FLOATING-POINT REPRESENTATION

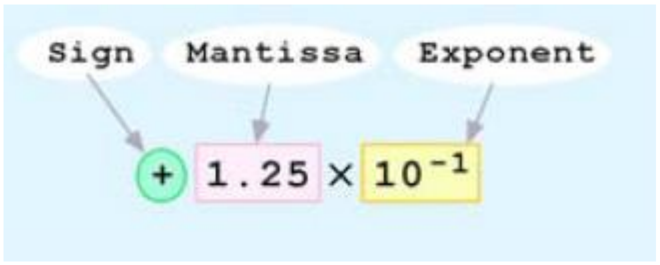
- Most of today's computers are equipped with specialized hardware that performs floating-point arithmetic with no special programming required.
- Floating-point numbers allow an arbitrary number of decimal places to the right of the decimal point.

  – For example: $0.125 = 1.25 \times 10^{-1}$

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

Simple Model

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



- We will use a 14-bit model with a 5-bit exponent, an 8-bit significand, and a sign bit (see Figure 2.1).

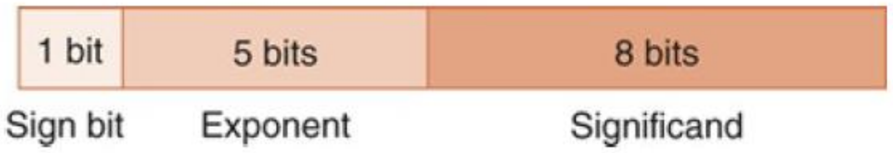
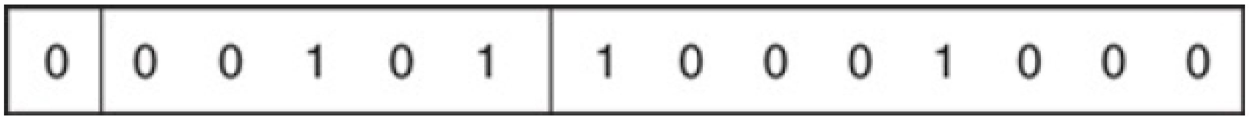


FIGURE 2.1 A Simple Model Floating-Point Representation

- 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.

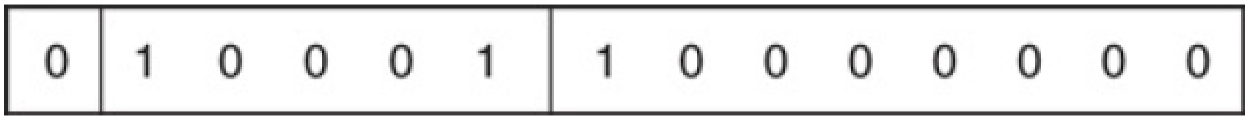
- Let’s say that we wish to store the decimal number 17 in our model.
- We know that 17 = $17.0 \times 10^0 = 1.7 \times 10^1 = 0.17 \times 10^2$.
- Analogously, in binary,17 = $(10001 \times 2^0) = (1000.1 \times 2^1) = (100.01\times 2^2) = (10.001 \times 2^3) = (1.0001 \times 2^4) = (0.10001 \times 2^5)$.

- If we use this last form, our fractional part will be **10001000** and our exponent will be **00101**, as shown here:



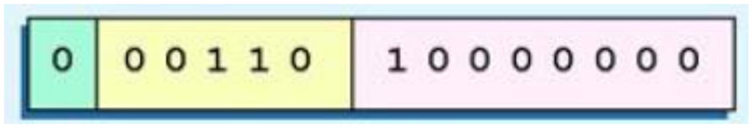
0	0	0	1	0	1	1	0	0	0	1	0	0	0

- Using this form, we can store numbers of much greater magnitude than we could using a fixed-point representation of 14 bits (which uses a total of 14 binary digits plus a binary, or radix, point).
- If we want to represent 65536 = 0.1×2^{17} in this model, we have:

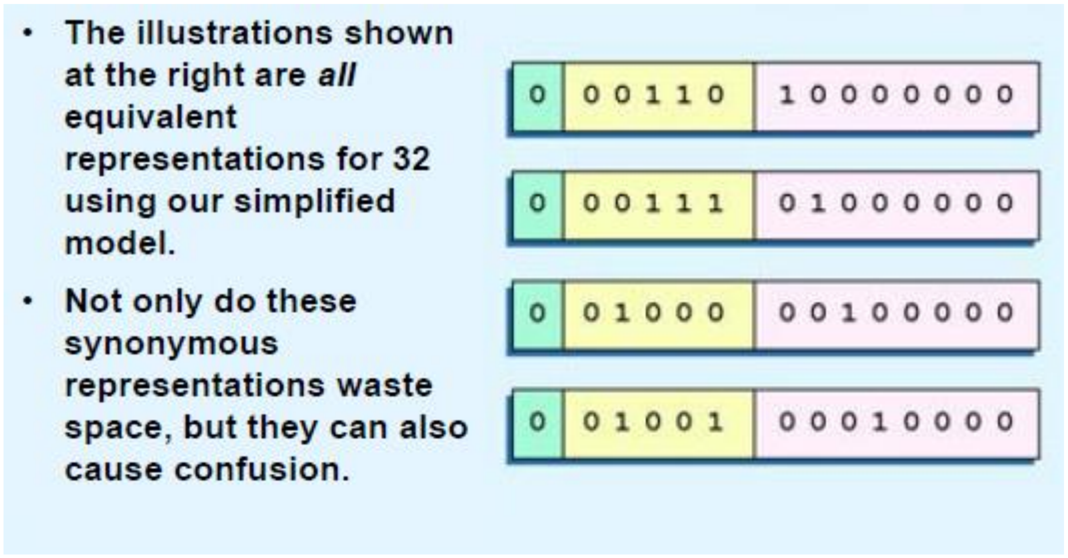


0	1	0	0	0	1	1	0	0	0	0	0	0	0

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



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



- 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.

- One obvious problem with this model is that we haven’t provided for negative exponents.
- If we wanted to store 0.25, we would have no way of doing so because 0.25 is 2^{-2} and the exponent -2 cannot be represented.
- 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 15 for our bias. This is called excess-15 representation.
- In our model, exponent values less than 15 are negative, representing fractional numbers.
- Returning to our example of storing 17, we calculated $17 = 0.10001_2 \times 2^5$
- If we update our model to use a biased exponent, the biased exponent is 15 + 5 = 20:

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

- If we wanted to store $0.25 = 0.1 \times 2^{-1}$, we would have:

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

EXAMPLE 2.34

Express 0.03125_{10} in normalized floating-point form using the simple model with excess-15 bias.

$0.03125_{10} = 0.00001_2 \times 2^0 = 0.0001 \times 2^{-1} = 0.001 \times 2^{-2} = 0.01 \times 2^{-3} = 0.1 \times 2^{-4}$. Applying the bias, the exponent field is $15 - 4 = 11$.

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

CHARACTER CODES

- We have seen how digital computers use the binary system to represent and manipulate numeric values.
- We have yet to consider how these internal values can be converted to a form that is meaningful to humans.
- The manner in which this is done depends on both the coding system used by the computer and how the values are stored and retrieved.

Binary-Coded Decimal

- Binary-coded decimal (BCD) is very common in electronics, particularly those that display numerical data, such as alarm clocks and calculators.
- BCD encodes each digit of a decimal number into a 4-bit binary form.
- Each decimal digit is individually converted to its binary equivalent.
- For example, to encode 146, the decimal digits are replaced by 0001, 0100, and 0110, respectively.

ASCII

- The American Standard Code for Information Interchange (ASCII).
- ASCII is a direct descendant of the coding schemes used for decades by teletype (telex) devices.

Unicode

- Both EBCDIC and ASCII were built around the Latin alphabet. As such, they are restricted in their abilities to provide data representation for the non-Latin alphabets used by the majority of the world's population.
- Unicode is a 16-bit alphabet that is downward compatible with ASCII and the Latin-1character set.
- Because the base coding of Unicode is 16 bits, it has the capacity to encode the majority of characters used in every language of the world.

ERROR DETECTION AND CORRECTION

- No communications channel or storage medium can be completely error-free. It is a physical impossibility.
- As more bits are packed per square millimeter of storage, magnetic flux densities increase. Error rates increase in direct proportion to the number of bits per second transmitted, or the number of bits per square millimeter of magnetic storage.

Cyclic Redundancy Check

- A cyclic redundancy check (CRC) is a type of checksum used primarily in data communications that determines whether an error has occurred within a large block or stream of information bytes.
- The larger the block to be checked, the larger the checksum must be to provide adequate protection.
- Checksums and CRCs are types of systematic error detection schemes, meaning that the errorchecking bits are appended to the original information byte.
- The group of error-checking bits is called a syndrome.

- The original information byte is unchanged by the addition of the error-checking bits.

