CHAPTER 2
Data Representation in Computer Systems

2.1 Introduction 37
2.2 Positional Numbering Systems 38
2.3 Decimal to Binary Conversions 38
  2.3.1 Converting Unsigned Whole Numbers 39
  2.3.2 Converting Fractions 41
  2.3.3 Converting between Power-of-Two Radices 44
2.4 Signed Integer Representation 44
  2.4.1 Signed Magnitude 44
  2.4.2 Complement Systems 49
2.5 Floating-Point Representation 55
  2.5.1 A Simple Model 56
  2.5.2 Floating-Point Arithmetic 58
  2.5.3 Floating-Point Errors 59
  2.5.4 The IEEE-754 Floating-Point Standard 61
2.6 Character Codes 62
  2.6.1 Binary-Coded Decimal 62
  2.6.2 EBCDIC 63
  2.6.3 ASCII 63
  2.6.4 Unicode 65
2.7 Codes for Data Recording and Transmission 67
  2.7.1 Non-Return-to-Zero Code 68
  2.7.2 Non-Return-to-Zero-Invert Encoding 69
  2.7.3 Phase Modulation (Manchester Coding) 70
  2.7.4 Frequency Modulation 70
  2.7.5 Run-Length-Limited Code 71
2.8 Error Detection and Correction 73
  2.8.1 Cyclic Redundancy Check 73
  2.8.2 Hamming Codes 77
  2.8.3 Reed-Solomon 82
Chapter Summary 83
Further Reading 84
References 85
Review of Essential Terms and Concepts 85
Exercises 86
2.1 Introduction 37

- This chapter describes the various ways in which computers can store and manipulate numbers and characters.
- Bit: The most basic unit of information in a digital computer is called a **bit**, which is a contraction of binary digit.
- Byte: In 1964, the designers of the IBM System/360 main frame computer established a convention of using groups of 8 bits as the basic unit of **addressable** computer storage. They called this collection of 8 bits a **byte**.
- Word: Computer words consist of two or more adjacent bytes that are sometimes addressed and almost always are manipulated collectively. **Words** can be 16 bits, 32 bits, 64 bits.
- Nibbles: Eight-bit bytes can be divided into two 4-bit halves call **nibbles**.

2.2 Positional Numbering Systems 38

- Radix (or Base): The general idea behind positional numbering systems is that a numeric value is represented through increasing powers of a **radix** (or base).

<table>
<thead>
<tr>
<th>System</th>
<th>Radix</th>
<th>Allowable Digits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal</td>
<td>10</td>
<td>0,1,2,3,4,5,6,7,8,9</td>
</tr>
<tr>
<td>Binary</td>
<td>2</td>
<td>0,1</td>
</tr>
<tr>
<td>Octal</td>
<td>8</td>
<td>0,1,2,3,4,5,6,7</td>
</tr>
<tr>
<td>Hexadecimal</td>
<td>16</td>
<td>0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Powers of 2</th>
<th>Decimal</th>
<th>4-Bit Binary</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{-2} = \frac{1}{4} = 0.25$</td>
<td>0</td>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>$2^{-1} = \frac{1}{2} = 0.5$</td>
<td>1</td>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>$2^0 = 1$</td>
<td>2</td>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>$2^1 = 2$</td>
<td>3</td>
<td>0011</td>
<td>3</td>
</tr>
<tr>
<td>$2^2 = 4$</td>
<td>4</td>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>$2^3 = 8$</td>
<td>5</td>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>$2^4 = 16$</td>
<td>6</td>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>$2^5 = 32$</td>
<td>7</td>
<td>0111</td>
<td>7</td>
</tr>
<tr>
<td>$2^6 = 64$</td>
<td>8</td>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>$2^7 = 128$</td>
<td>9</td>
<td>1001</td>
<td>9</td>
</tr>
<tr>
<td>$2^8 = 256$</td>
<td>10</td>
<td>1010</td>
<td>A</td>
</tr>
<tr>
<td>$2^9 = 512$</td>
<td>11</td>
<td>1011</td>
<td>B</td>
</tr>
<tr>
<td>$2^{10} = 1,024$</td>
<td>12</td>
<td>1100</td>
<td>C</td>
</tr>
<tr>
<td>$2^{15} = 32,768$</td>
<td>13</td>
<td>1101</td>
<td>D</td>
</tr>
<tr>
<td>$2^{16} = 65,536$</td>
<td>14</td>
<td>1110</td>
<td>E</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>1111</td>
<td>F</td>
</tr>
</tbody>
</table>

