Data Representation in Computer Systems

Outline

- Data Organization
  - Bits, Nibbles, Bytes, Words, Double Words
- Numbering Systems
  - Unsigned Binary System
  - Signed and Magnitude System
  - 1’s Complement System
  - 2’s Complement System
  - Hexadecimal System
- Floating Point Representation
- BCD Representation
- Characters
  - ASCII Code
  - UNICODE
- Other Representations
  - Display colors
  - Audio

CS 2401 Comp. Org. & Assembly

Data Representation in Computer Systems -- Chapter 2

Data Organization

Computers use binary number system to store information as 0’s and 1’s

Bits
- A bit is the fundamental unit of computer storage
- A bit can be 0 (off) or 1 (on)
- Related bits are grouped to represent different types of information such as numbers, characters, pictures, sound, instructions

Nibbles

A nibble is a group of 4 bits

A nibble is used to represent a digit in
Hex (from 0-15) and
BCD (Binary-Coded Decimal) (from 0-9) numbers

<table>
<thead>
<tr>
<th>BCD</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>A</td>
</tr>
<tr>
<td>1011</td>
<td>B</td>
</tr>
<tr>
<td>1100</td>
<td>C</td>
</tr>
<tr>
<td>1101</td>
<td>D</td>
</tr>
<tr>
<td>1110</td>
<td>E</td>
</tr>
<tr>
<td>1111</td>
<td>F</td>
</tr>
</tbody>
</table>
A byte is a group of 8 bits that is used to represent numbers and characters. A standard code for representing numbers and characters is ASCII (American Standard Code for Information Interchange).

How many different combinations of 0’s and 1’s with 8 bits can form? In general, how many different combinations of 0’s and 1’s with N bits can form? How many different characters that a byte (8 bits) can represent?

A word is a group of 16 bits or 2 bytes that is used to represent non-Roman characters in UNICODE. An international standard code for representing non-Roman characters like Asian, Greek, and Russian characters is UNICODE.

A double word is a group of 32 bits or 4 bytes or 2 words.
Related Bytes
- A **nibble** is a half-byte (4-bit) - hex representation
- A **word** is a 2-byte (16-bit) data item
- A **doubleword** is a 4-byte (32-bit) data item
- A **quadword** is an 8-byte (64-bit) data item
- A **paragraph** is a 16-byte (128-bit) area
- A **kilobyte** (KB) is $2^{10} = 1,024$ bytes ≈ 1K bytes
- A **megabyte** (MB) is $2^{20} = 1,048,576 = 1$ MB
- A **gigabyte** (GB) is $2^{30} = 1,073,741,824 = 1$ GB
- A **terabyte** (TB) is $2^{40} = 1,099,511,627,776 = 1$ TB

Numbering Systems
- **Unsigned number system**
- **Signed binary Systems**
- Signed and magnitude system
- 1’s complement system
- 2’s complement system
- **Hexadecimal system**

Binary Number System
- base 10 -- has ten digits:
  0,1,2,3,4,5,6,7,8,9
- positional notation
  $2401 = 2 \times 10^3 + 4 \times 10^2 + 0 \times 10^1 + 1 \times 10^0$
- base 2 -- has two digits: 0 and 1
- positional notation
  $1101_2 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0
  = 8 + 4 + 0 + 1 = 13$

Binary Positional Notation
- If
  $N = b_{n-1}b_{n-2}...b_1b_0$
- then
  $N = b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + ... + b_0 \times 2^0$
Positional Notation – Convert base 2 or 16 to base 10

<table>
<thead>
<tr>
<th>Position</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>10000</td>
<td>1000</td>
<td>100</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2 x1000+ 4 x100+ 0 x10 + 1 x1 = 240110</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Unsigned Binary Code

Use for representing integers without signed (natural numbers)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0000</th>
<th>8</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>0001</td>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0010</td>
<td>10</td>
<td>1010</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0011</td>
<td>11</td>
<td>1011</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0100</td>
<td>12</td>
<td>1100</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0101</td>
<td>13</td>
<td>1101</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>0110</td>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>0111</td>
<td>15</td>
<td>1111</td>
</tr>
</tbody>
</table>

BinHex Application

http://cms.dt.uh.edu/faculty/ongards/links/links.php?id=1

Introduction to Computer Technology

Information Representations

Number of Bits Required in Unsigned Binary Code

- What is the range of values that can be represented with $n$ bits in the Unsigned Binary Code?
  