Arithmetic Modulo 2

The addition rules are as follows:

$$0 + 0 = 0$$
$$0 + 1 = 1$$
$$1 + 0 = 1$$
$$1 + 1 = 0$$

Find the sum of 1011_2 and 110_2 modulo 2.

$$\begin{array}{r} 1011 \\ +\,110 \\ \hline 1101_2 \text{ (mod 2)} \end{array}$$

This sum makes sense only in modulo 2.

- Modulo 2 division operates through a series of partial sums using the modulo 2 addition rules.

EXAMPLE 2.39

- Find the quotient and remainder when 1001011 is divided by 1011.

1011)1001011	1. Write the divisor directly beneath the first bit of the dividend.
$\underline{1011}$	2. Add these numbers using modulo 2.
0010	3. Bring down bits from the dividend so that the first 1 of the difference can align with the first 1 of the divisor.
001001	4. Copy the divisor as in Step 1.
$\underline{1011}$	5. Add as in Step 2.
0010	6. Bring down another bit.
00101	7. 101_2 is not divisible by 1011_2, so this is the remainder.

The quotient is 1010_2.

- We can now proceed to show how CRCs are constructed.
- We will do this by example:

1. Let the information byte I = 1001011. (Any number of bytes can be used to form a message block.)
2. The sender and receiver agree upon an arbitrary binary pattern, say, P = 1011. (Patterns beginning and ending with 1 work best.)

3. Shift **I** to the left by one less than the number of bits in P, giving a new **I** = 1001011000.

4. Using **I** as a dividend and P as a divisor, perform the modulo 2 division (as shown in Example 2.39). We ignore the quotient and note that the remainder is 100.
The remainder is the actual CRC checksum.

5. Add the remainder to **I**, giving the message M:
1001011000 + 100 = 1001011100

6. M is decoded and checked by the message receiver using the reverse process.

- Only now P divides M exactly:

1. Let the information byte I = 1001011_2. (Any number of bytes can be used to form a message block.)

2. The sender and receiver agree upon an arbitrary binary pattern, say, P = 1011_2. (Patterns beginning and ending with 1 work best.)

3. Shift I to the left by one less than the number of bits in P, giving a new I = 1001011000_2.

4. Using I as a dividend and P as a divisor, perform the modulo 2 division (as shown in Example 2.39). We ignore the quotient and note that the remainder is 100_2. The remainder is the actual CRC checksum.

5. Add the remainder to I, giving the message M:
   $1001011000_2 + 100_2 = 1001011100_2$

6. M is decoded and checked by the message receiver using the reverse process. Only now P divides M exactly:

```
          1010100
     ____________
1011)1001011100
     1011
     001001
        1011
        0010
        001011
           1011
           0000
```

1. Note: The reverse process would include appending the remainder.

- A remainder other than 0 indicates that an error has occurred in the transmission of M.

Hamming Codes

- Data communications channels are simultaneously more error prone and more tolerant of errors than disk systems.
- In data communications, it is sufficient to have only the ability to detect errors.
- If a communications device determines that a message contains an erroneous bit, all it has to do is request retransmission.
- Storage systems and memory do not have this luxury.
- A disk can sometimes be the sole repository of a financial transaction or other collection of nonreproducible real-time data.
- Storage devices and memory must therefore have the ability to not only detect but to correct a reasonable number of errors.
- One of the most effective codes—and the oldest—is the Hamming code.
- Hamming codes are an adaptation of the concept of **parity**, whereby error detection and correction capabilities are increased in proportion to the number of parity bits added to an information word.
- Hamming codes are used in situations where random errors are likely to occur.
- We mentioned that Hamming codes use **parity** bits, also called check bits or redundant bits.
- The memory word itself consists of **m** bits, but **r** redundant bits are added to allow for error detection and/or correction.
- Thus, the final word, called a **code word**, is an **n-bit** unit containing **m** data bits and **r** check bits.
- There exists a unique code word consisting of n = m + r bits for each data word as follows:

m bits	*r* bits

- The number of bit positions in which two code words differ is called the Hamming distance of those two code words.

- For example, if we have the following two code words:

```
1 0 0 0 1 0 0 1
1 0 1 1 0 0 0 1
    * * *
```

- We see that they differ in 3 bit positions (marked by *), so the Hamming distance of these two code words is 3. (we have not yet discussed how to create code words; we will do that shortly.)
- The Hamming distance between two code words is important in the context of error detection.
- If we wish to create a code that guarantees detection of all single-bit errors (an error in only 1 bit), all pairs of code words must have a Hamming distance of at least 2.
- So, to detect k (or fewer) single-bit errors, the code must have a Hamming distance of $D(min) = k + 1$.
- Hamming codes can always detect $D(min) - 1$ errors and correct $\lfloor (D(min) - 1)/2 \rfloor$ errors.
- Accordingly, the Hamming distance of a code must be at least $2k + 1$ in order for it to be able to correct k errors.
- Code words are constructed from information words using **r** parity bits.
- Because each code word consists of **n** bits, where $n = m + r$, there are 2^n total bit patterns possible.
- Because $n = m + r$, we can write 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 is important because it specifies the lower limit on the number of check bits required (we always use as few check bits as possible) to construct a code with m data bits and r check bits that corrects all single-bit errors.

