Data Representation in Computer Systems

Hamza Osman İLHAN

hoilhan@yildiz.edu.tr

YTU-CE / D037

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.

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.

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.

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.

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*).

Positional Numbering Systems

- Positive radix, positional number systems
- A number with *radix* ***r*** is represented by a string of digits:

  $$A_{n-1}A_{n-2}\ \ldots\ A_1A_0\ .\ A_{-1}\ A_{-2}\ \ldots\ A_{-m+1}\ A_{-m}$$

  in which $\mathbf{0} \leq A_i < r$ and **.** is the *radix point*.
- The string of digits represents the power series:

$$(\text{Number})_r = \left(\sum_{i=0}^{i=n-1} A_i \cdot r^i \right) + \left(\sum_{j=-m}^{j=-1} A_j \cdot r^j \right)$$

(Integer Portion) + (Fraction Portion)

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:

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

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:

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

Converting Binary to Decimal

- To convert to decimal, use decimal arithmetic to form Σ (digit × respective power of 2).
- Example:Convert 11010_2 to N_{10}:

$$1\times2^4 + 1\times2^3 + 0\times2^2 + 1\times2^1 + 0\times2^0 = 26$$

Converting Decimal to Binary

- Method 1
  - Subtract the largest power of 2 that gives a positive remainder and record the power.
  - Repeat, subtracting from the prior remainder and recording the power, until the remainder is zero.
  - Place 1's in the positions in the binary result corresponding to the powers recorded; in all other positions place 0's.
- Example: Convert 625_{10} to N_2
  - $625 - 512 = 113 = N_1$ $\quad 512 = 2^9$
  - $113 - 64 = 49 = N_2$ $\quad 64 = 2^6$
  - $49 - 32 = 17 = N_3$ $\quad 32 = 2^5$
  - $17 - 16 = 1 = N_4$ $\quad 16 = 2^4$
  - $1 - 1 = 0 = N_5$ $\quad 1 = 2^0$

$$(625)_{10} = 1\cdot 2^9 + 0\cdot 2^8 + 0\cdot 2^7 + 1\cdot 2^6 + 1\cdot 2^5 + 1\cdot 2^4 + 0\cdot 2^3 + 0\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0$$

$$= (1001110001)_2$$

Conversion Between Bases

- **Method 2**
- **To convert from one base to another:**

  1) Convert the Integer Part

  2) Convert the Fraction Part

  3) Join the two results with a radix point

Conversion Details

- **To Convert the Integer Part:**

  Repeatedly divide the number by the new radix and save the remainders. The digits for the new radix are the remainders in *reverse order* of their computation. If the new radix is > 10, then convert all remainders > 10 to digits A, B, …

- **To Convert the Fractional Part:**

  Repeatedly multiply the fraction by the new radix and save the integer digits that result. The digits for the new radix are the integer digits in *order* of their computation. If the new radix is > 10, then convert all integers > 10 to digits A, B, …

Example: Convert 46.6875_{10} To Base 2

- **Convert 46 to Base 2:**
  - $(101110)_2$
- **Convert 0.6875 to Base 2:**
  - $(0.1011)_2$
- **Join the results together with the radix point:**
  - $(101110.1011)_2$

Octal to Binary and Back

- Octal to Binary:
  - Restate the octal as three binary digits starting at the radix point and going both ways.
- Binary to Octal:
  - Group the binary digits into three bit groups starting at the radix point and going both ways, padding with zeros as needed in the fractional part.
  - Convert each group of three bits to an octal digit.

Hexadecimal to Binary and Back

- Hexadecimal to Binary:
  - Restate the hexadecimal as four binary digits starting at the radix point and going both ways.
- Binary to Hexadecimal:
  - Group the binary digits into four bit groups starting at the radix point and going both ways, padding with zeros as needed in the fractional part.
  - Convert each group of four bits to a hexadecimal digit.