  $[0, 2^n - 1]$

- How many bits are required to represent a given number $N$ in decimal?
  
  Min. Number of Bits = $\log_2(N+1)$

Decimal to Binary Conversion

- 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.
Decimal to Binary Conversion

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.

<table>
<thead>
<tr>
<th>$190$</th>
<th>$162$</th>
<th>$28$</th>
<th>$21$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$3^3$</td>
<td>$3^2$</td>
<td>$3^1$</td>
</tr>
<tr>
<td>$-162$</td>
<td>$= 3^3 \times 2$</td>
<td>$= 3^2 \times 1$</td>
<td>$= 3^1 \times 1$</td>
</tr>
<tr>
<td>$28$</td>
<td>$-21$</td>
<td>$= 3^2 \times 0$</td>
<td>$= 3^1 \times n$</td>
</tr>
<tr>
<td>$21$</td>
<td>$1$</td>
<td>$= 3^1 \times 1$</td>
<td>$= 3^0 \times 1$</td>
</tr>
</tbody>
</table>

Our result, reading from top to bottom is: $190_{10} = 21001_3$

Decimal to Binary Conversion

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.

<table>
<thead>
<tr>
<th>3</th>
<th>190</th>
<th>63</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>63</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>21</td>
<td>1</td>
</tr>
</tbody>
</table>

Decimal to Binary Conversion

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

- 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:
    \[ 190_{10} = 21001_3 \]

Successive division by 2

- What is representation of \(79_{10}\) in binary?
  
  \[
  \begin{array}{c|c}
  \text{Dividend} & \text{Quotient} \\
  79 & 39 \\
  39 & 19 \\
  19 & 9 \\
  9 & 4 \\
  4 & 2 \\
  2 & 1 \\
  1 & 0 \\
  \end{array}
  \]
  Therefore \(79_{10} = 101111_2\)

Reverse Positional Notation

- 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:

  \[
  0.4710_{10} = 4 \times 10^{-1} + 7 \times 10^{-2} + 1 \times 10^{-3}
  \]
  \[
  0.110_2 = 1 \times 2^{-1} + 1 \times 2^{-2}
  \]
  \[
  = \frac{1}{2} + \frac{1}{4}
  \]
  \[
  = 0.5 + 0.25 = 0.75
  \]

Decimal to Binary Conversion

- 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:

  \[
  0.4710_{10} = 4 \times 10^{-1} + 7 \times 10^{-2} + 1 \times 10^{-3}
  \]
  \[
  0.110_2 = 1 \times 2^{-1} + 1 \times 2^{-2}
  \]
  \[
  = \frac{1}{2} + \frac{1}{4}
  \]
  \[
  = 0.5 + 0.25 = 0.75
  \]
Decimal to Binary Conversion

- 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.

Decimal to Binary Conversion

\[
\begin{align*}
0.8125_{10} &= 0.1101_2 \\
0.8125 &= 2^{-1} 	imes 1 \\
-0.5000 &= 2^{-2} 	imes 1 \\
0.3125 &= 2^{-3} 	imes 0 \\
-0.0625 &= 2^{-4} 	imes 1 \\
0.0625 &= 2^{-5} 	imes 1 \\
0 &= 2^{-6} 	imes 0 \\
0.1101_2 &= 0.8125_{10}
\end{align*}
\]

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

Decimal to Binary Conversion

- 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 top to bottom is:

\[
\begin{align*}
0.8125_{10} &= 0.1101_2 \\
0.8125 &= 2^{-1} 	imes 1 \\
-0.5000 &= 2^{-2} 	imes 1 \\
0.3125 &= 2^{-3} 	imes 0 \\
-0.0625 &= 2^{-4} 	imes 1 \\
0.0625 &= 2^{-5} 	imes 1 \\
0 &= 2^{-6} 	imes 0 \\
0.1101_2 &= 0.8125_{10}
\end{align*}
\]

Decimal to Binary Conversion

- Using the multiplication method to convert the decimal 0.8125 to binary, we multiply by the radix 2.