**FIGURE 2.1** Some Number to Remember

**EXAMPLE 2.1** Three numbers represented as powers of a radix.

$243.51_{10} = 2 \times 10^2 + 4 \times 10^1 + 3 \times 10^0 + 5 \times 10^{-1} + 1 \times 10^{-2}$

$212_3 = 2 \times 3^2 + 1 \times 3^1 + 2 \times 3^0 = 23_{10}$

$10110_2 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 22_{10}$
2.3 Decimal to Binary Conversions 38

- There are two important groups of number base conversions:
  1. Conversion of decimal numbers to base-r numbers
  2. Conversion of base-r numbers to decimal numbers

2.3.1 Converting Unsigned Whole Numbers 39

- EXAMPLE 2.3 Convert $104_{10}$ to base 3 using the division-remainder method.
  $104_{10} = 10212_3$
- EXAMPLE 2.4 Convert $147_{10}$ to binary
  $147_{10} = 10010011_2$
- A binary number with N bits can represent unsigned integer from 0 to $2^n - 1$.
- Overflow: the result of an arithmetic operation is outside the range of allowable precision for the give number of bits.

2.3.2 Converting Fractions 41

- EXAMPLE 2.6 Convert $0.4304_{10}$ to base 5.
  $0.4304_{10} = 0.2034_5$
- EXAMPLE 2.7 Convert $0.34375_{10}$ to binary with 4 bits to the right of the binary point.
  Reading from top to bottom, $0.34375_{10} = 0.0101_2$ to four binary places. We simply discard (or truncate) our answer when the desired accuracy has been achieved.
- EXAMPLE 2.8 Convert $31214_4$ to base 3
  First, convert to decimal $31214_4 = 217_{10}$
  Then convert to base 3
  We have $31214_4 = 22001_3$

2.3.3 Converting between Power-of-Two Radices 44

- EXAMPLE 2.9 Convert $110010011101_2$ to octal and hexadecimal.
  $110010011101_2 = 6235_8$  Separate into groups of 3 for octal conversion
  $110010011101_2 = C9D_{16}$  Separate into groups of 4 for octal conversion

2.4 Signed Integer Representation 44

- By convention, a “1” in the high-order bit indicate a negative number.

2.4.1 Signed Magnitude 44

- A signed-magnitude number has a sign as its left-most bit (also referred to as the high-order bit or the most significant bit) while the remaining bits represent the magnitude (or absolute value) of the numeric value.
- N bits can represent $-(2^{n-1} - 1)$ to $2^{n-1} - 1$
- EXAMPLE 2.10 Add $01001111_2$ to $00100011_2$ using signed-magnitude arithmetic.
  $01001111_2 (79) + 00100011_2 (35) = 01110010_2 (114)$ There is no overflow in this example
- EXAMPLE 2.11 Add $01001111_2$ to $01100011_2$ using signed-magnitude arithmetic.
  An overflow condition and the carry is discarded, resulting in an incorrect sum.
  We obtain the erroneous result of $01001111_2 (79) + 01100011_2 (99) = 01100102 (50)$
- EXAMPLE 2.12 Subtract $01001111_2$ from $01100011_2$ using signed-magnitude arithmetic.
We find $01100001_2 (99) - 01001111_2 (79) = 00010100_2 (20)$ in signed-magnitude representation.

- EXAMPLE 2.14
- EXAMPLE 2.15
- The signed magnitude has two representations for zero, 10000000 and 00000000 (and mathematically speaking, the simple shouldn’t happen!).

