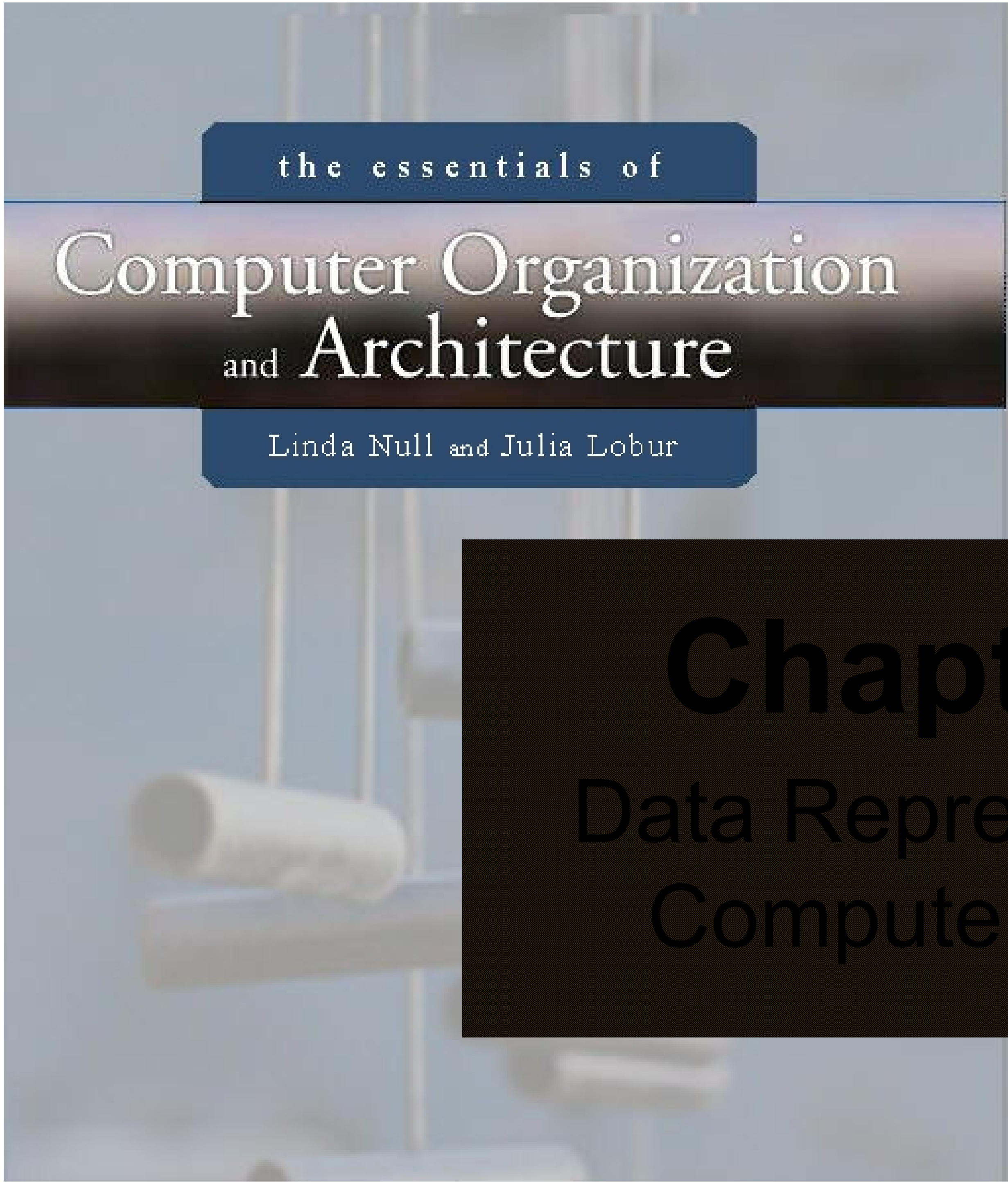


Chapter 2

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

Chapter 2 Objectives

- Gain familiarity with the most popular character codes.
- Become aware of the differences between how data is stored in computer memory, how it is transmitted over telecommunication lines, and how it is stored on disks.
- Understand the concepts of error detecting and correcting codes.

2.1 Introduction

- 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.

2.1 Introduction

- 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* (or *nybble*).
  - Bytes, therefore, consist of two nibbles: a “high-order nibble,” and a “low-order” nibble.

2.2 Positional Numbering Systems

- Bytes store numbers when the position of each bit represents 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*).

2.2 Positional Numbering Systems

- 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:

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

2.2 Positional Numbering Systems

- The binary number 11001 in powers of 2 is:

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

- 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:

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

2.3 Decimal to Binary Conversions

- 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.

2.3 Decimal to Binary Conversions

- In a previous slide, we said that every integer value can be represented exactly using any radix system.
- You can use either of 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.

2.3 Decimal to Binary Conversions

- **Suppose we want to convert the decimal number 190 to base 3.**
  – We know that $3^5 = 243$ so our result will be less than six digits wide. The largest power of 3 that we need is therefore $3^4 = 81$, and $81 \times 2 = 162$.
  – Write down the 2 and subtract 162 from 190, giving 28.

$$\begin{array}{r} 190 \\ -\ 162 \\ \hline 28 \end{array} \quad = 3^4 \times 2$$

2.3 Decimal to Binary Conversions

- **Converting 190 to base 3...**
  - The next power of 3 is $3^3 = 27$. We'll need one of these, so we subtract 27 and write down the numeral 1 in our result.
  - The next power of 3, $3^2 = 9$, is too large, but we have to assign a placeholder of zero and carry down the 1.

[redacted]

$$\begin{array}{rl} -\ \ 27 & = 3^3 \times 1 \\ \hline 1 & \\ -\ \ 0 & = 3^2 \times 0 \\ \hline 1 & \end{array}$$

2.3 Decimal to Binary Conversions

- **Converting 190 to base 3...**
  - $3^{1} = 3$ is again too large, so we assign a zero placeholder.
  - The last power of 3, $3^{0} = 1$, is our last choice, and it gives us a difference of zero.
  - Our result, reading from top to bottom is:

    $190_{10} = 21001_{3}$

[redacted] 2

1

0

$$
\begin{array}{rl}
-\quad 0 & = 3^{1} \times 0 \\
\hline
1 & \\
-\quad 1 & = 3^{0} \times 1 \\
\hline
0 &
\end{array}
$$

2.3 Decimal to Binary Conversions

- 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.

