CHAPTER 2

Data Representation in Computer Systems

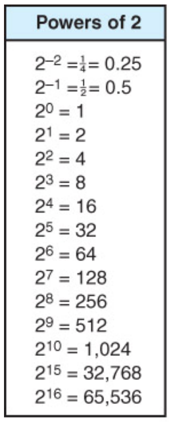
2.1 Introduction 47

- 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. The word size represents the data size that is handled **most efficiently** by a particular architecture. **Words** can be 16 bits, 32 bits, 64 bits.
- Nibbles: Eight-bit bytes can be divided into two 4-bit halves call **nibbles**.

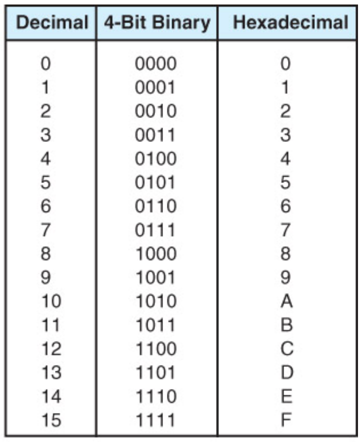
2.2 Positional Numbering Systems 48

- 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).

System	Radix	Allowable Digits
Decimal	10	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Binary	2	0, 1
Octal	8	0, 1, 2, 3, 4, 5, 6, 7
Hexadecimal	16	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F



Powers of 2
$2^{-2} = \frac{1}{4} = 0.25$
$2^{-1} = \frac{1}{2} = 0.5$
$2^0 = 1$
$2^1 = 2$
$2^2 = 4$
$2^3 = 8$
$2^4 = 16$
$2^5 = 32$
$2^6 = 64$
$2^7 = 128$
$2^8 = 256$
$2^9 = 512$
$2^{10} = 1,024$
$2^{15} = 32,768$
$2^{16} = 65,536$



Decimal	4-Bit Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

TABLE 2.1 Some Number to Remember

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

$243.51_{10} = 2 * 10^2 + 4 * 10^1 + 3 * 10^0 + 5 * 10^{-1} + 1 * 10^{-2}$
$212_3 = 2 * 3^2 + 1 * 3^1 + 2 * 3^0 = 23_{10}$

$10110_2 \quad = 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0 = 22_{10}$

2.3 Converting Between Bases 48

- 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 49

- EXAMPLE 2.3 Convert 104_{10} to base 3 using the division-remainder method.

  104_{10} = **10212_3**

```
3|104    2
3| 34    1
3| 11    2
 3| 3    0
  3|1    1
    0
```

- EXAMPLE 2.4 Convert 147_{10} to binary

  147_{10} = **10010011_2**

```
2|147    1
2| 73    1
 2|36    0
 2|18    0
  2|9    1
  2|4    0
  2|2    0
  2|1    1
    0
```

- 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 51

- 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.

  $0.34375_{10} = \mathbf{0.0101_2}$

```
 0.34375
X      2
 0.68750
X      2
 1.37500
X      2
 0.75000
X      2
 1.50000
```

- EXAMPLE 2.8 Convert 3121_4 to base 3

  First, convert to decimal $3121_4 = 217_{10}$
  Then convert to base 3 $217_{10} = 22001_3$
  We have $3121_4 = 22001_3$

2.3.3 Converting between Power-of-Two Radices 54

- EXAMPLE 2.9 Convert 110010011101_2 to octal and hexadecimal.

  $\underline{110}\ \underline{010}\ \underline{011}\ \underline{101}_2 = 6235_8$ Separate into groups of **3** for octal conversion

  $\underline{1100}\ \underline{1001}\ \underline{1101}_2 = C9D_{16}$ Separate into groups of **4** for octal conversion

2.4 Signed Integer Representation 54

- By convention, a "1" in the **high-order bit** indicate a negative number.

2.4.1 Signed Magnitude 54

- 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) = 0110010_2 (50)

- EXAMPLE 2.12 Subtract 01001111_2 from 01100011_2 using signed-magnitude arithmetic.