2.4.2 Complement Systems 49

- One’s Complement
  - This sort of bit-flipping is very simple to implement in computer hardware.
  - EXAMPLE 2.16 Express $23_{10}$ and $-9_{10}$ in 8-bit binary one’s complement form.
    $23_{10} = + (00010111_2) = 00010111_2$
    $-9_{10} = - (00001001_2) = 11110110_2$
  - EXAMPLE 2.17
  - EXAMPLE 2.18
  - The primary disadvantage of one’s complement is that we still have two representations for zero: 00000000 and 11111111

- Two’s Complement
  - Find the one’s complement and add 1.
  - EXAMPLE 2.19 Express $23_{10}$, $-23_{10}$, and $-9_{10}$ in 8-bit binary two’s complement form.
    $23_{10} = + (00010111_2) = 00010111_2$
    $-23_{10} = - (00010111_2) = 11110110_2 + 1 = 11110101_2$
    $-9_{10} = - (00001001_2) = 11110110_2 + 1 = 11111110_2$
  - EXAMPLE 2.20
  - EXAMPLE 2.21
  - **A Simple Rule for Detecting an Overflow Condition:** If the carry in the sign bit equals the carry out of the bit, no overflow has occurred. If the carry into the sign bit is different from the carry out of the sign bit, overflow (and thus an error) has occurred.
  - EXAMPLE 2.22 Find the sum of $126_{10}$ and $8_{10}$ in binary using two’s complement arithmetic.
    A one is carried into the leftmost bit, but a zero is carried out. Because these carries are not equal, an overflow has occurred.
  - N bits can represent $-(2^{n-1})$ to $2^{n-1} - 1$. With signed-magnitude number, for example, 4 bits allow us to represent the value -7 through +7. However using two’s complement, we can represent the value -8 through +7.

- Integer Multiplication and Division
  - For each digit in the multiplier, the multiplicand is “shifted” one bit to the left. When the multiplier is 1, the “shifted” multiplicand is added to a running sum of partial products.
  - EXAMPLE Find the product of 00000110$_2$ and 00001011$_2$.
  - When the divisor is much smaller than the dividend, we get a condition known as divide underflow, which the computer sees as the equivalent of division by zero.
Computer makes a distinction between integer division and floating-point division.

- With integer division, the answer comes in two parts: a **quotient** and a **remainder**.
- Floating-point division results in a **number** that is expressed as a binary fraction.
- Floating-point calculations are carried out in dedicated circuits called floating-point units, or FPU.

### 2.5 Floating-Point Representation 55

- In scientific notation, numbers are expressed in two parts: a **fractional** part called a mantissa, and an **exponential** part that indicates the power of ten to which the mantissa should be raised to obtain the value we need.

#### 2.5.1 A Simple Model 56

- In digital computers, floating-point numbers consist of three parts: a **sign** bit, an **exponent** part (representing the exponent on a power of 2), and a fractional part called a significand (which is a fancy word for a mantissa).

```
<table>
<thead>
<tr>
<th>1 bit</th>
<th>5 bits</th>
<th>8 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sign bit</td>
<td>Exponent</td>
<td>Significand</td>
</tr>
</tbody>
</table>
```

FIGURE 2.2 Floating-Point Representation

- **Unbiased Exponent**
  - 17₁₀ = 0.10001₂ * 2⁵
  - 65536₁₀ = 0.1₂ * 2¹⁷

- **Biased Exponent**: We select 16 because it is midway between 0 and 31 (our exponent has 5 bits, thus allowing for 2⁵ or 32 values). Any number *larger* than 16 in the exponent field will represent a positive value. Value *less* than 16 will indicate negative values.
  - 17₁₀ = 0.10001₂ * 2⁵ The biased exponent is 16 + 5 = 21
  - 0.25₁₀ = 0.1₂ * 2⁻¹

- **EXAMPLE 2.23**

- A **normalized** form is used for storing a floating-point number in memory. A normalized form is a floating-point representation where the leftmost bit of the significand will always be a 1.