2.3 Decimal to Binary Conversions

- **Converting 190 to base 3...**
  - 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, 3 divides 190 63 times, with a remainder of 1.
  - Record the quotient and the remainder.

$$
\begin{array}{r|l l}
3 & 190 & 1 \\
\hline
 & \;\;63 &
\end{array}
$$

2.3 Decimal to Binary Conversions

- **Converting 190 to base 3...**
  - 63 is evenly divisible by 3.
  - Our remainder is zero, and the quotient is 21.

$$
\begin{array}{r|l l}
3 & 190 & 1 \\
\hline
3 & 63 & 0 \\
\hline
 & 21 &
\end{array}
$$

2.3 Decimal to Binary Conversions

- **Converting 190 to base 3...**
  - Continue in this way until the quotient is zero.
  - In the final calculation, we note that 3 divides 2 zero times with a remainder of 2.
  - Our result, reading from bottom to top is:

$$190_{10} = 21001_3$$

3	190	1
3	63	0
3	21	0
3	7	1
3	2	2
	0	

2.3 Decimal to Binary Conversions

- 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.

2.3 Decimal to Binary Conversions

- 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:

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

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

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

$$= 0.5 + 0.25 = 0.75$$

2.3 Decimal to Binary Conversions

- As with whole-number conversions, you can use either of two methods: a subtraction method and 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.

2.3 Decimal to Binary Conversions

- **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 <u>top to bottom</u> is:

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

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

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

2.3 Decimal to Binary Conversions

- **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.

$$
\begin{array}{r}
.8125 \\
\times \quad 2 \\
\hline
1.6250
\end{array}
$$

2.3 Decimal to Binary Conversions

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

$$
\begin{array}{r}
.8125 \\
\times \quad\ 2 \\
\hline
1.6250
\end{array}
$$

$$
\begin{array}{r}
.6250 \\
\times \quad\ 2 \\
\hline
1.2500
\end{array}
$$

$$
\begin{array}{r}
.2500 \\
\times \quad\ 2 \\
\hline
0.5000
\end{array}
$$

2.3 Decimal to Binary Conversions

- **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
```

2.3 Decimal to Binary Conversions

- 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.

2.3 Decimal to Binary Conversions

- The hexadecimal numbering system uses the numerals 0 through 9 and the letters A through F.
  - The decimal number 12 is B_{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

2.3 Decimal to Binary Conversions

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

0011	0101	0001	1011
3	5	1	B

- 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.

2.4 Signed Integer Representation

- The conversions we have so far presented have involved only positive numbers.
- To represent negative values, computer systems allocate the high-order bit to indicate the sign of a value.
  - The high-order bit is the leftmost bit in a byte. It is also called the most significant bit.
- The remaining bits contain the value of the number.

2.4 Signed Integer Representation

- There are three ways in which signed binary numbers may be expressed:
  - Signed magnitude,
  - One’s complement and
  - 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.

2.4 Signed Integer Representation

- For example, in 8-bit signed magnitude, positive 3 is: `00000011`
- Negative 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.

2.4 Signed Integer Representation

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

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

- The simplicity of this system makes it possible for digital circuits to carry out arithmetic operations.
  - We will describe these circuits in Chapter 3.

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

2.4 Signed Integer Representation

- 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
  ---------
```

2.4 Signed Integer Representation

- 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     1001011
0  +  0101110
   ----------
            1
```

2.4 Signed Integer Representation

- Example:
  – Using signed magnitude binary arithmetic, find the sum of 75 and 46.
- In the second bit, we have a carry, so we note it above the third bit.

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

2.4 Signed Integer Representation

- 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
```

2.4 Signed Integer Representation

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

```
            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.

2.4 Signed Integer Representation

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

```
1
          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
```

2.4 Signed Integer Representation

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

```
         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.

2.4 Signed Integer Representation

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

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

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

2.4 Signed Integer Representation

- 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) computers systems employ *complement systems* for numeric value representation.

2.4 Signed Integer Representation

- In complement systems, negative values are represented by some difference between a number and its base.
- In *diminished radix complement* systems, a negative value is given by the difference between the absolute value of a number and one less than its base.
- In the binary system, this gives us *one's complement*. It amounts to little more than flipping the bits of a binary number.

2.4 Signed Integer Representation

- For example, in 8-bit one's complement, positive 3 is: `00000011`
- Negative 3 is: `11111100`
  - In one's complement, 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 special circuitry for subtraction. The difference of two values is found by adding the minuend to the complement of the subtrahend.

2.4 Signed Integer Representation

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

```
1   1 1
    00110000
    11101100
    --------
    00011100
         + 1
    --------
    00011101
```

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

2.4 Signed Integer Representation

- 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.

2.4 Signed Integer Representation

- To express a value in two's complement:
  - 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 one's complement, positive 3 is: 00000011
  - Negative 3 in one's complement is: 11111100
  - Adding 1 gives us -3 in two's complement form: 11111101.

2.4 Signed Integer Representation

- 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.

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

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

2.4 Signed Integer Representation

- When we use any finite number of bits to represent a number, we always run the risk of the result of our calculations becoming too large 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.

2.4 Signed Integer Representation

- Example:
  - Using two's complement binary arithmetic, find the sum of 107 and 46.
- We see that the nonzero carry from the seventh bit *overflows* into the sign bit, giving us the erroneous result: 107 + 46 = -103.

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

<u>Rule for detecting two's complement overflow:</u> When the "carry in" and the "carry out" of the sign bit differ, overflow has occurred.

2.5 Floating-Point Representation

- The signed magnitude, one’s complement, and two’s complement representation that we have just presented deal with 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.

2.5 Floating-Point Representation

- 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.