Octal to Hexadecimal via Binary

- Convert octal to binary.
- Use groups of four bits and convert as above to hexadecimal digits.
- Example: Octal to Binary to Hexadecimal

$$(6 \quad 3 \quad 5 \;.\; 1 \quad 7 \quad 7)_8$$

$$(\underline{110}\ \underline{011}\ \underline{101}\ .\ \underline{001}\ \underline{111}\ \underline{111})_2$$

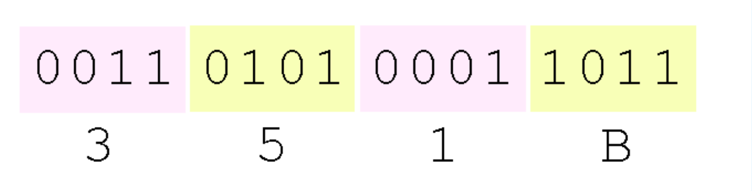
$$(\underline{0001}\ \underline{1001}\ \underline{1101}\ .\ \underline{0011}\ \underline{1111}\ \underline{1000})_2$$

$$(1 \quad 9 \quad D \;.\; 3 \quad F \quad 8)_{16}$$

- Why do these conversions work?

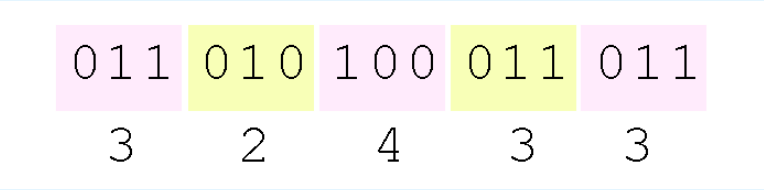
Decimal to Binary Conversions

- 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.

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.

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.

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.

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 . . .

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
  _________
```

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
```

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
```

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
```

Signed Integer Representation

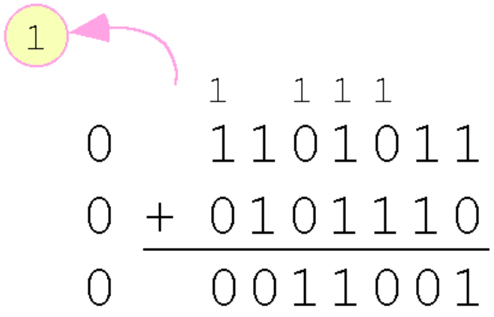
- Example:
  - Using signed magnitude binary arithmetic, find the sum of 75 and 46.
- Once we have worked our way through all eight bits, we are done.

```
        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.

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.



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.

```
     11
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.

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.

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

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

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.

Signed Integer Representation

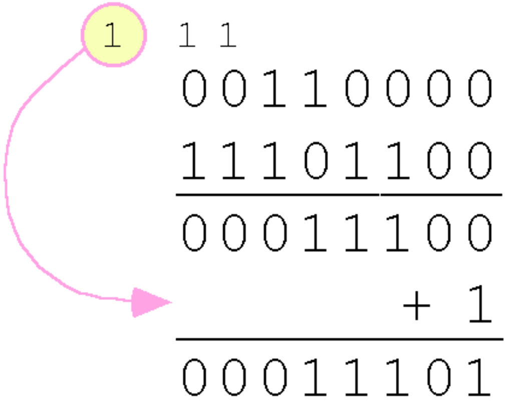
- In complement systems, negative values are represented by some difference between a number and its base.
- In *diminished radix complement* systems, a negative value is given by the difference between the absolute value of a number and one less than its base.
- In the binary system, this gives us *one’s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

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.

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`.

Signed Integer Representation

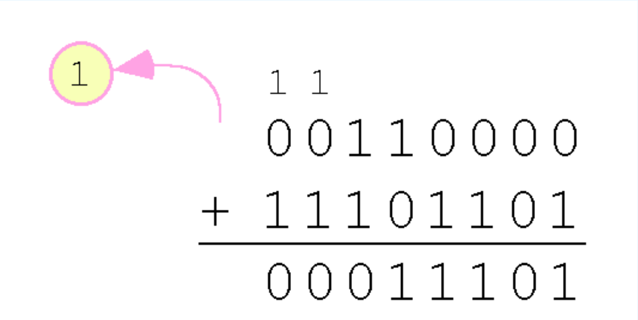
- Although the “end carry around” adds some complexity, one’s complement is simpler to implement than signed magnitude.
- But it still has the disadvantage of having two different representations for zero: positive zero and negative zero.
- Two’s complement solves this problem.
- Two’s complement is the *radix complement* of the binary numbering system.

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`.

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
   00110000
 + 11101101
   00011101
```

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`.

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.

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
```

Rule for detecting two’s complement overflow:
When the “carry in” and the “carry out” of the sign bit differ, overflow has occurred.