Example: Internal representation of (10.25)₁₀

\[
(10.25)_{10} = (1010.01)_{2} \quad \text{(Un-normalized form)} \\
= (1010.01)_{2} \times 2^0 \quad . \\
= (101.001)_{2} \times 2^1 \quad . \\
= (.101001)_{2} \times 2^4 \quad \text{(Normalized form)} \\
= (.0101001)_{2} \times 2^5 \quad \text{(Un-normalized form)} \\
= (.00101001)_{2} \times 2^6 \quad .
\]
Internal representation of \((10.25)_{10}\) is \(0.10100100\)

2.5.2 Floating-Point Arithmetic 58

- **EXAMPLE 2.24:** Renormalizing we retain the larger exponent and **truncate** the low-order bit.
- **EXAMPLE 2.25**

2.5.3 Floating-Point Errors 59

- We intuitively understand that we are working in the system of real number. We know that this system is **infinite**.
- Computers are **finite** systems, with **finite** storage. The more bits we use, the better the **approximation**. However, there is always some element of error, no matter how many bits we use.

2.5.4 The IEEE-754 Floating-Point Standard 61

- The IEEE-754 single precision floating point standard uses bias of 127 over its 8-bit exponent. An exponent of 255 indicates a special value.
- The double precision standard has a bias of 1023 over its 11-bit exponent. The “special” exponent value for a double precision number is 2047, instead of the 255 used by the single precision standard.

<table>
<thead>
<tr>
<th>Sign bit</th>
<th>Exponent</th>
<th>Significand</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0..0</td>
<td>0..0</td>
<td>Zero</td>
</tr>
<tr>
<td>x</td>
<td>0..0</td>
<td>not all zeros</td>
<td>Denormalized number</td>
</tr>
<tr>
<td>0</td>
<td>1..1</td>
<td>0..0</td>
<td>Plus infinity (+inf)</td>
</tr>
<tr>
<td>1</td>
<td>1..1</td>
<td>0..0</td>
<td>Minus infinity (-inf)</td>
</tr>
<tr>
<td>x</td>
<td>1..1</td>
<td>not all zeros</td>
<td>Not a Number (NaN)</td>
</tr>
</tbody>
</table>

Special bit patterns in IEEE-754

2.6 Character Codes 62

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

2.6.1 Binary-Coded Decimal 62

- Binary-coded Decimal (BCD) is a numeric coding system used primarily in IBM mainframe and midrange systems.
- When stored in an 8-bit byte, the upper nibble is called the **zone** and the lower part is called the **digit**.
- **EXAMPLE 2.26** Represent -1265 in 3 bytes using packed BCD. 
  The zoned-decimal coding for 1265 is: 
  1111 0001 1111 0010 1111 0110 1111 0101
  After packing, this string becomes:
0001 0010 0110 0101
Adding the sign after the low-order digit and padding in high order bit should be 0000 for a result of:
0000 0001 0010 0110 0101 1101

<table>
<thead>
<tr>
<th>Digit</th>
<th>BCD</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
</tr>
</tbody>
</table>

Zones

<table>
<thead>
<tr>
<th>Zone</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>Unsigned</td>
</tr>
<tr>
<td>1100</td>
<td>Positive</td>
</tr>
<tr>
<td>1101</td>
<td>Negative</td>
</tr>
</tbody>
</table>

FIGURE 2.5 Binary-Coded Decimal

2.6.2 EBCDIC 63
- EBCDIC (Extended Binary Coded Decimal Interchange Code) expand BCD from 6 bits to 8 bits. See Page 64 FIGURE 2.6.

2.6.3 ASCII 63
- ASCII: American Standard Code for Information Interchange
- In 1967, a derivative of this alphabet became the official standard that we now call ASCII.

2.6.4 Unicode 65
- Both EBCDIC and ASCII were built around the Latin alphabet.
- In 1991, a new international information exchange code called Unicode.
- Unicode is a 16-bit alphabet that is downward compatible with ASCII and Latin-1 character 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.
- Unicode is currently the default character set of the Java programming language.
2.7 Codes for Data Recording and Transmission

2.7.1 Non-Return-to-Zero Code

- 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.

![NRZ Encoding of OK](image)

2.7.2 Non-Return-to-Zero-Invert Encoding

- 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.

![NRZI Encoding OK](image)

2.7.3 Phase Modulation (Manchester Coding)

- 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.