2.5 Floating-Point Representation

- Numbers with fractions
- Could be done in pure binary
  - $1001.1010 = 2^4 + 2^0 + 2^{-1} + 2^{-3} = 9.625$
- Where is the binary point?
- $Q_{A.B}$ format to represent Numbers.
- Add, Sub, Multiply Operations with $Q_{A.B}$ format integer numbers

2.5 Floating-Point Representation

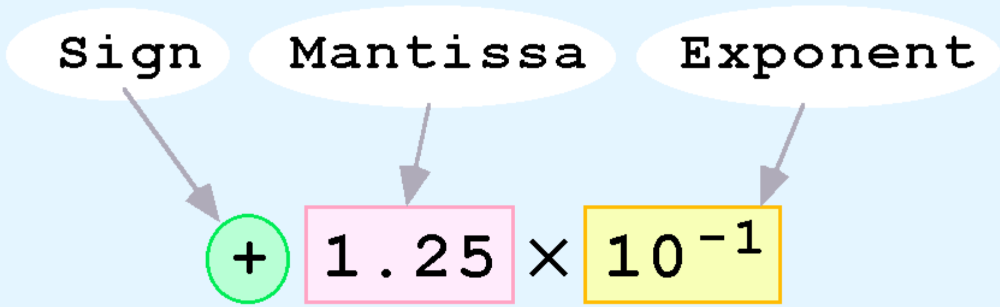
- Floating-point numbers allow an arbitrary number of decimal places to the right of the decimal point.
  - For example: $0.5 \times 0.25 = 0.125$
- They are often expressed in scientific notation.
  - For example:

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

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

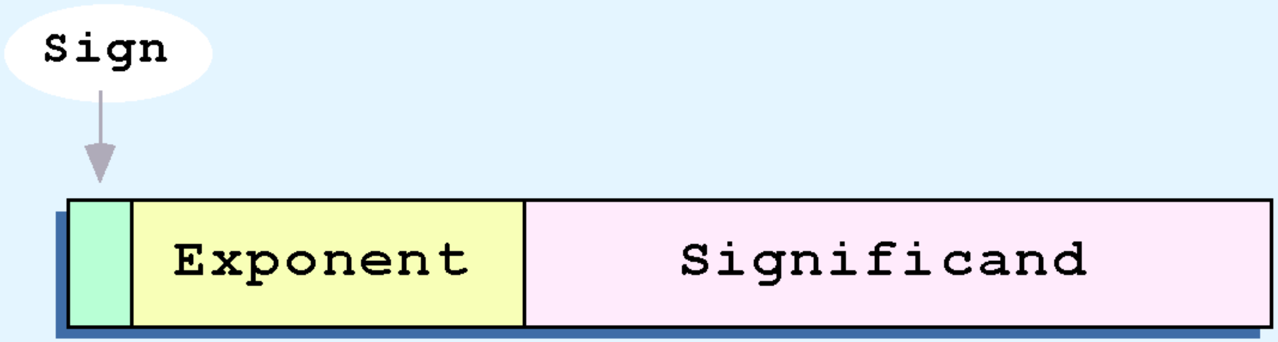
2.5 Floating-Point Representation

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



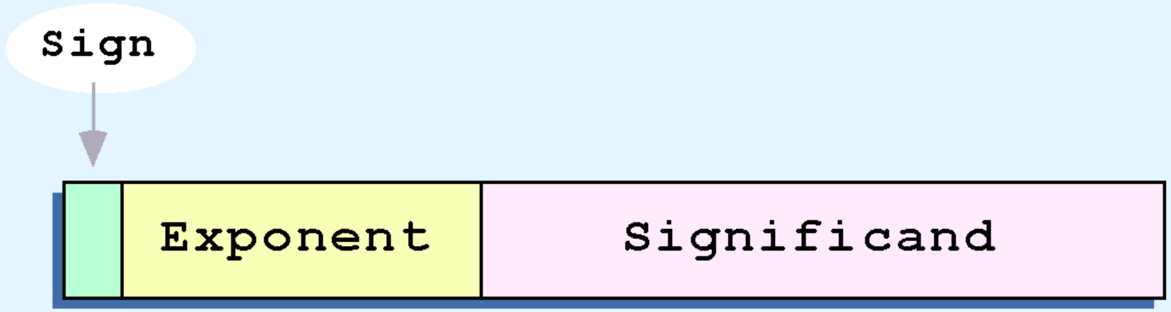
2.5 Floating-Point Representation

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



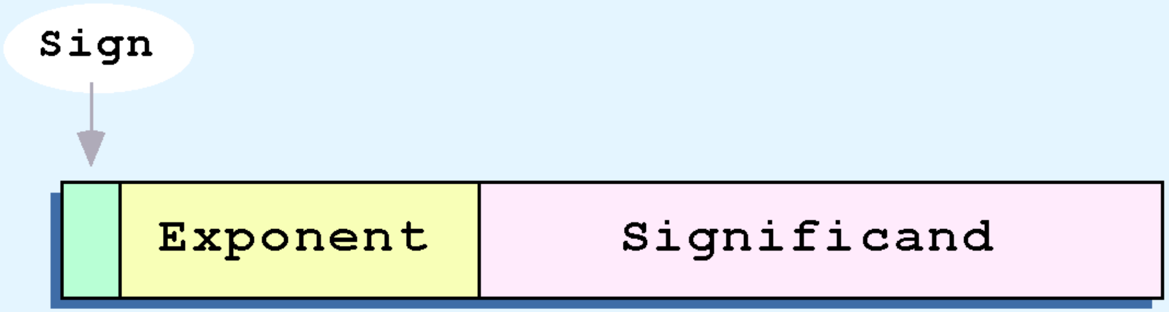
- This is the standard arrangement of these fields.

2.5 Floating-Point Representation



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

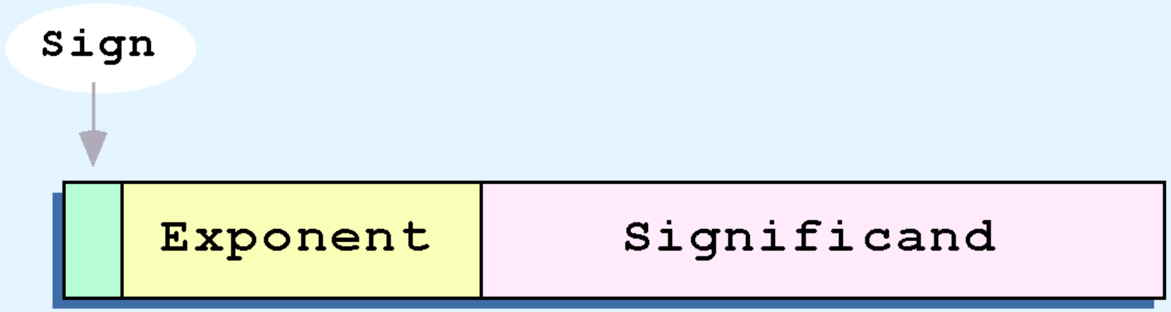
2.5 Floating-Point Representation



- The IEEE-754 *single precision* floating point standard uses an 8-bit exponent and a 23-bit significand.
- The IEEE-754 *double precision* standard uses an 11-bit exponent and a 52-bit significand.