\[
\begin{align*}
0.8125_{10} &= 0.1101_2 \\
0.8125 &= 2^{-1} 	imes 1 \\
0.6250 &= 2^{-2} 	imes 1 \\
0.1250 &= 2^{-3} 	imes 0 \\
0.0625 &= 2^{-4} 	imes 1 \\
0.0000 &= 2^{-5} 	imes 0 \\
0.1101_2 &= 0.8125_{10}
\end{align*}
\]

Decimal to Binary Conversion

- Converting 0.8125 to binary...

\[
\begin{align*}
0.8125 &
\end{align*}
\]

- Ignoring the value in the units place at each step, continue multiplying each fractional part by the radix.

\[
\begin{align*}
0.5000 \\
0.5000 &
\end{align*}
\]
Decimal to Binary Conversion

- 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
    \[ \times 2 \]
    1.6250
    \[ \times 2 \]
    3.2500
    \[ \times 2 \]
    6.5000
    \[ \times 2 \]
    1.0000
  - This method also works with any base. Just use the target radix as the multiplier.

Decimal to Binary Conversion

- 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.

Unsigned Conversion

- Convert an unsigned binary number to decimal
  use positional notation (polynomial expansion)

- Convert a decimal number to unsigned Binary
  use successive division by 2

Examples

- Represent \( 26_{10} \) in unsigned Binary Code
  \[ 26_{10} = 11010_2 \]

- Represent \( 26_{10} \) in unsigned Binary Code using 8 bits
  \[ 26_{10} = 00011010_2 \]

- Represent \( (26)_{10} \) in Unsigned Binary Code using 4 bits -- not possible
Signed Binary Codes

These are codes used to represent positive and negative numbers.
- Signed and Magnitude System
- 1’s Complement System
- 2’s Complement System

Signed and Magnitude

- The most significant (left most) bit represents the sign bit
  - 0 is positive
  - 1 is negative
- The remaining bits represent the magnitude

Examples of Signed & Magnitude

<table>
<thead>
<tr>
<th>Decimal</th>
<th>5-bit Sign and Magnitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>+5</td>
<td>00101</td>
</tr>
<tr>
<td>-5</td>
<td>10101</td>
</tr>
<tr>
<td>+13</td>
<td>01101</td>
</tr>
<tr>
<td>-13</td>
<td>11101</td>
</tr>
</tbody>
</table>

Signed and Magnitude in 4 bits

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>-0</td>
<td>1000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>-1</td>
<td>1001</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>-2</td>
<td>1010</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>-3</td>
<td>1011</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>-4</td>
<td>1100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>-5</td>
<td>1101</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>-6</td>
<td>1110</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>-7</td>
<td>1111</td>
</tr>
</tbody>
</table>
### Examples

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Signed</th>
<th>8-bit Signed</th>
</tr>
</thead>
<tbody>
<tr>
<td>26_{10}</td>
<td>011010₂</td>
<td>00011010₂</td>
</tr>
<tr>
<td>-26_{10}</td>
<td>111010₂</td>
<td>10011010₂</td>
</tr>
</tbody>
</table>

### 1’s Complement System

- **Positive numbers:**
  - same as in unsigned binary system
  - pad a 0 at the leftmost bit position

- **Negative numbers:**
  - convert the magnitude to unsigned binary system
  - pad a 0 at the leftmost bit position
  - complement every bit

### Examples of 1’s Complement

<table>
<thead>
<tr>
<th>Decimal</th>
<th>5-bit 1’s complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>00101</td>
</tr>
<tr>
<td>-5</td>
<td>11010</td>
</tr>
<tr>
<td>13</td>
<td>01101</td>
</tr>
<tr>
<td>-13</td>
<td>10010</td>
</tr>
</tbody>
</table>

### 1’s Complement in 4 bits

<table>
<thead>
<tr>
<th>Decimal</th>
<th>4-bit 1’s complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000  -0  1111</td>
</tr>
<tr>
<td>1</td>
<td>0001  -1  1110</td>
</tr>
<tr>
<td>2</td>
<td>0010  -2  1101</td>
</tr>
<tr>
<td>3</td>
<td>0011  -3  1100</td>
</tr>
<tr>
<td>4</td>
<td>0100  -4  1011</td>
</tr>
<tr>
<td>5</td>
<td>0101  -5  1010</td>
</tr>
<tr>
<td>6</td>
<td>0110  -6  1001</td>
</tr>
<tr>
<td>7</td>
<td>0111  -7  1000</td>
</tr>
</tbody>
</table>
### Examples