![Phase Modulation (Manchester Coding) of the Word OK](image)
2.7.4 Frequency Modulation

- 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.

![Figure 2.12 Frequency Modulation Coding of OK](image)

- 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.

![Figure 2.13 Modified Frequency Modulation Coding of OK](image)

2.7.5 Run-Length-Limited Code

- The main challenge for data recording and transmission is how to retain synchronization without chewing up more resources than necessary.
- Run-length-limited, RLL, is a code specifically designed to reduce the number of consecutive ones and zeros.
- Some extra bits are inserted into the code.
- But even with these extra bits RLL is remarkably efficient.
- An RLL(d,k) code dictates a minimum of d and a maximum of k consecutive zeros between any pair of consecutive ones.
- RLL(2,7) has been the dominant disk storage coding method for many years.
- An RLL(2,7) code contains more bit cells than its corresponding ASCII or EBCDIC character.
- However, the coding method allows bit cells to be smaller, thus closer together, than in MFM or any other code.
• The RLL(2,7) coding for OK is shown below, compared to MFM. The RLL code (bottom) contains 25% fewer transitions than the MFM code (top).

![Figure 2.16 MFM (top) and RLL(2, 7) Coding (bottom) for OK](image)

• If the limiting factor in the design of a disk is the number of flux transitions per square millimeter, we can pack 50% more OKs in the same magnetic area using RLL than we could using MFM.
• RLL is used almost exclusively in the manufacture of high-capacity disk drive.

2.8 Error Detection and Correction 73
• No communications channel or storage medium can be completely error-free.

2.8.1 Cyclic Redundancy Check 73
• 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.
• Arithmetic Modulo 2
  The addition rules are as follows:
  0 + 0 = 0
  0 + 1 = 1
  1 + 0 = 1
  1 + 1 = 0
• EXAMPLE 2.27 Find the sum of 1011₂ and 110₂ modulo 2.
  \[1011₂ + 110₂ = 1101₂ (\text{mod } 2)\]
• EXAMPLE 2.28 Find the quotient and remainder when 100101₁₂ is divided by 101₁₂.
  Quotient 1010₁₂ and Remainder 10₁₂.
• Calculating and Using CRC
  o Suppose we want to transmit the information string: 100101₁₂.
  o The receiver and sender decide to use the (arbitrary) polynomial pattern, 1101.
  o The information string is shifted left by one position less than the number of positions in the divisor. \(I = 1001011000₂\)
  o The remainder is found through modulo 2 division (at right) and added to the information string: \(1001011000₂ + 100₂ = 1001011000100₂\).
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.

- A remainder other than zero indicates that an error has occurred in the transmission.
- This method works best when a large prime polynomial is used.
- There are four standard polynomials used widely for this purpose:
  - CRC-CCITT (ITU-T): \(X^{16} + X^{12} + X^5 + 1\)
  - CRC-12: \(X^{12} + X^{11} + X^3 + X^2 + X + 1\)
  - CRC-16 (ANSI): \(X^{16} + X^{15} + X^2 + 1\)
  - CRC-32: \(X^{32} + X^{26} + X^{23} + X^{22} + X^{16} + X^{12} + X^{11} + X^{10} + X^8 + X^7 + X^6 + X^4 + X + 1\)
- CRC-32 has been proven that CRCs using these polynomials can detect over 99.8% of all single-bit errors.

2.8.2 Hamming Codes

- Data communications channels are simultaneously more error-prone and more tolerant of errors than disk systems.
- Hamming code use parity bits, also called check bits or redundant bits.
- The final word, called a code word is an n-bit unit containing m data bits and r check bits.
  \(n = m + r\)
- The Hamming distance between two code words is the number of bits in which two code words differ.
  \(10001001\)
  \(10110001\)
  *** Hamming distance of these two code words is 3
- The minimum Hamming distance, \(D(\text{min})\), for a code is the smallest Hamming distance between all pairs of words in the code.
- Hamming codes can detect \(D(\text{min}) - 1\) errors and correct \([D(\text{min}) - 1 / 2]\) errors.
- EXAMPLE 2.29
- EXAMPLE 2.30
  \(00000\)
  \(01011\)
  \(10110\)
  \(11101\)
  \(D(\text{min}) = 3.\) Thus, this code can detect up to two errors and correct one single bit error.