For illustrative purposes, we will use a 14-bit model with a 5-bit exponent and an 8-bit significand.

2.5 Floating-Point Representation



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

2.5 Floating-Point Representation

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

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

- How about 17.25_{10} and 17.0625_{10}?

2.5 Floating-Point Representation

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

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

2.5 Floating-Point Representation

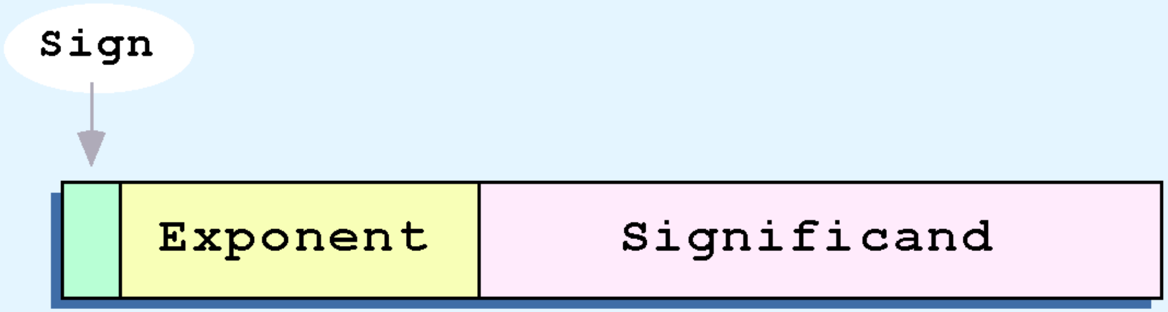
- Normalization
- FP numbers are usually normalized
- i.e. exponent is adjusted so that leading bit (MSB) of mantissa is 1
- Since it is always 1 there is no need to store it (consider the former examples again)
- (Scientific notation where numbers are normalized to give a single digit before the decimal point, e.g. 3.123×10^3)

2.5 Floating-Point Representation

- To resolve the problem of synonymous forms, we will establish a rule that the first digit of the significand must be 1. This results in a unique pattern for each floating-point number.
  - In the IEEE-754 standard, this 1 is implied meaning that a 1 is assumed after the binary point.
  - By using an implied 1, we increase the precision of the representation by a power of two. (Why?)

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

2.5 Floating-Point Representation



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

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

2.5 Floating-Point Representation

- To 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.

2.5 Floating-Point Representation

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

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

2.5 Floating-Point Representation

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

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

2.5 Floating-Point Representation

- Example:
  - Express -26.625_{10} in the revised 14-bit floating-point model.
- We find $26.625_{10} = 11010.101_2$. Normalizing, we have: $26.625_{10} = 0.11010101 \times 2^{5}$.
- To use our excess 16 biased exponent, we add 16 to 5, giving 21_{10} ($=10101_2$). We also need a 1 in the sign bit.

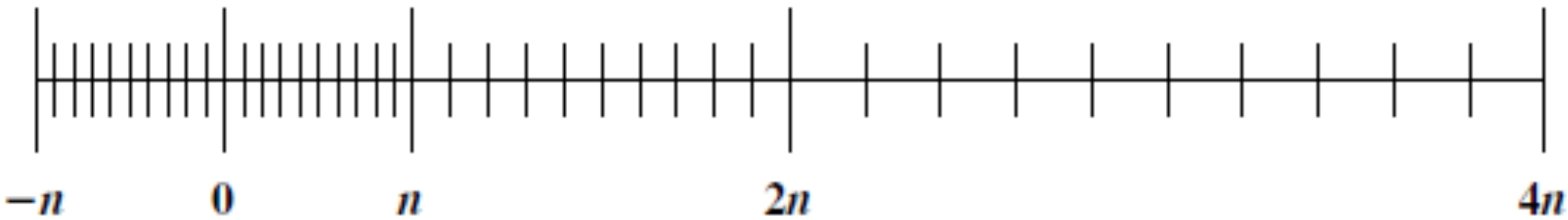
1	1 0 1 0 1	1 1 0 1 0 1 0 1

2.5 Floating-Point Representation

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

2.5 Floating-Point Representation

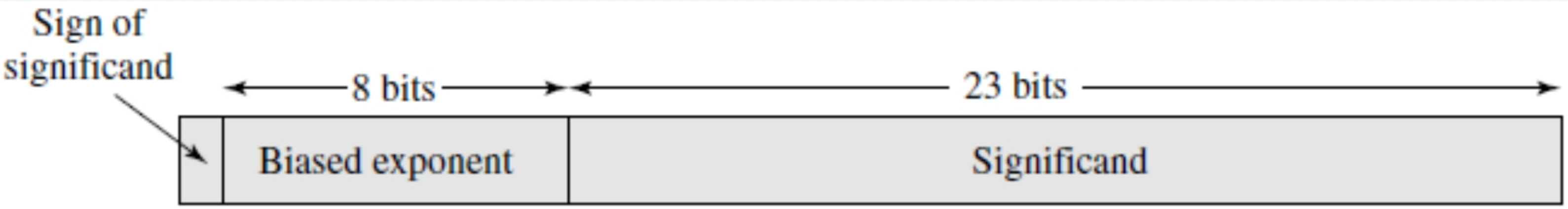
- numbers represented in floating-point notation are not spaced evenly along the number line, as are fixed-point numbers. The possible values get closer together near the origin and farther apart as you move away



2.5 Floating-Point Representation

- The IEEE-754 single precision floating point standard uses bias of 127 over its 8-bit exponent.
- The sign is stored in the first bit of the word.
- The first bit of the true significand is always 1 and need not be stored in the significand field.
- The value 127 is added to the true exponent to be stored in the exponent field.
- The base is 2.