  We find 011000011_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 60

- **One's Complement**
  - This sort of bit-flipping is very simple to implement in computer hardware.
  - EXAMPLE 2.16 Express 2310 and -910 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.

    $23_{10} = + (00010111_2) = 00010111_2$
    $-23_{10} = - (00010111_2) = 11101000_2 + 1 = 11101001_2$
    $-9_{10} = - (00001001_2) = 11110110_2 + 1 = 11110111_2$

  - EXAMPLE 2.20 Add 9_{10} to -23_{10} using two's complement arithmetic.

    $00001001_2\ (9_{10}) + 11101001_2\ (-23_{10}) = 11110010_2\ (-14_{10})$

```
 00001001    <= Carries
  00001001₂   (  9)
+ 11101001₂  +(-23)
  11110010₂   (-14)
```

  - EXAMPLE 2.21 Find the sum of 23_{10} and -9_{10} in binary using two's complement arithmetic.

    $00010111_2\ (23_{10}) + 11110111_2\ (-9_{10}) = 00001110_2\ (14_{10})$

```
 11110111    <= Carries
  00010111₂   ( 23)
+ 11110111₂  +(- 9)
  00001110₂   ( 14)
```

  - **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, over (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.

    $01111110_2\ (126_{10}) + 00001000_2\ (8_{10}) = 10000110_2\ (-122_{10})$

```
 01111000    <= Carries
  01111110₂   ( 126)
+ 00001000₂  +(   8)
  10000110₂   (-122)
```

    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 (6_{10}) and 00001011_2 (11_{10}).

```
  00000110  ( 6)
x 00001011  (11)

  Multiplicand    Partial Products
  00000110     +  00000000 (1; add multiplicand and shift left)
  00001100     +  00000110 (1; add multiplicand and shift left)
  00011000     +  00010010 (0; Don't add, just shift multiplicand left)
  00110000     +  00010010 (1; add multiplicand and shift left)
               =  01000010 (Product; 6 X 11 = 66)
```

  - 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 call floating-point units, or FPU.

2.4.3 Unsigned Versus Signed Numbers 66

- If the 4-bit binary value 1101 is unsigned, then it represents the decimal value 13, but as a signed two's complement number, it represents -3.
- C programming language has **int** and **unsigned int** as possible types for integer variables.
- If we are using 4-bit unsigned binary numbers and we add 1 to 1111, we get 0000 ("return to zero").
- If we add 1 to the largest positive 4-bit two's complement number 0111 (+7), we get 1000 (-8).

2.4.4 Computers, Arithmetic, and Booth's Algorithm 66

- Consider the following standard pencil and paper method for multiplying two's complement numbers (-5 X -4):

```
      1011   (-5)
    x 1100   (-4)
    + 0000   (0 in multiplier means simple shift)
   + 0000    (0 in multiplier means simple shift)
  + 1011     (1 in multiplier means add multiplicand and shift)
 + 1011____  (1 in multiplier means add multiplicand and shift)
  10000100   (-4 X -5 = -124)
```

  Note that: "Regular" multiplication clearly yields the **incorrect** result.

- Research into finding better arithmetic algorithms has continued apace for over 50 years. One of the many interesting products of this work is Booth's algorithm.
- In most cases, Booth's algorithm carries out multiplication faster and more accurately than naïve pencil-and-paper methods.
- The general idea of Booth's algorithm is to increase the speed of a multiplication when there are consecutive zeros or ones in the multiplier.
- Consider the following standard multiplication example (3 X 6):

```
      0011   (3)
    x 0110   (6)
    + 0000   (0 in multiplier means simple shift)
   + 0011    (1 in multiplier means add multiplicand and shift)
  + 0011     (1 in multiplier means add multiplicand and shift)
 + 0000____  (0 in multiplier means simple shift)
   0010010   (3 X 6 = 18)
```