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Signed</th>
<th>8-bit Signed</th>
</tr>
</thead>
<tbody>
<tr>
<td>26&lt;sub&gt;10&lt;/sub&gt;</td>
<td>011010&lt;sub&gt;2&lt;/sub&gt;</td>
<td>00011010&lt;sub&gt;2&lt;/sub&gt;</td>
</tr>
<tr>
<td>-26&lt;sub&gt;10&lt;/sub&gt;</td>
<td>100101&lt;sub&gt;2&lt;/sub&gt;</td>
<td>11100101&lt;sub&gt;2&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

### 2’s Complement System

- **Positive numbers:**
  - same as in unsigned binary system
  - pad a 0 at the leftmost bit position
- **Negative numbers:**
  - convert the magnitude to unsigned binary system
  - pad a 0 at the leftmost bit position
  - complement every bit
  - add 1 to the complement number

### Examples of 2’s Complement

<table>
<thead>
<tr>
<th>Decimal</th>
<th>5-bit 2’s complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>00101</td>
</tr>
<tr>
<td>-5</td>
<td>11011</td>
</tr>
<tr>
<td>13</td>
<td>01101</td>
</tr>
<tr>
<td>-13</td>
<td>10011</td>
</tr>
</tbody>
</table>

### 2’s Complement in 4 bits

<table>
<thead>
<tr>
<th>Decimal</th>
<th>4-bit 2’s complement</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>-1</td>
<td>1111</td>
</tr>
<tr>
<td>-2</td>
<td>1110</td>
</tr>
<tr>
<td>-3</td>
<td>1101</td>
</tr>
<tr>
<td>-4</td>
<td>1100</td>
</tr>
<tr>
<td>-5</td>
<td>1011</td>
</tr>
<tr>
<td>-6</td>
<td>1010</td>
</tr>
<tr>
<td>-7</td>
<td>1001</td>
</tr>
<tr>
<td>-8</td>
<td>1000</td>
</tr>
</tbody>
</table>
Examples

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Signed 8-bit Signed</th>
</tr>
</thead>
<tbody>
<tr>
<td>26₁₀</td>
<td>01101₀₂ 0001₁₀₁₀₂</td>
</tr>
<tr>
<td>-26₁₀</td>
<td>10₀₁₁₀₂ 1₁₁₀₀₁₁₀₂</td>
</tr>
</tbody>
</table>

More Examples

- Represent 65 in 2’s complement
  \[65 = 0100\ 0001₂\]
- Represent -65 in 2’s complement
  \[-65 = 1011\ 1111₂\]

Convert 2’s Complement to decimal

Positive 2’s complement numbers
- convert the same as in unsigned binary

Negative 2’s complement numbers
- complement the 2’s complement number
- add 1 to the complemented number
- convert the same as in unsigned binary

Examples

<table>
<thead>
<tr>
<th>2’s complement</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>00101</td>
<td>(4 + 1 = 5)</td>
</tr>
<tr>
<td>11011 (\rightarrow 00100 + 1)</td>
<td>(4 + 1 = 5 \rightarrow -5)</td>
</tr>
<tr>
<td>01101</td>
<td>(8 + 4 + 1 = 13)</td>
</tr>
<tr>
<td>10011 (\rightarrow 01100 + 1)</td>
<td>(8 + 4 + 1 = 13 \rightarrow -13)</td>
</tr>
</tbody>
</table>
Mathematical Formula

- Formula to convert a decimal number to a 1’s complement --
  \[ N' = 2^n - N - 1 \]
- Formula to convert a decimal number to a 2’s complement --
  \[ N' = 2^n - N \]

where \( N \) is the binary number representing the decimal with \( n \) number of bits.