2.5 Floating-Point Representation



(a) Format

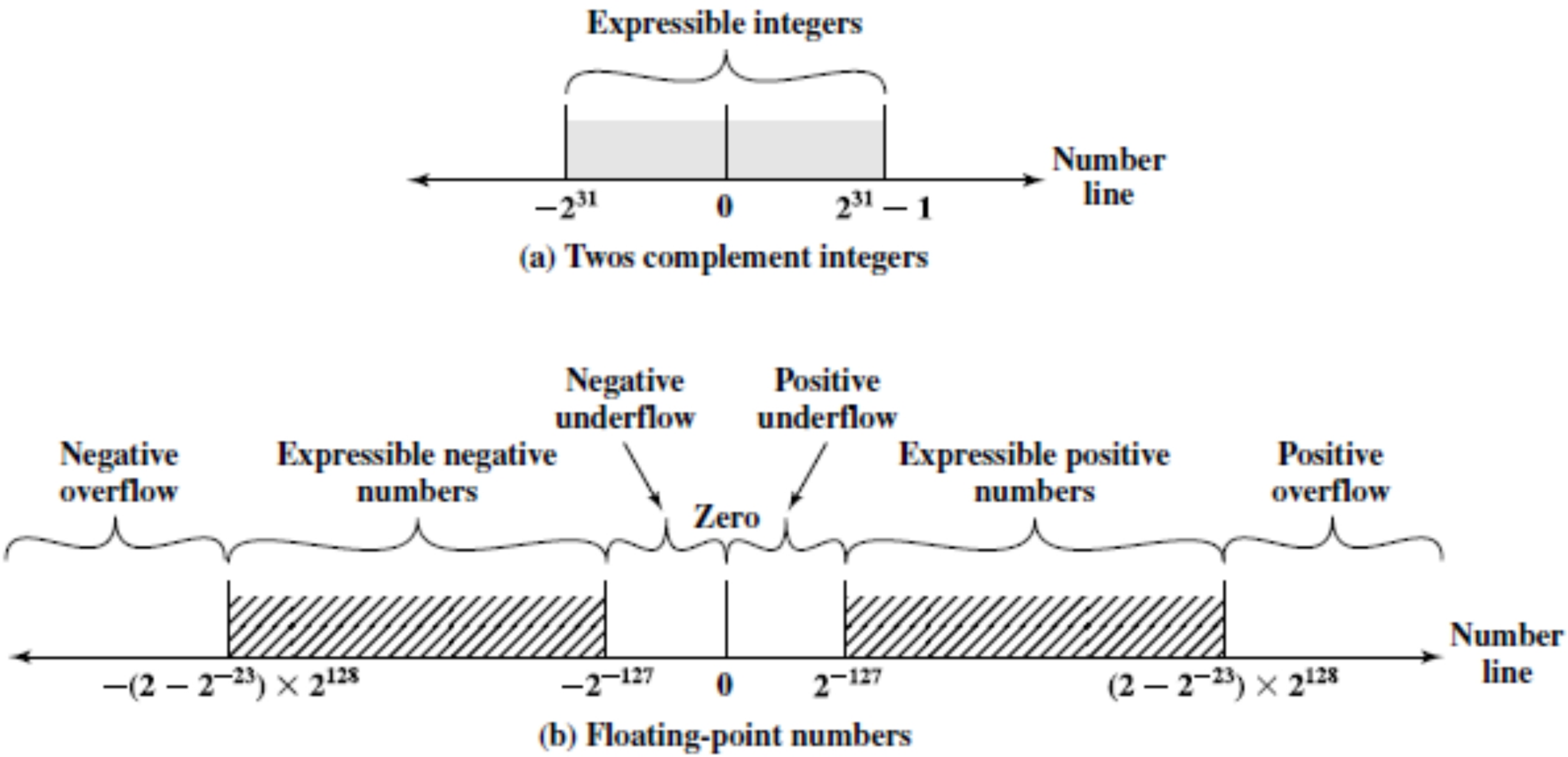
$$
\begin{array}{rcll}
1.1010001 \times 2^{10100} & = & 0\ 10010011\ 10100010000000000000000 & = 1.6328125 \times 2^{20} \\
-1.1010001 \times 2^{10100} & = & 1\ 10010011\ 10100010000000000000000 & = -1.6328125 \times 2^{20} \\
1.1010001 \times 2^{-10100} & = & 0\ 01101011\ 10100010000000000000000 & = 1.6328125 \times 2^{-20} \\
-1.1010001 \times 2^{-10100} & = & 1\ 01101011\ 10100010000000000000000 & = -1.6328125 \times 2^{-20}
\end{array}
$$

(b) Examples

2.5 Floating-Point Representation

- the range of numbers that can be represented in a 32-bit word.
- Using two's complement integer representation, all of the integers from -2^{31} to $2^{31}-1$ can be represented, for a total of 2^{32} different numbers
- With the example floating-point format above, the following ranges of numbers are possible:
  – Negative numbers between $-(2 - 2^{-23}) * 2^{128}$ and -2^{-127}
  – Positive numbers between 2^{-127} and $(2 - 2^{-23}) * 2^{128}$

2.5 Floating-Point Representation



2.5 Floating-Point Representation

- Five regions on the number line are not included in these ranges:
  - Negative numbers less than $-(2 - 2^{-23}) * 2^{128}$, called **negative overflow**
  - Negative numbers greater than 2^{-127} called **negative underflow**
  - Zero
  - Positive numbers less than 2^{-127} called **positive underflow**
  - Positive numbers greater than $(2 - 2^{-23}) * 2^{128}$ called **positive overflow**

The representation as presented will not accommodate a value of 0

2.5 Floating-Point Representation

- IEEE 754 Format Parameters

Parameter	Format			
	Single	Single Extended	Double	Double Extended
Word width (bits)	32	≥43	64	≥79
Exponent width (bits)	8	≥11	11	≥15
Exponent bias	127	unspecified	1023	unspecified
Maximum exponent	127	≥1023	1023	≥16383
Minimum exponent	−126	≤−1022	−1022	≤−16382
Number range (base 10)	$10^{-38}, 10^{+38}$	unspecified	$10^{-308}, 10^{+308}$	unspecified
Significand width (bits)*	23	≥31	52	≥63
Number of exponents	254	unspecified	2046	unspecified
Number of fractions	2^{23}	unspecified	2^{52}	unspecified
Number of values	1.98×2^{31}	unspecified	1.99×2^{63}	unspecified

