011      (subtract)
    +   0000      (shift)
    +   0011      (add)
    _____
      00010010
```

We see that 3 × 6 = 18!

- Here is a larger example.

Ignore all bits over 2n.

$53 \times 126 = 6678!$

```
                  00110101
x                 01111110
+  0000000000000000
+  11111111001011
+  0000000000000
+  000000000000
+  00000000000
+  0000000000
+  0000000000
+  000110101
   10001101000010110
```

- 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.

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

- 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.

- 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.

- 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.

1234 (16)
+ 0AC2
ICF6

1
1284
+ 0 A C 2
I D 4 6

- The signed magnitude, one's complement, and two's complement representations that we have just presented deal with signed 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:
    - $0.125 = 1.25 \times 10^{-1}$
    - $5,000,000 = 5.0 \times 10^{6}$

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

- Numbers written in scientific notation have three components:
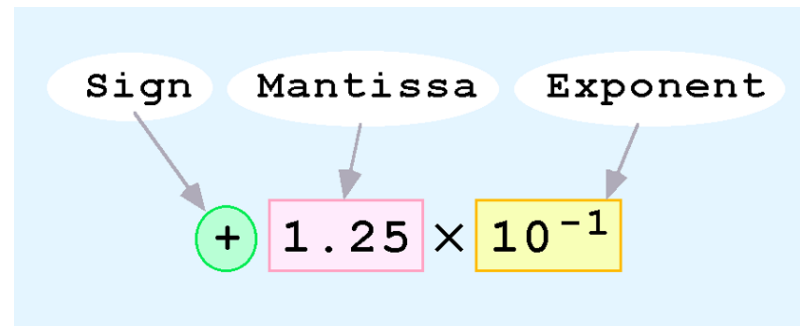
- Computer representation of a floating-point number consists of three fixed-size fields:
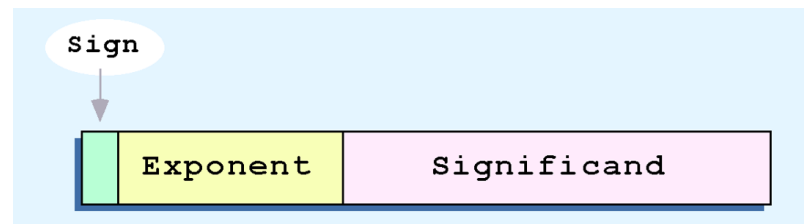
- This is the standard arrangement of these fields.

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.

- 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.
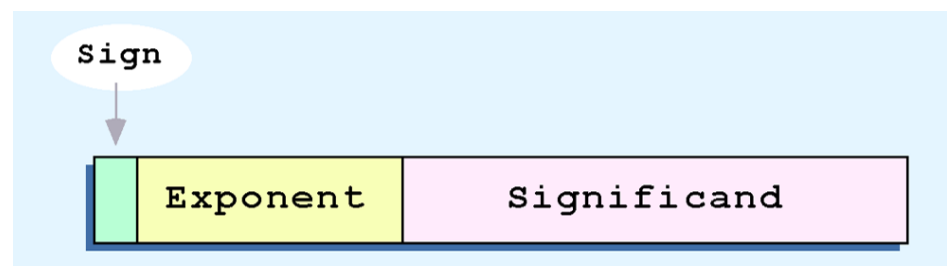
- We introduce a hypothetical "Simple Model" to explain the concepts.

- In this model:
  - A floating-point number is 14 bits in length.
  - The exponent field is 5 bits.
  - The significand field is 8 bits.
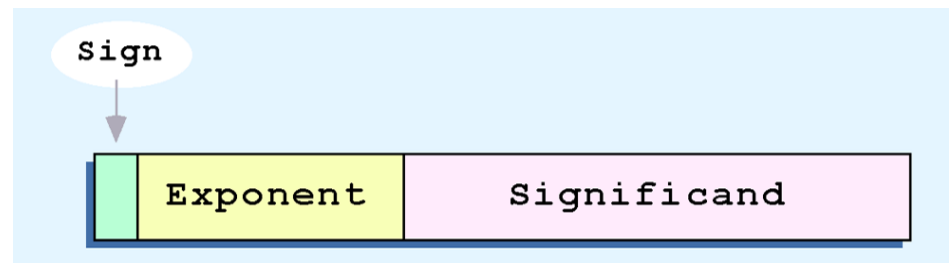
- The significand is always preceded by an implied binary point.

- Thus, the significand always contains a fractional binary value.

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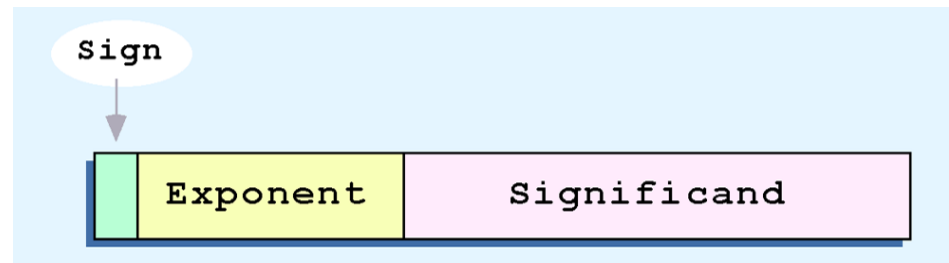
- Example:
  - Express 32_{10} in the simplified 14-bit floating-point model.
- We know that 32 is 2^5. So, in (binary) scientific notation $32 = 1.0 \times 2^5 = 0.1 \times 2^6$.
  - In a moment, we'll explain why we prefer the second notation versus the first.
- Using this information, we put 110 ($=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

0	0 0 1 1 0	1 0 0 0 0 0 0 0
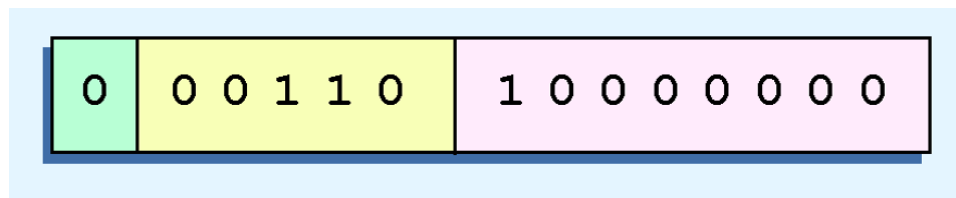
- The illustrations shown on the right are all equivalent representations for 32 using our simplified model.