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

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

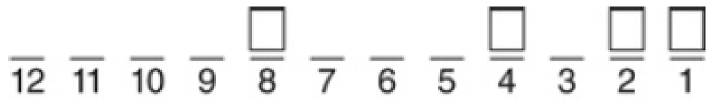
- Which implies that **r** must be greater than or equal to 3.
- We choose **r = 3**. This means to build a code with data words of 4 bits that should correct single-bit errors, we must add 3 check bits.

The Hamming Algorithm

- The Hamming algorithm provides a straightforward method for designing codes to correct single-bit errors.
- To construct error-correcting codes for any size memory word, we follow these steps:
  1. Determine the number of check bits, **r**, necessary for the code and then number the **n** bits (where n = m + r), right to left, starting with 1 (not 0).
  2. Each bit whose bit number is a power of 2 is a parity bit—the others are data bits.
  3. Assign parity bits to check bit positions as follows: Bit **b** is checked by those parity bits b_1 , b_2 , . . . , b_j such that $b_1 + b_2 + . . . + b_j = b$ (where "+" indicates the modulo 2 sum).
- We now present an example to illustrate these steps and the actual process of error correction.
  EXAMPLE
- Using the Hamming code just described and even parity, encode the 8-bit ASCII character K. (The high-order bit will be 0.) Induce a single-bit error and then indicate how to locate the error.
- We first determine the code word for K.

Step 1: Determine the number of necessary check bits, add these bits to the data bits, and number all n bits.

- Because m = 8, we have: $(8 + r + 1) < 2^r$, which implies that r must be greater than or equal to 4. We choose r = 4.
- Step 2: Number the n bits right to left, starting with 1, which results in:



- The parity bits are marked by boxes.
- Step 3: Assign parity bits to check the various bit positions.
- To perform this step, we first write all bit positions as sums of those numbers that are powers of 2 (1, 2, 4, 8):
  - 1 = 1
  - 2 = 2
  - 3 = 1 + 2
  - 4 = 4
  - 5 = 1 + 4
  - 6 = 2 + 4
  - 7 = 1 + 2 + 4
  - 8 = 8
  - 9 = 1 + 8
  - 10 = 2 + 8
  - 11 = 1 + 2 + 8
  - 12 = 4 + 8
- The number 1 contributes to 1, 3, 5, 7, 9, and 11, so this parity bit will reflect the parity of the bits in these positions.
- Similarly, 2 contributes to 2, 3, 6, 7, 10, and 11, so the parity bit in position 2 reflects the parity of this set of bits.
- Bit 4 provides parity for 4, 5, 6, 7, and 12, and bit 8 provides parity for bits 8, 9, 10, 11, and 12.

- If we write the data bits in the nonboxed blanks, and then add the parity bits, we have the following code word as a result:

0	1	0	0	$\boxed{1}$	1	0	1	$\boxed{0}$	1	$\boxed{1}$	$\boxed{0}$
12	11	10	9	8	7	6	5	4	3	2	1

Therefore, the code word for K is 010011010110.

- Let's introduce an error in bit position $\mathbf{b_9}$, resulting in the code word 010111010110.
- If we use the parity bits to check the various sets of bits, we find the following:
- Bit 1 checks 1, 3, 5, 7, 9, and 11: With even parity, this produces an error.
- Bit 2 checks 2, 3, 6, 7, 10, and 11: This is okay.
- Bit 4 checks 4, 5, 6, 7, and 12: This is okay.
- Bit 8 checks 8, 9, 10, 11, and 12: This produces an error.
- Parity bits 1 and 8 show errors.
- These two parity bits both check 9 and 11, so the single-bit error must be in either bit 9 or bit 11.
- However, because bit 2 checks bit 11 and indicates no error has occurred in the subset of bits it checks, the error must occur in bit 9. (We know this because we created the error;
- However, note that even if we have no clue where the error is, using this method allows us to determine the position of the error and correct it by simply flipping the bit.)
- An easier method to detect and correct the error bit is to add the positions of the parity bits that indicate an error.
- We found that parity bits 1 and 8 produced an error, and 1 + 8 = 9, which is exactly where the error occurred.