Hexadecimal Notation

- base 16 -- has 16 digits:
  \[ 0 1 2 3 4 5 6 7 8 9 A B C D E F \]
- each Hex digit represents a group of 4 bits (i.e. half of a byte or a nibble)
  0000 to 1111
- use as a shorthand notation for convenient

Convert Binary <-> Hex

<table>
<thead>
<tr>
<th>Binary</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111 0110b</td>
<td>F6h</td>
</tr>
<tr>
<td>1001 1101 0000 1010b</td>
<td>9D0Ah</td>
</tr>
<tr>
<td>1111 0110 1110 0111b</td>
<td>F6E7h</td>
</tr>
<tr>
<td>1011011b</td>
<td>5Bh</td>
</tr>
</tbody>
</table>
Examples
- ASCII value of character ‘D’ in Hex
  \[ D = 0100 \ 0100_{\text{ASCII}} = 44_{\text{ASCII}} \]
- Represent 37d in 2’s complement using Hex.
  \[ 37d = 010 \ 0101b_{2's} = 0010 \ 0101b_{2's} = 25h_{2's} \]
- Represent -37d in 2’s complement using Hex.
  \[ -37d = 101 \ 1111b_{2's} = 1101 \ 1011b_{2's} = DBh_{2's} \]

Convert Large Binary to Decimal
Convert 1001 0011 0101 1100b to decimal

**Method 1:**
- Use polynomial expansion methods

**Method 2:**
- Convert number to hex, then convert it to decimal.

\[
1001 \ 0011 \ 0101 \ 1100b = 935Ch
935Ch = 37724d
\]

Convert Hex ↔ Decimal

**Convert Hex to decimal**
- Use positional (polynomial expansion) notation
  \[ 3BAh = 3 \times 16^2 + B \times 16^1 + A \times 16^0 = 3 \times 256 + 11 \times 16 + 10 \times 1 = 954d \]

**Convert decimal to Hex**
- Use successive divisions by 16
  \[
  \begin{align*}
  359/16 &= 22 \ R 7, \\
  22/16 &= 1 \ R 6, \\
  1/16 &= 0 \ R 1
  \end{align*}
  \]
  \[ 359d = 167h \]

Addition and Subtraction in Signed and Magnitude

\[
\begin{align*}
(a) & \quad 5 & 0101 \\
& & +2 \ 0010 \\
& & \overline{7} \ 0111 \\
(b) & \quad -5 & 1101 \\
& & -2 \ 1010 \\
& & \overline{7} \ 1111 \\
(c) & \quad 5 & 0101 \\
& & -2 \ 1010 \\
& & \overline{3} \ 0111 \\
(d) & \quad -5 & 1101 \\
& & +2 \ 0010 \\
& & \overline{3} \ 1011
\end{align*}
\]
Addition and Subtraction in 1’s Complement

(a) \[ \begin{array}{c}
5 \quad 0101 \\
+2 \quad +0010 \\
\hline
7 \quad 0111 \\
\end{array} \]

(b) \[ \begin{array}{c}
-5 \quad 1010 \\
-2 \quad +1101 \\
\hline
0000 \\
\end{array} \]

(c) \[ \begin{array}{c}
5 \quad 0101 \\
-2 \quad +1101 \\
3 \quad +0011 \\
\hline
0011 \\
\end{array} \]

(d) \[ \begin{array}{c}
-5 \quad 1010 \\
+2 \quad +0010 \\
\hline
3 \quad 1100 \\
\end{array} \]

Theoretical Facts

Why is the carry out from adding 1’s complements added to the sum?
- \( N_1' = 2^n - N_1 - 1 \) and \( N_2' = 2^n - N_2 - 1 \)

Why is the carry out from adding 2’s complements dropped?
- \( N_1' = 2^n - N_1 \) and \( N_2' = 2^n - N_2 \)

Addition and Subtraction in 2’s Complement

(a) \[ \begin{array}{c}
5 \quad 0101 \\
+2 \quad +0010 \\
\hline
7 \quad 0111 \\
\end{array} \]

(b) \[ \begin{array}{c}
-5 \quad 1011 \\
-2 \quad +1110 \\
\hline
-7 \quad 1001 \\
\end{array} \]

(c) \[ \begin{array}{c}
5 \quad 0101 \\
-2 \quad +1110 \\
3 \quad +0011 \\
\hline
3 \quad 1001 \\
\end{array} \]