- In Booth's algorithm, if the multiplicand and multiplier are **n-bit** two's complement numbers, the result is a **2n-bit** two's complement value. Therefore, when we perform our intermediate steps, we must **extend** our n-bit numbers to 2n-bit numbers. For example, the 4-bit number 1000 (-8) extended to 8 bits would be 11111000.
- Booth's algorithm is interested in pairs of bits in the multiplier and proceed according to the following rules:
  - If the current multiplier bit is 1 and the preceding bit was, we are at the beginning of a string of ones, so **subtract (10 pair)** the multiplicand form the product.
  - If the current multiplier bit is 0 and the preceding bit was 1, we are at the end of a string of ones, so we **add (01 pair)** the multiplicand to the product.
  - If it is a **00** pair, or a **11** pair, do no arithmetic operation (we are in the middle of a string of zeros or a string of ones). Simply **shift**. The power of the algorithm is in this step: we can now treat a string of ones as a string of zeros and do nothing more than shift.

```
         0011   (3)
       x 0110   (6)
   + 00000000   (00 = simple shift; assume a mythical 0 as the previous bit)
  + 11111101    (10 = subtract = add 1111 1101, extend sign)
 + 00000000     (11 simple shift)
+ 00000011____  (01 = add )
  01000010010   (3 X 6 = 18; 010 ignore extended sign bit that go beyond 2n)
```

Note that: **010** Ignore extended sign bit that go beyond 2n.

- EXAMPLE 2.23 (-3 X 5) Negative 3 in 4-bit two's complement is 1101. Extended to 8 bits, it is 11111101. Its complement is 00000011. When we see the rightmost 1 in the multiplier, it is the beginning of a string of 1s, so we treat it as if it were the string 10:

```
         1101   (-3; for subtracting, we will add -3's complement, or 00000011)
       x 0101   (5)
   + 00000011   (10 = subtract 1101 = add 0000 0011)
  + 11111101    (01 = add 1111 1101 to product: note sign extension)
 + 00000011     (10 = subtract 1101 = add 0000 0011)
+ 11111101____  (01 = add 1111 1101 to product)
 100111110001   (-3 X 5 = -15; using the 8 rightmost bits, 11110001 or -15)
```

Note that: Ignore extended sign bit that go beyond 2n.

- EXAMPLE 2.24 Let’s look at the larger example of 53 X 126:

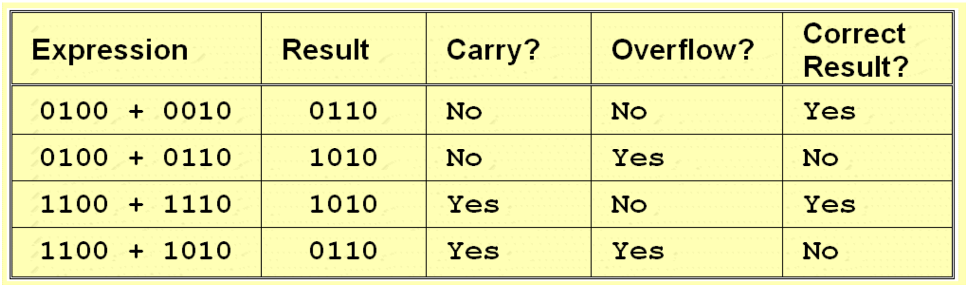
```
         00110101  (53; for subtracting, we will add the complement of 53 or 11001011)
       x 01111110  (126)
+0000000000000000  (00 = simple shift)
+111111111001011   (10 = subtract = add 11001011, sign extension)
+00000000000000    (11 = simple shift)
+0000000000000     (11 = simple shift)
+000000000000      (11 = simple shift)
+00000000000       (11 = simple shift)
+0000000000        (11 = simple shift)
+000110101____     (01 = add 00110101, sign extension)
10001101000010110  (53 X 126 = 6678; using the 16 rightmost bits)
```

Note that: Ignore extended sign bit that go beyond 2n.