- Not only do these synonymous representations waste space, but they can also cause confusion.

0	0 0 1 1 0	1 0 0 0 0 0 0 0
0	0 0 1 1 1	0 1 0 0 0 0 0 0
0	0 1 0 0 0	0 0 1 0 0 0 0 0
0	0 1 0 0 1	0 0 0 1 0 0 0 0
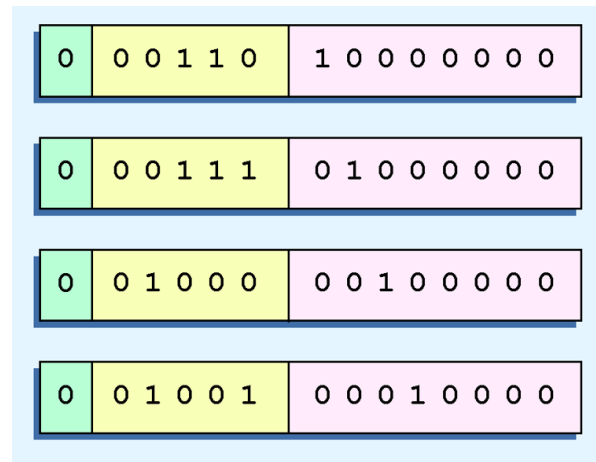
- Another problem with our system is that we have made no allowances for negative exponents. We have no way to express 0.5 ($= 2^{-1}$)! (Notice that there is no sign in the exponent field.)

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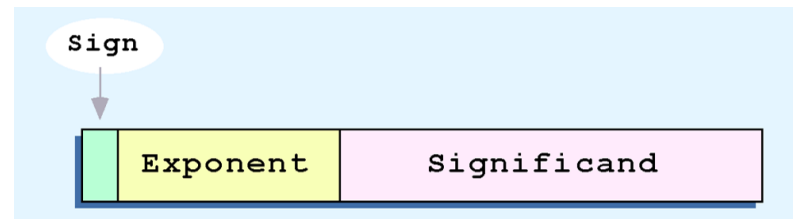
All of these problems can be fixed with no changes to our basic model.

- 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.

- 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, we 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.

- 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:

0	1 0 1 0 1	1 0 0 0 0 0 0 0
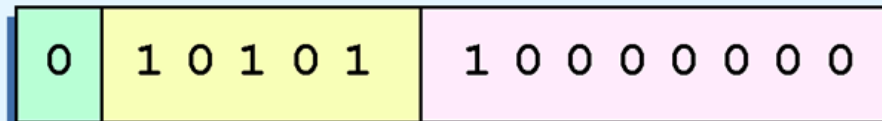
- Example:
  - Express 0.0625_{10} in the revised 14-bit floating-point model.

- 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-15 biased exponent, we add 15 to -3, giving 12_{10} (= 01100_2).

0	0 1 1 0 0	1 0 0 0 0 0 0 0
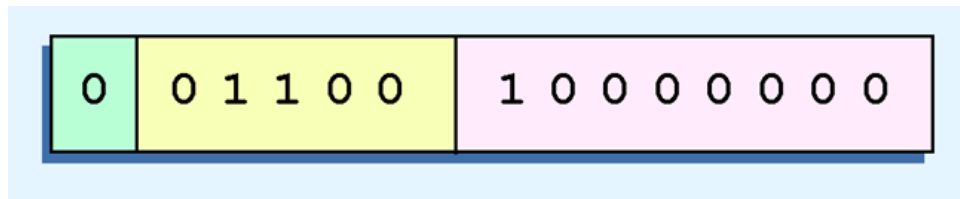
- Example:
  - Express -26.625_{10} in the revised 14-bit floating-point model.

- 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.

1	1 0 1 0 0	1 1 0 1 0 1 0 1
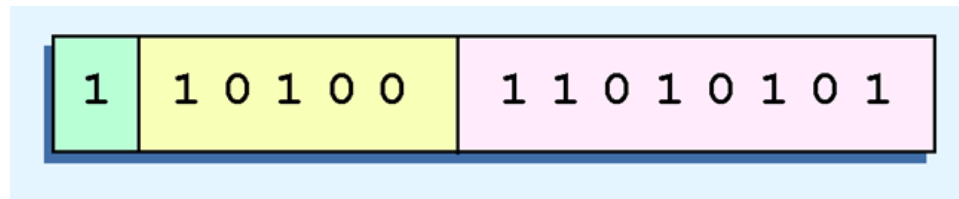
- The IEEE has established a standard for floating-point numbers.

- 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.

- In both the IEEE single-precision and double-precision floating-point standard, the significand 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).

- 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)
  - The first 1 in the significand is implied, so we have:

1	1	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

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

- 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.

- 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.

- 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.

- 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 $1.25_{10} = 0.101 \times 2^1 = 0.000101 \times 2^4$.

$$
\begin{array}{c|c|c|c}
 & 0 & 10011 & 11000000 \\
+ & 0 & 10011 & 00010100 \\
\hline
 & 0 & 10011 & 11010100 \\
\end{array}
$$

- Thus, our sum is 0.110101×2^4.

- 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 $19_{10} = 10011_2$.

- 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.

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

- 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.