2.5 Floating-Point Representation

- Interpretation of IEEE 754 Floating-Point Numbers

	Single Precision (32 bits)				Double Precision (64 bits)			
	Sign	**Biased exponent**	**Fraction**	**Value**	**Sign**	**Biased exponent**	**Fraction**	**Value**
positive zero	0	0	0	0	0	0	0	0
negative zero	1	0	0	-0	1	0	0	-0
plus infinity	0	255 (all 1s)	0	∞	0	2047 (all 1s)	0	∞
minus infinity	1	255 (all 1s)	0	$-\infty$	1	2047 (all 1s)	0	$-\infty$
quiet NaN	0 or 1	255 (all 1s)	$\neq 0$	NaN	0 or 1	2047 (all 1s)	$\neq 0$	NaN
signaling NaN	0 or 1	255 (all 1s)	$\neq 0$	NaN	0 or 1	2047 (all 1s)	$\neq 0$	NaN
positive normalized nonzero	0	$0 < e < 255$	f	$2^{e-127}(1.f)$	0	$0 < e < 2047$	f	$2^{e-1023}(1.f)$
negative normalized nonzero	1	$0 < e < 255$	f	$-2^{e-127}(1.f)$	1	$0 < e < 2047$	f	$-2^{e-1023}(1.f)$
positive denormalized	0	0	$f \neq 0$	$2^{e-126}(0.f)$	0	0	$f \neq 0$	$2^{e-1022}(0.f)$
negative denormalized	1	0	$f \neq 0$	$-2^{e-126}(0.f)$	1	0	$f \neq 0$	$-2^{e-1022}(0.f)$

2.5 Floating-Point Representation

- Floating-point addition and subtraction are done using methods analogous to how we perform calculations using pencil and paper.
- The first thing that we do is express both operands in the same exponential power, then add the numbers, preserving the exponent in the sum.
- If the exponent requires adjustment, we do so at the end of the calculation.

2.5 Floating-Point Representation

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

	0	1 0 1 0 0	1 1 0 0 0 0 0 0
+	0	1 0 1 0 0	0 0 0 1 0 1 0 0
	0	1 0 1 0 0	1 1 0 1 0 1 0 0

2.5 Floating-Point Representation

- Floating-point multiplication is also carried out in a manner akin to how we perform multiplication using pencil and paper.
- We multiply the two operands and add their exponents.
- If the exponent requires adjustment, we do so at the end of the calculation.

2.5 Floating-Point Representation

- Example:
  - Find the product of 12_{10} and 1.25_{10} using the 14-bit floating-point model.
- We find $12_{10} = 0.1100 \times 2^4$. And $1.25_{10} = 0.101 \times 2^1$.
- Thus, our product is $0.0111100 \times 2^5 = 0.1111 \times 2^4$.
- The normalized product requires an exponent of $20_{10} = 10110_2$.

	0	1 0 1 0 0	1 1 0 0 0 0 0 0
×	0	1 0 0 0 1	1 0 1 0 0 0 0 0
	0	1 0 1 0 1	0 1 1 1 1 0 0 0

2.5 Floating-Point Representation

- A floating-point operation may produce one of these conditions:
- **Exponent overflow:** A positive exponent exceeds the maximum possible exponent value. In some systems, this may be designated as +∞ or -∞
- **Exponent underflow:** A negative exponent is less than the minimum possible exponent value. This means that the number is too small to be represented, and it may be reported as 0

2.5 Floating-Point Representation

- **Significand underflow:** In the process of aligning significands, digits may flow off the right end of the significand.
- **Significand overflow:** The addition of two significands of the same sign may result in a carry out of the most significant bit.This can be fixed by realignment.

2.5 Floating-Point Representation

- Floating-Point Numbers and Arithmetic Operations

Floating Point Numbers	Arithmetic Operations
$X = X_S \times B^{X_E}$ $Y = Y_S \times B^{Y_E}$	$\left.\begin{array}{l} X + Y = (X_S \times B^{X_E - Y_E} + Y_S) \times B^{Y_E} \\ X - Y = (X_S \times B^{X_E - Y_E} - Y_S) \times B^{Y_E} \end{array}\right\} X_E \le Y_E$ $X \times Y = (X_S \times Y_S) \times B^{X_E + Y_E}$ $\frac{X}{Y} = \left(\frac{X_S}{Y_S}\right) \times B^{X_E - Y_E}$

Examples:

$$X = 0.3 \times 10^2 = 30$$
$$Y = 0.2 \times 10^3 = 200$$

$$X + Y = (0.3 \times 10^{2-3} + 0.2) \times 10^3 = 0.23 \times 10^3 = 230$$
$$X - Y = (0.3 \times 10^{2-3} - 0.2) \times 10^3 = (-0.17) \times 10^3 = -170$$
$$X \times Y = (0.3 \times 0.2) \times 10^{2+3} = 0.06 \times 10^5 = 6000$$
$$X \div Y = (0.3 \div 0.2) \times 10^{2-3} = 1.5 \times 10^{-1} = 0.15$$

2.5 Floating-Point Representation

- No matter how many bits we use in a floating-point representation, our model must be finite.
- The real number system is, of course, infinite, so our models can give nothing more than an approximation of a real value.
- At some point, every model breaks down, introducing errors into our calculations.
- By using a greater number of bits in our model, we can reduce these errors, but we can never totally eliminate them.

2.5 Floating-Point Representation

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

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

2.5 Floating-Point Representation

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

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

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

2.5 Floating-Point Representation

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

2.5 Floating-Point Representation

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

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

2.5 Floating-Point Representation

- Another Exm: 1.0 is NOT EQUAL to 10*0.1
- Floating Point arithmetic IS NOT associative
  - x + (y + z) is not necessarily equal to (x + y) + z
- Addition may not even result in a change
  - (x + 1) MAY == x

2.6 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.

2.6 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.

2.6 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.

2.6 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.