- We are focused on single bit error. 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. 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:
\[
(m + r + 1) \cdot 2^m \leq 2^{m+r} \quad \text{or} \quad (m + r + 1) \leq 2^r
\]

- **EXAMPLE 2.31** Using the Hamming code just described and even parity, encode the 8-bit ASCII character K. (The high-order bit will be zero.) Induce a single-bit error and then indicate how to locate the error.

\( m = 8 \), we have \((8 + r + 1) \leq 2^r\) then we choose \( r = 4 \)

Parity bit at 1, 2, 4, 8
Char K \( 75_{10} = 01001011_2 \)

\[
\begin{align*}
1 &= 1 \\
2 &= 2 \\
3 &= 1 + 2 \\
4 &= 4 \\
5 &= 1 + 4 \\
6 &= 2 + 4 \\
7 &= 1 + 2 + 4 \\
8 &= 8 \\
9 &= 1 + 8 \\
10 &= 2 + 8 \\
11 &= 1 + 2 + 8 \\
12 &= 4 + 8
\end{align*}
\]

We have the following code word as a result:

\[
\begin{array}{cccccccccc}
0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\
12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1
\end{array}
\]

Parity b1 = \( b3 + b5 + b7 + b9 + b11 \) = 1 + 1 + 1 + 0 + 1 = 0

Parity b2 = \( b3 + b6 + b7 + b10 + b11 \) = 1 + 0 + 1 + 0 + 1 = 1

Parity b4 = \( b5 + b6 + b7 \) = 1 + 0 + 1 = 0

Parity b8 = \( b9 + b10 + b11 + b12 \) = 0 + 0 + 1 + 0 = 1

Let’s introduce an error in bit position b9, resulting in the code word:

\[
\begin{array}{cccccccccc}
0 & 1 & 0 & \textbf{1} & 1 & 0 & 1 & 0 & 1 & 0 \\
12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1
\end{array}
\]

Parity b1 = \( b3 + b5 + b7 + b9 + b11 \) = 1 + 1 + 1 + 1 + 1 = 1 (Error, should be 0)

Parity b2 = \( b3 + b6 + b7 + b10 + b11 \) = 1 + 0 + 1 + 0 + 1 = 1 (OK)

Parity b4 = \( b5 + b6 + b7 \) = 1 + 0 + 1 = 0 (OK)

Parity b8 = \( b9 + b10 + b11 + b12 \) = 1 + 0 + 1 + 0 = 0 (Error, should be 1)

We found that parity bits 1 and 8 produced an error, and 1 + 8 = 9, which in exactly where the error occurred.

2.8.3 Reed-Soloman 82

- If we expect errors to occur in blocks, it stands to reason that we should use an error-correcting code that operates at a block level, as opposed to a Hamming code, which operates at the bit level.

- A Reed-Soloman (RS) code can be thought of as a CRC that operates over entire characters instead of only a few bits.

- RS codes, like CRCs, are systematic: The parity bytes are append to a block of information bytes.

- RS \((n, k)\) code are defined using the following parameters:
  - \( s \) = The number of bits in a character (or “symbol”).
  - \( k \) = The number of \( s \)-bit characters comprising the data block.
- $n$ = The number of bits in the code word.
- RS $(n, k)$ can correct $(n-k)/2$ errors in the $k$ information bytes.
- Reed-Soloman error-correction algorithms lend themselves well to implementation in computer hardware.
- They are implemented in high-performance disk drives for mainframe computers as well as compact disks used for music and data storage. These implementations will be described in Chapter 7.

**Chapter Summary 83**
- 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 (or nybbles).
- 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.
- Character data is stored using ASCII, EBCDIC, or Unicode.
- Data transmission and storage codes are devised to convey or store bytes reliably and economically.
- Error detecting and correcting codes are necessary because we can expect no transmission or storage medium to be perfect.
- CRC, Reed-Soloman, and Hamming codes are three important error control codes.