(d) \[ \begin{array}{c}
-5 \quad 1011 \\
+2 \quad +0010 \\
\hline
-3 \quad 1101 \\
\end{array} \]

Overflow Conditions

**Carry-in \neq \text{carry-out}**

\[
\begin{array}{c|c}
0111 & 1000 \\
5 \quad 0101 & -5 \quad 1011 \\
+3 \quad +0011 & -4 \quad +1100 \\
-8 \quad 1000 & -7 \quad 1011 \\
\end{array}
\]

**Carry-in = \text{carry-out}**

\[
\begin{array}{c|c}
0000 & 1110 \\
+5 \quad 0101 & -2 \quad 1110 \\
+2 \quad +0010 & -6 \quad +1010 \\
-7 \quad 0111 & -8 \quad 11000 \\
\end{array}
\]
Signed Integer Representation

- Overflow and carry are tricky ideas.
- Signed number overflow means nothing in the context of unsigned numbers, which set a carry flag instead of an overflow flag.
- If a carry out of the leftmost bit occurs with an unsigned number, overflow has occurred.
- Carry and overflow occur independently of each other.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
<th>Carry?</th>
<th>Overflow?</th>
<th>Correct result?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100(+4) x 0010(+2)</td>
<td>0110(+6)</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>0100(+4) x 0110(+6)</td>
<td>1010(-4)</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>1100(-4) x 1110(-2)</td>
<td>0110(-6)</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>1100(-4) x 1010(-6)</td>
<td>0110(+6)</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Booth’s Algorithm

- Fast multiplication
- Signed multiplication
- In Booth’s algorithm, the first 1 in a string of 1s in the multiplier is replaced with a subtraction of the multiplicand.
- Shift the partial sums until the last 1 of the string is detected.
- Then add the multiplicand.
Signed Integer Representation

<table>
<thead>
<tr>
<th>$a_i$</th>
<th>$a_{i-1}$</th>
<th>$a_{i-2} - a_i$</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>in middle of string 0. No operation.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>end of string 1. Add multiplicand.</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>-1</td>
<td>beginning of string 1. Subtract multiplicand.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>in middle of string 1. No operation.</td>
</tr>
</tbody>
</table>

Signed Integer Representation

$0011 \times 0110$

$+00000000$

$+01100011$

$+000000$

$+00011$

Ignore all bits over $2^n$.

$100010010$

Signed Integer Representation

$1101 (-3) 0011 (+3)$

$\times 1100 (-4)$

$+00000000$

$+0000000$

$+000000$

$+00011$

$+00000$

$00001100 (+12)$

Signed Integer Representation

$0101 (+5)$

$\times 1100 (-4)$

$+00000000$

$+0000000$

$+111011$

$+00000$

$11101100 (-20)$

<table>
<thead>
<tr>
<th>$a_{i-2} - a_i$</th>
<th>Action</th>
<th>Register</th>
<th>Carry</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>rshf</td>
<td>0000</td>
<td>0110</td>
</tr>
<tr>
<td>00</td>
<td>rshf</td>
<td>0000</td>
<td>0011</td>
</tr>
<tr>
<td>10</td>
<td>sub</td>
<td>+1011</td>
<td>0011</td>
</tr>
<tr>
<td>11</td>
<td>rshf</td>
<td>1110</td>
<td>1100</td>
</tr>
</tbody>
</table>
**Signed Integer Representation**

- 0010110 (+22) 1101010 (-22)
- × 1011110 (-34)

Addition:

+ 00000000000000
+ 111111101010
+ 0000000000
+ 0000000000
+ 0000000000
+ 0000000000
+ 11101010

\[
\begin{array}{c}
\text{111110100010100 (-748)}
\end{array}
\]

**Addition and Subtraction in Hexadecimal System**

**Addition**

\[
\begin{align*}
(9F1B)_{16} + (4A36)_{16} &= 11101101 + 101000110 \quad 1 \\
&= (E951)_{16}
\end{align*}
\]

**Subtraction**

\[
\begin{align*}
(9F1B)_{16} - (4A36)_{16} &= 11101101 - 101000110 \quad 1 \\
&= (54E5)_{16}
\end{align*}
\]

**Floating-Point Representation**

- Floating-point numbers allow an arbitrary number of decimal places to the right of the decimal point.
- For example: 0.5 × 0.25 = 0.125
- They are often expressed in scientific notation.
- For example:
  - 0.125 = 1.25 × 10⁻¹
  - 5,000,000 = 5.0 × 10⁵
Floating-Point Representation

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

\[ + \ 1.25 \times 10^{-1} \]

Floating-Point Representation

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

\[ \text{Sign} \quad \text{Exponent} \quad \text{Significand} \]

- This is the standard arrangement of these fields.