2.6 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

2.7 Codes for Data Recording And Transmission

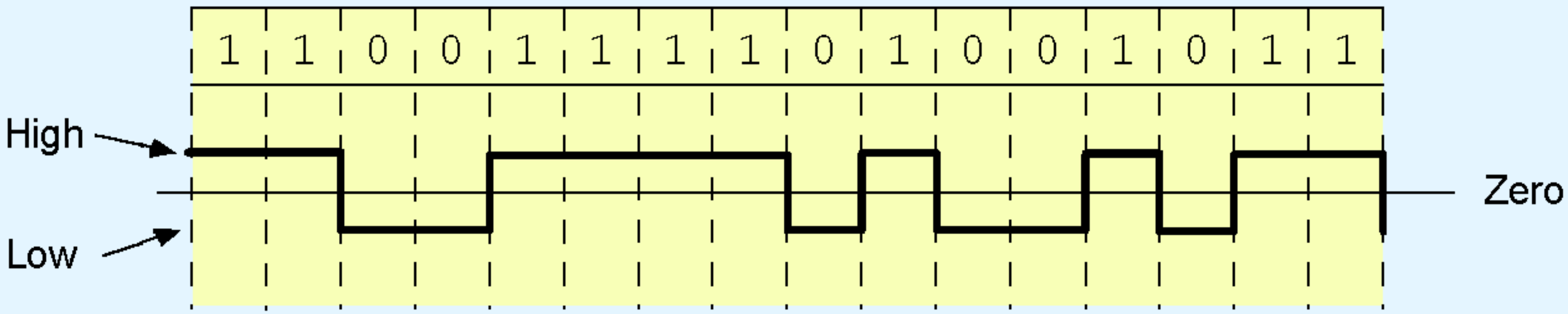
- When character codes or numeric values are stored in computer memory, their values are unambiguous.
- This is not always the case when data is stored on magnetic disk or transmitted over a distance of more than a few feet.
  - Owing to the physical irregularities of data storage and transmission media, bytes can become garbled.
- Data errors are reduced by use of suitable coding methods as well as through the use of various error-detection techniques.

2.7 Codes for Data Recording And Transmission

- To transmit data, pulses of “high” and “low” voltage are sent across communications media.
- To store data, changes are induced in the magnetic polarity of the recording medium.
  - These polarity changes are called *flux reversals(磁通量逆转).*
- The period of time during which a bit is transmitted, or the area of magnetic storage within which a bit is stored is called a *bit cell*.

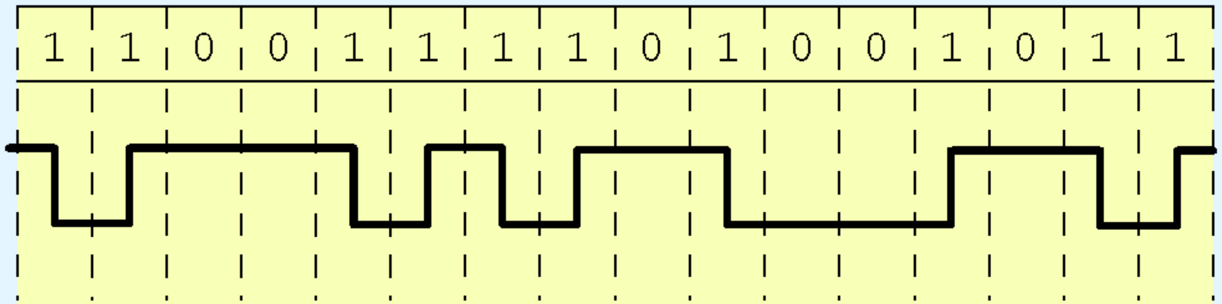
2.7 Codes for Data Recording And Transmission

- The simplest data recording and transmission code is the non-return-to-zero (NRZ) code.
- NRZ encodes 1 as “high” and 0 as “low.”
- The coding of *OK* (in ASCII) is shown below.



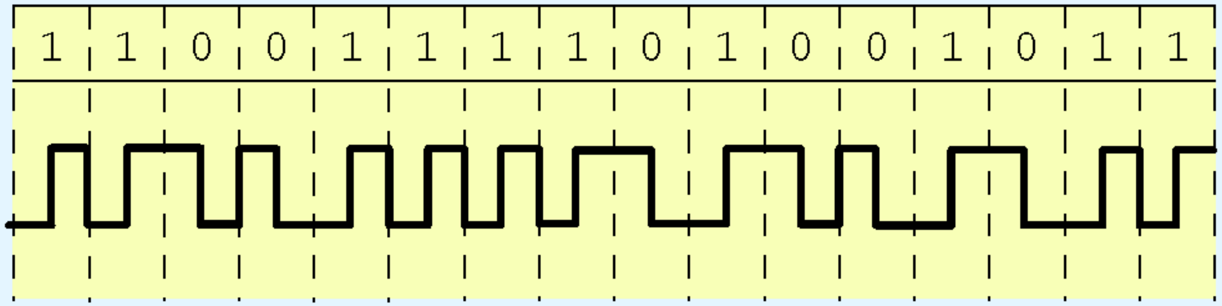
2.7 Codes for Data Recording And Transmission

- The problem with NRZ code is that long strings of zeros and ones cause synchronization loss.
- Non-return-to-zero-invert (NRZI) reduces this synchronization loss by providing a transition (either low-to-high or high-to-low) for each binary 1.



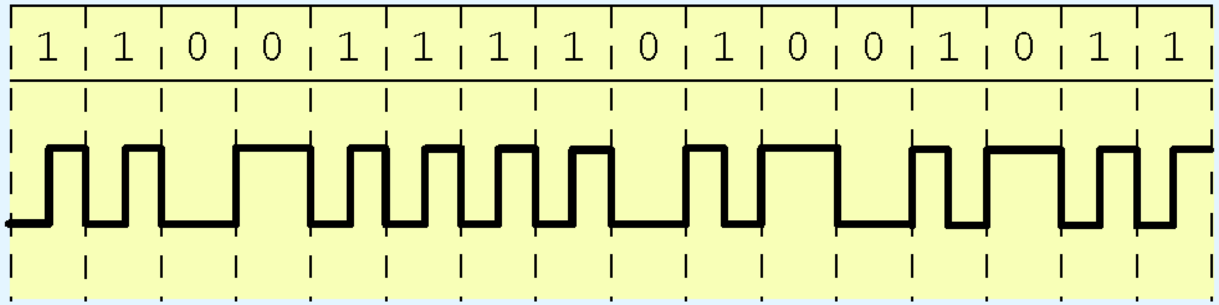
2.7 Codes for Data Recording And Transmission

- Although it prevents loss of synchronization over long strings of binary ones, NRZI coding does nothing to prevent synchronization loss within long strings of zeros.
- Manchester coding (also known as phase modulation) prevents this problem by encoding a binary one with an "up" transition and a binary zero with a "down" transition.