- Booth’s algorithm not only allows multiplication to be performed **faster** in most cases, but it also has the added bonus in that it works correctly on **signed** numbers.

2.4.5 Carry Versus Overflow 70

- For **unsigned** numbers, a **carry** (out of the leftmost bit) indicates the total number of bits was not large enough to hold the resulting value, and overflow has occurred.
- For **signed** numbers, if the carry in to the sign bit and the carry (out of the sign bit) **differ**, then overflow has occurred.



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

TABLE 2.2 Examples of Carry and Overflow in Signed Numbers

2.4.6 Binary Multiplication and Division Using Shifting 71

- We can do binary multiplication and division by 2 very easily using an arithmetic shift operation
- A **left** arithmetic shift inserts a 0 in for the rightmost bit and shifts everything else left one bit; in effect, it **multiplies** by 2
- A **right** arithmetic shift shifts everything one bit to the right, but copies the sign bit; it **divides** by 2
- EXAMPLE 2.25: Multiply the value 11 (expressed using 8-bit signed two's complement representation) by 2.

  We start with the binary value for 11:
  00001011 (+11)
  We shift left one place, resulting in:
  00010110 (+22)
  The sign bit has not changed, so the value is valid.

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

- EXAMPLE 2.28: Divide the value 12 (expressed using 8-bit signed two's complement representation) by 2.

  We start with the binary value for 12:
  00001100 (+12)
  We shift left one place, resulting in:
  00000110 (+6)
  (Remember, we carry the sign bit to the left as we shift.)

  To divide 12 by **4**, we **right shift twice**.

2.5 Floating-Point Representation 73

- In scientific notion, numbers are expressed in two parts: a **fractional** part call 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 74

- In digital computers, floating-point number 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).

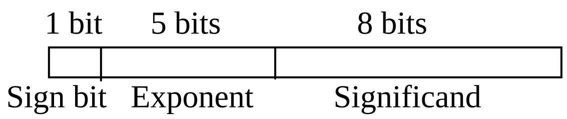


FIGURE 2.1 Simple Model Floating-Point Representation

- Unbiased Exponent

0	00101	10001000	17_{10}	$= 0.10001_2$	$* 2^5$
0	10001	10000000	65536_{10}	$= 0.1_2$	$* 2^{17}$

- Biased Exponent: We select 16 because it is **midway** between 0 and 31 (our exponent has 5 bits, thus allowing for 2^5 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.

0	10101	10001000	$17_{10} = 0.10001_2 * 2^5$ The biased exponent is 16 + 5 = 21
0	01111	10000000	$0.25_{10} = 0.1_2 * 2^{-1}$

- EXAMPLE 2.31
- 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 $(10.25)_{10}$

$(10.25)_{10} = (1010.01)_2$ (Un-normalized form)
$= (1010.01)_2 \times 2^0$.
$= (101.001)_2 \times 2^1$.
:
$= (.101001)_2 \times 2^4$ (**Normalized form**)
$= (.0101001)_2 \times 2^5$ (Un-normalized form)
$= (.00101001)_2 \times 2^6$.

Internal representation of $(10.25)_{10}$ is $\underline{0}$ $\underline{10100}$ $\underline{10100100}$

2.5.2 Floating-Point Arithmetic 76

- EXAMPLE 2.32: Add the following binary numbers as represented in a normalized 14-bit format, using the simple model with a bias of 16.

0	10010	11001000

+

0	10000	10011010

```
   11.001000
+   0.10011010
   11.10111010
```

Renormalizing we retain the larger exponent and **truncate** the low-order bit.

0	10010	11101110

- EXAMPLE 2.33 Multiply:

0	10010	11001000	= 0.11001000 X 2^2

X

0	10000	10011010	= 0.10011010 X 2^0

```
       11001000
     x 10011010
       00000000
      11001000
     00000000
    11001000
   11001000
  00000000
 00000000
11001000_______
111100001010000
```

Renormalizing 0.0111100001010000 * 2^2 = 0.11110000**1010000** * 2^1 we retain the larger exponent and **truncate** the low-order bit.