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.

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.
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.

Example:

Express 32 \text{ (10)} in the simplified 14-bit floating-point model.

We know that 32 is 2^5. So in (binary) scientific notation

\[ 32 = 100000 = 1.0 \times 2^5 = 0.1 \times 2^6. \]

Using this information, we put 110 (= 6_{10}) in the exponent field and 1 in the significand as shown.

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.

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.
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.

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.

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
```

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 = \(1.0 \times 2^{-4}\).

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
```
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
```

Floating-Point Representation

- 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.
    - If the significand is zero, the value is ± infinity.
    - If the significand is nonzero, the value is NaN, "not a number," often used to flag an error condition.
- 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.

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.
Floating-Point Numbers

A floating-point number is a representation for real numbers.

IEEE standards set a format for representing floating-point numbers in binary.

Example of an IEEE single-precision format (32 bits long):

\[ 78.375_{10} \approx 429CC000_h \]

Floating Point Structure

The Sign Bit

- 0 denotes a positive number; 1 denotes a negative number.

The Exponent

- represent both positive and negative exponents.
- a bias is added to the actual exponent in order to get the stored exponent.
- For IEEE single-precision floats, this value is 127 with 8 bits.
- For double precision, the exponent field is 11 bits, and has a bias of 1023.

The Mantissa

The mantissa, also known as the significand, represents the precision bits of the number. It is composed of an implicit leading bit and the fraction bits.

<table>
<thead>
<tr>
<th>Sign</th>
<th>Exponent</th>
<th>Mantissa</th>
<th>Bias</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>1</td>
<td>0</td>
<td>23</td>
</tr>
<tr>
<td>Double</td>
<td>1</td>
<td>11</td>
<td>52</td>
</tr>
</tbody>
</table>

IEEE Single-Precision Format

Integral part: 78 ⇒ 1001110
fractional part: 0.375 ⇒ 3/8 = 1/4 + 1/8

\[ \approx .012 + .0012 \]

so

\[ 78.375_{10} = 1.001110011 \times 2^6 \]
**IEEE Single-Precision Format**

1.001110011 × 2^6

- Sign bit is 0
- Exponent including bias of 127 (127 +6 = 133) is 1000 0101 in 8 bits
- Fraction is 00111001100000000000000 23 bits

\[0 1000 0101 00111001100000000000000 = 0100 0010 1001 1100 1100 0000 0000\]

= 42 9C C0 00

**Conversion Procedure**

- The leftmost bit is 0 for positive and 1 for negative.
- Convert the magnitude to decimal binary.
- Convert the binary decimal number to scientific notation.
- Add a bias of 127\_10 to the exponent to form the next 8 bits (to store exponent as a signed number).
- Fraction bits form the last 23 bits.

**Example**

45.5 \(\rightarrow\) 45 = 101101

\[0.5 = 1/2 = .1\]

45.5

\[= 101101.1 = 1.011011 \times 2^5\]

\[= 0 1000 0100 011011000000000000000000\]

\[= 0100 0010 0011 0110 0000 0000 0000\]

\[= 42 36 00 00\]

**Example**

-11.25 \(\rightarrow\) 1 = 1011

\[0.25 = 1/4 = .01\]

-11.25

\[= -1011.01 = -1.01101 \times 2^3\]

\[= 1 1000 0010 011010000000000000000000\]

\[= 1100 0001 0011 0100 0000 0000 0000 0000\]

\[= C1 34 00 00\]
Example

0.125 \rightarrow 0 = 0
0.125 = 1/8 = .001

0.125
= 0.001 = 1.0 \times 2^{-3}
= 0.0111 1100 00000000000000000000000000
= 0011 1110 0000 0000 0000 0000 0000
= 3E 00 00 00

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.

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 2^1.

Thus, our sum is 0.110101 \times 2^4.

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.
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 $0.011100 \times 2^5 = 0.1111 \times 2^4$.
- The normalized product requires an exponent of $22_{10} = 10110_2$.