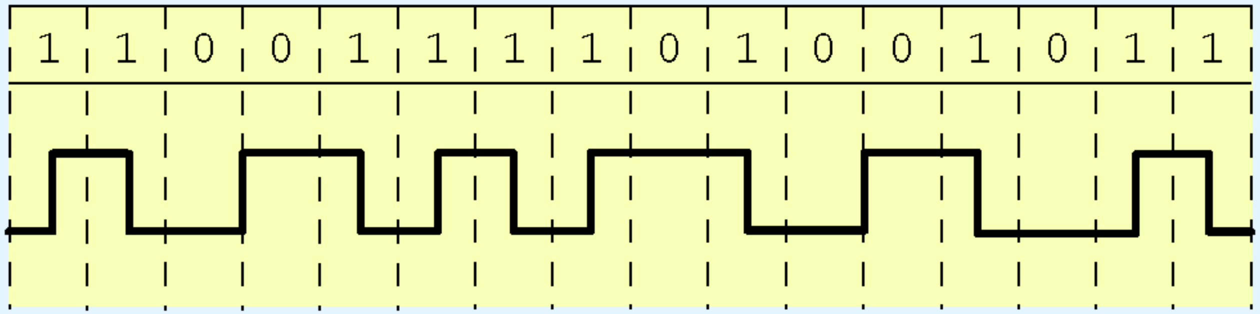
2.7 Codes for Data Recording And Transmission

- For many years, Manchester code was the dominant transmission code for local area networks.
- It is, however, wasteful of communications capacity because there is a transition on every bit cell.
- A more efficient coding method is based upon the frequency modulation (FM) code. In FM, a transition is provided at each cell boundary. Cells containing binary ones have a mid-cell transition.



2.7 Codes for Data Recording And Transmission

- At first glance, FM is worse than Manchester code, because it requires a transition at each cell boundary.
- If we can eliminate some of these transitions, we would have a more economical code.
- Modified FM does just this. It provides a cell boundary transition only when adjacent cells contain zeros.
- An MFM cell containing a binary one has a transition in the middle as in regular FM.



2.8 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.

2.8 Error Detection and Correction

- Check digits, appended to the end of a long number can provide some protection against data input errors.
  - The last character 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.

2.8 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.
- 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.

2.8 Error Detection and Correction

- 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.

2.8 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.

```
            1
     ________
1101)1111101
     1101
     ----
     0010
```

2.8 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.

```
           10
     _________
1101)1111101
     1101
     ----
     00101
```

2.8 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.

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

2.8 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.**

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

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

2.8 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
```

2.8 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.**
  - Some of the standard poly-nomials are listed in the text.

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

2.8 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.

2.8 Error Detection and Correction

- Hamming codes and Reed-Soloman codes are two important error correcting codes.
- Reed-Soloman 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-Soloman error correction.
- Because the mathematics of Hamming codes is much simpler than Reed-Soloman, we discuss Hamming codes in detail.

2.8 Error Detection and Correction

- 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.

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 is the smallest Hamming distance between *all* pairs of words in the code.

2.8 Error Detection and Correction

- 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) = $k + 1$.

2.8 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 2*k* + 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.

2.8 Error Detection and Correction

- Example: Assume a memory with 2 data bits and 1 parity that use even parity.

Data Word	Parity Bit	Code Word
00	0	000
01	1	011
10	1	101
11	0	110

2.8 Error Detection and Correction

- Question: Assume we wish to create a code using 3 information bits, 1 parity bit (appended to the end of the information), and odd parity. List all legal code words in this code. What is the Hamming distance of your code

2.8 Error Detection and Correction

- Example: Suppose we have the following code

0 0 0 0 0

0 1 0 1 1

1 0 1 1 0

1 1 1 0 1

What is the D(min)?

2.8 Error Detection and Correction

- Suppose we have a set of *n*-bit code words consisting of *m* data bits and *r* (redundant) parity bits.
- An error could occur in any of the *n* bits, so each code word can be associated with *n* erroneous words at a Hamming distance of 1.
- Therefore,we have *n* + 1 bit patterns for each code word: one valid code word, and *n* erroneous words.

2.8 Error Detection and Correction

- With n-bit code words, we have 2^n possible code words consisting of 2^m data bits (where $m = n + r$).
- This gives us the inequality:

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

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

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

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

2.8 Error Detection and Correction

- 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.
- 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.

2.8 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.
- 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?

2.8 Error Detection and Correction

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

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

  – 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.

2.8 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 an even 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

2.8 Error Detection and Correction

- Since 2 (= 2^1) contributes to the digits, 2, 3, 6, 7, 10, and 11. Position 2 will contain the parity for bits 3, 6, 7, 10, and 11.
- When we use even parity, this is the modulo 2 sum of the participating bit values.
- For the bit values shown, we have a parity value of 0 in the second bit position.

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

What are the values for the other parity bits?

2.8 Error Detection and Correction

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

- The completed code word is shown above.
  - Bit 1checks the digits, 3, 5, 7, 9, and 11, so its value is 1.
  - Bit 4 checks the digits, 5, 6, 7, and 12, so its value is 1.
  - Bit 8 checks the digits, 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!

2.8 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 digits, 3, 5, 7, 9, and 11. *Its value is 1, but should be zero.*
  - Bit 2 checks digits 2, 3, 6, 7, 10, and 11. The zero is correct.
  - Bit 4 checks digits, 5, 6, 7, and 12. *Its value is 1, but should be zero.*
  - Bit 8 checks digits, 9, 10, 11, and 12. This bit is correct.

2.8 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

- We have erroneous bits in positions 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.

2.5 Floating-Point Representation

- Homework
  - Programming to verify 1.0 is NOT EQUAL to 10*0.1
  - P89: 18, 20
  - P90: 23
  - P91: 34