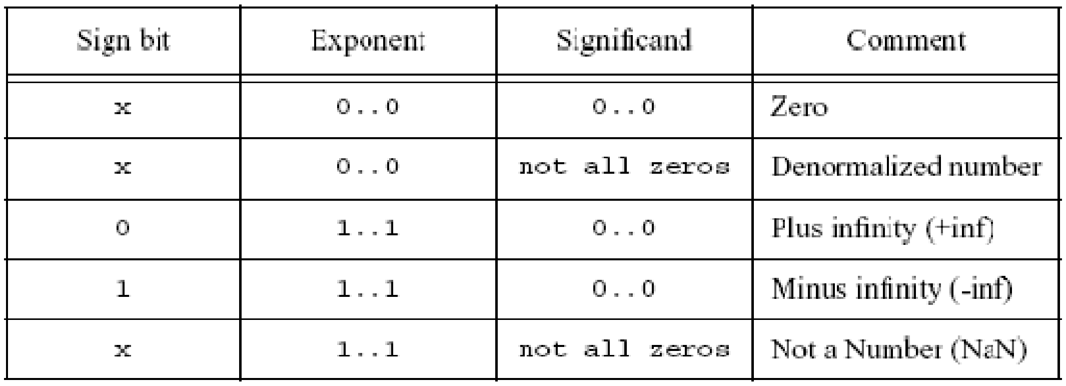
0	10001	11110000

2.5.3 Floating-Point Errors 78

- 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 79

- 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.



Sign bit	Exponent	Significand	Comment
x	0..0	0..0	Zero
x	0..0	not all zeros	Denormalized number
0	1..1	0..0	Plus infinity (+inf)
1	1..1	0..0	Minus infinity (-inf)
x	1..1	not all zeros	Not a Number (NaN)

Special bit patterns in IEEE-754

2.5.5 Range, Precision, and Accuracy 81

- The range of a numeric integer format is the difference between the largest and smallest values that is can express.
- The precision of a number indicates how much information we have about a value
- Accuracy refers to how closely a numeric representation approximates a true value.

2.5.6 Additional Problems with Floating-Point Numbers 82

- Because of truncated bits, you cannot always assume that a particular floating point operation is commutative or distributive.

This means that we **cannot** assume:

$(a + b) + c = a + (b + c)$

or

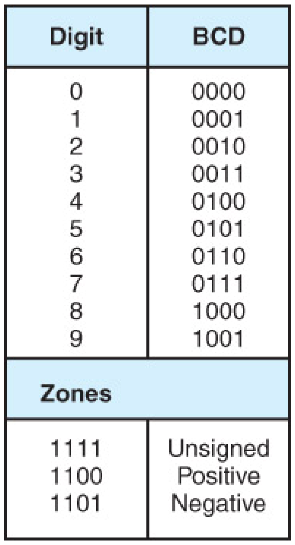
$a * (b + c) = ab + ac$

2.6 Character Codes 85

- 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 86

- Binary-coded Decimal (BCD) is a numeric coding system used primarily in IBM mainframe and midrange systems in the 1950s and 1960s.
- 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.
- When stored in an 8-bit byte, the upper nibble is called the **zone** and the lower part is called the **digit**.



Digit	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
Zones	
1111	Unsigned
1100	Positive
1101	Negative