- 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.

- 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.

- 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.

- 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.

- This means that we cannot assume:
    - (a + b) + c = a + (b + c) 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 (abs(x - 2) < epsilon) then ...
    - (assuming we have epsilon defined correctly!)

- Calculations aren't useful until their results can be displayed in a manner that is meaningful to people.

- We also need to store the results of calculations and provide a means for data input.

- Thus, human-understandable characters must be converted to computer-understandable bit patterns using some sort of character encoding scheme.

- 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.

- 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.

- 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.

- 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.

- The Unicode codespace allocation is shown on the right.

- The lowest-numbered Unicode characters comprise the ASCII code.

- The highest provide for user-defined codes.

Character Types	Character Set Description	Number of Characters	Hexadecimal Values
Alphabets	Latin, Cyrillic, Greek, 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
	Expansion or spillover from Han	4096	E000 to EFFF
User defined		4095	F000 to FFFE

- 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 are critical to accurate data transmission, storage, and retrieval.

- Check digits, appended to the end of a long number, can provide some protection against data input errors.
  - The last characters of UPC barcodes and ISBNs are check digits.

- Longer data streams require more economical and sophisticated error detection mechanisms.

- Cyclic redundancy checking (CRC) codes provide error detection for large blocks of data.

▪ 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.

- Modulo 2 arithmetic works like clock arithmetic.

- In clock arithmetic, if we add 2 hours to 11:00, we get 1:00.

- In modulo 2 arithmetic, if we add 1 to 1, we get 0. The addition rules couldn't be simpler:

$$0 + 0 = 0 \qquad 0 + 1 = 1$$
$$1 + 0 = 1 \qquad 1 + 1 = 0$$

> You will fully understand why modulo 2 arithmetic is so handy after you study digital circuits in Chapter 3.

- 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.

$$
\begin{array}{r}
1 \\
1101 \overline{)\,1111101} \\
1101 \\
\hline
0010
\end{array}
$$

- 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.