TABLE 2.5 Binary-Coded Decimal

2.6.2 EBCDIC 87

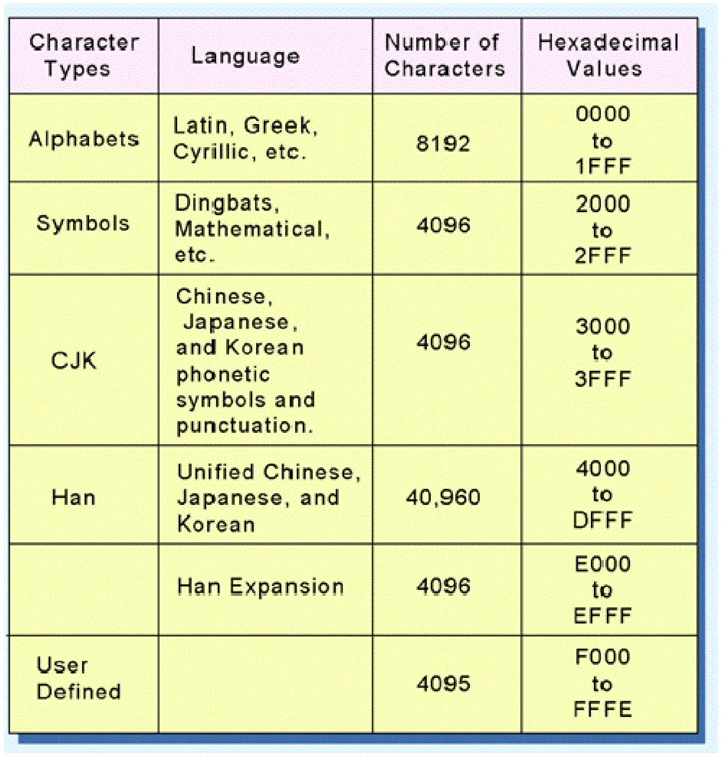
- In 1964, BCD was extended to an 8-bit code, Extended Binary-Coded Decimal Interchange Code (EBCDIC).
- EBCDIC was one of the first widely-used computer codes that supported upper and lowercase alphabetic characters, in addition to special characters, such as punctuation and control characters.
- EBCDIC and BCD are still in use by IBM mainframes today. See Page 77 TABLE 2.6

2.6.3 ASCII 88

- 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 88

- 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.



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

TABLE 2.8 Unicode Codespace

- The Unicode codespace is divided into six parts. The first part is for Western alphabet codes, including English, Greek, and Russian.
- The lowest-numbered Unicode characters comprise the ASCII code.
- The highest provide for user-defined codes.

2.7 Error Detection and Correction 92

- **No** communications channel or storage medium can be completely error-free.

2.7.1 Cyclic Redundancy Check 92

- 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 2The rules are as follows:

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

- EXAMPLE 2.35 Find the sum of 1011_2 and 110_2 modulo 2.

  $1011_2 + 110_2 = 1101_2$ (mod 2)

- EXAMPLE 2.36 Find the quotient and remainder when 1001011_2 is divided by 1011_2.

  Quotient 1010_2 and Remainder 101_2

- Calculating and Using CRC
  - Suppose we want to transmit the information string: 1001011_2.
  - The receiver and sender decide to use the (arbitrary) **polynomial pattern**, 1011.
  - The information string is shifted left by one position less than the number of positions in the divisor. I = $1001011\mathbf{000}_2$
  - The remainder is found through modulo 2 division (at right) and added to the information string: $1001011000_2 + \mathbf{100_2} = 1001011100_2$.
  - 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 work 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.

2.7.2 Hamming Codes 95

- 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(min), for a code is the smallest Hamming distance between all pairs of words in the code.
- Hamming codes can detect (D(min) - 1) errors and correct [(D(min) – 1) / 2] errors.
- EXAMPLE 2.37
- EXAMPLE 2.38

  00000
  01011
  10110
  11101
  D(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:

  $(n + 1) * 2^m < = 2^n$

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

  $(m + r + 1) * 2^m$ **<=** $2^{m + r}$ or

  $\mathbf{(m + r + 1) <= 2^r}$

- EXAMPLE 2.39 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 $(8 + r + 1) <= 2^r$ then We choose r = 4
  Parity bit at 1, 2, 4, 8

  Char K $75_{10} = 01001011_2$

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

We have the following code word as a result:

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

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 + b12 = 1 + 0 + 1 + 0 = **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:

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

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 + b12 = 1 + 0 + 1 + 0 = **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.

2.7.3 Reed-Soloman 102

- 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 103

- 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.
- **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.