```
              10
      _____
1101 ) 1111101
       1101
       _____
        00101
```

- 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.

```
                    101
          1101)1111101
                1101
               001010
                1101
                0111
```

- 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.

```
                1011
         _____
  1101)1111101
         1101
         _____
         001010
           1101
           _____
           01111
             1101
             _____
             0010
```

Note: The divisor in this example corresponds to a modulo 2 polynomial: $X^3 + X^2 + 1$.

- 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
```

- 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 on the right.

- Real applications use longer polynomials to cover larger information strings.
  - Some of the standard polynomials are listed in the text.

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

- 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.

- 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.

- Hamming codes are code words formed by adding redundant check bits, or parity bits, to a data word.

- The *Hamming distance* between two code words is the number of bits in which two code words differ.

- The minimum Hamming distance for a code is the smallest Hamming distance between all pairs of words in the code.

This pair of bytes has a Hamming distance of 3:

```
1 0 0 0 1 0 0 1
1 0 1 1 0 0 0 1
```

- The minimum Hamming distance for a code, $D(min)$, determines its error-detecting and error-correcting capability.

- For any code word, X, to be interpreted as a different valid code word, Y, at least $D(min)$ single-bit errors must occur in X.

- Thus, to detect k (or fewer) single-bit errors, the code must have a Hamming distance of $D(min) = k + 1$.

- Hamming codes can *detect* $D(min) - 1$ errors and *correct* errors.

$$\left\lfloor \frac{D(min) - 1}{2} \right\rfloor$$

- 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.

- Suppose we have a set of n-bit code words consisting of m data bits and r (redundant) parity bits.

- Suppose also that we wish to detect and correct one single-bit error only.

- An error could occur in any of the n bits, so each code word can be associated with n invalid code words at a Hamming distance of 1.

- Therefore, we have $n + 1$-bit patterns for each code word: one valid code word, and n invalid code words.

- 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 code word, 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.

- Suppose we have data words of length $m = 4$. Then:

    $$(4 + r + 1) \leq 2^r$$

    implies that r must be greater than or equal to 3.

    - We should always use the smallest value of r that makes the inequality true.

- This means to build a code with 4-bit data words that will correct single-bit errors, we must add 3 check bits.

- Finding the number of check bits is the hard part. The rest is easy.

- 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.

- 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.

- So how do we assign values to these check bits?

- With code words of length 12, we observe that each of the bits, numbered 1 though 12, can be expressed in powers of 2. Thus:

$$1 = 2^0 \qquad\qquad 5 = 2^2 + 2^0 \qquad\qquad 9 = 2^3 + 2^0$$
$$2 = 2^1 \qquad\qquad 6 = 2^2 + 2^1 \qquad\qquad 10 = 2^3 + 2^1$$
$$3 = 2^1 + 2^0 \qquad 7 = 2^2 + 2^1 + 2^0 \qquad 11 = 2^3 + 2^1 + 2^0$$
$$4 = 2^2 \qquad\qquad 8 = 2^3 \qquad\qquad\qquad 12 = 2^3 + 2^2$$

  - 1 (= 2^0) contributes to all of the odd-numbered digits.
  - 2 (= 2^1) contributes to the digits, 2, 3, 6, 7, 10, and 11.
  - . . . And so forth.
  - We can use this idea in the creation of our check bits.

- 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.

- These check bits contain the parity of each bit position for which it participates in the sum.

_	_	_	_	P	_	_	_	P	_	P	P
12	11	10	9	8	7	6	5	4	3	2	1
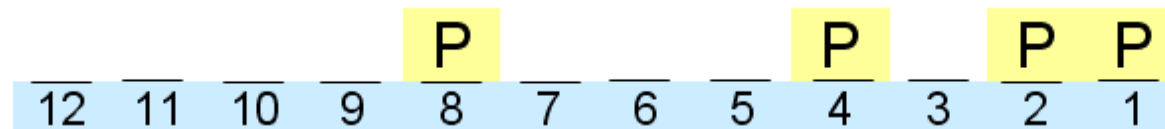
- Since 1 ($=2^0$) contributes to the values 1, 3, 5, 7, 9, and 11, bit 1 will check parity over bits in these positions.
- Since 2 ($=2^1$) contributes to the values 2, 3, 6, 7, 10, and 11, bit 2 will check parity over these bits.
- For the word 11010110, assuming even parity, we have a value of 1 for check bit 1, and a value of 0 for check bit 2.

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

What are the values for the other parity bits?

- The completed code word is shown above.
  - Bit 1 checks the bits 3, 5, 7, 9, and 11, so its value is 1 to ensure even parity within this group.
  - 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!

1	1	0	1	1	0	1	1	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 1, 3, 5, 7, 9, and 11. This is incorrect as we have a total of 3 ones (which is not even parity).
  - Bit 2 checks bits 2, 3, 6, 7, 10, and 11. The parity is correct.
  - Bit 4 checks bits 4, 5, 6, 7, and 12. This parity is incorrect, as we have 3 ones.
  - Bit 8 checks bits 8, 9, 10, 11, and 12. This parity is correct.

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

- 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.

- Which data bits are in error? We find out by adding the bit positions of the erroneous bits.

- Simply, 1 + 4 = 5. This tells us that the error is in bit 5. If we change bit 5 to a 1, all parity bits check, and our data is restored.

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

Conclusion

- Computers store data in the form of bits, bytes, and words using the binary numbering system.

- Hexadecimal numbers are formed using four-bit groups called nibbles.

- Signed integers can be stored in one's complement, two's complement, or signed magnitude representation.

- Floating-point numbers are usually coded using the IEEE 754 floating-point standard.

- Floating-point operations are not necessarily commutative or distributive.

- Character data is stored using ASCII, EBCDIC, or Unicode.

- Error detecting and correcting codes are necessary because we can expect no transmission or storage medium to be perfect.

- CRC, Reed-Solomon, and Hamming codes are three important error control codes.