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.

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:
  $10000000.1_2 = 128.5_{10}$

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

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

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.
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.

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.

Floating-Point Representation

- When discussing floating-point numbers, it is important to understand the terms range, precision, and accuracy.
- The range of a numeric integer format is the difference between the largest and smallest values that it can express.
- Accuracy refers to how closely a numeric representation approximates a true value.
- The precision of a number indicates how much information we have about a value.

Most of the time, greater precision leads to better accuracy, but this is not always true.
- For example, 3.1333 is a value of pi that is accurate to two digits, but has 5 digits of precision.
- There are other problems with floating point numbers.
- Because of truncated bits, you cannot always assume that a particular floating point operation is commutative or distributive.
Floating-Point Representation

- This means that we cannot assume:
  - \((a + b) + c = a + (b + c)\) or
  - \(a(b + c) = ab + ac\)
- Moreover, to test a floating point value for equality to some other number, first figure out how close one number can be to be considered equal. Call this value epsilon and use the statement:
  - if (abs(x) < epsilon) then ...

BCD – Binary Coded Decimal

- A BCD digit is represented by 4 binary bits or a nibble.
- A BCD number is formed by a group of 4 binary bits or nibbles
- That means 8 bits can represent BCD from 0 – 99 and 16 bits can represent BCD from 0 – 9999

Character Representations

- ASCII
- UNICODE

ASCII Code

- Used to represent characters and textual information
- Each character is represented with 1 byte
  - upper and lower case letters: a...z and A...Z
  - decimal digits -- 0,1,...,9
  - punctuation characters -- ; , . :
  - special characters -- $ & @ / { }
  - control characters -- carriage return (CR), line feed (LF), beep
Examples of ASCII Code

<table>
<thead>
<tr>
<th>Character</th>
<th>Binary</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>01010011</td>
<td>83 (binary), 53 (hex)</td>
</tr>
<tr>
<td>8</td>
<td>00111000</td>
<td>56 (binary), 38 (hex)</td>
</tr>
</tbody>
</table>

ASCII Code in Binary and Hex

<table>
<thead>
<tr>
<th>Character</th>
<th>Binary</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>01000001</td>
<td>41</td>
</tr>
<tr>
<td>D</td>
<td>01000100</td>
<td>44</td>
</tr>
<tr>
<td>a</td>
<td>01100001</td>
<td>61</td>
</tr>
<tr>
<td>?</td>
<td>00111111</td>
<td>3F</td>
</tr>
<tr>
<td>2</td>
<td>00110010</td>
<td>32</td>
</tr>
<tr>
<td>DEL</td>
<td>01111111</td>
<td>7F</td>
</tr>
</tbody>
</table>

ASCII Groups

<table>
<thead>
<tr>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Control Character</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Digits &amp; Punctuation</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Upper Case &amp; Special</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Lower Case &amp; Special</td>
</tr>
</tbody>
</table>

ASCII Codes for Numeric Digits

<table>
<thead>
<tr>
<th>Character</th>
<th>Decimal</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>48</td>
<td>30</td>
</tr>
<tr>
<td>1</td>
<td>49</td>
<td>31</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>32</td>
</tr>
<tr>
<td>3</td>
<td>51</td>
<td>33</td>
</tr>
<tr>
<td>4</td>
<td>52</td>
<td>34</td>
</tr>
<tr>
<td>5</td>
<td>53</td>
<td>35</td>
</tr>
<tr>
<td>6</td>
<td>54</td>
<td>36</td>
</tr>
<tr>
<td>7</td>
<td>55</td>
<td>37</td>
</tr>
<tr>
<td>8</td>
<td>56</td>
<td>38</td>
</tr>
<tr>
<td>9</td>
<td>57</td>
<td>39</td>
</tr>
</tbody>
</table>
UNICODE

- UNICODE uses a 16-bit word to represent a single character
- It can represent 65,536 different characters

---

Representing Colors on a Video Display

- An image is composed of pixels (Picture elements)
- Different display modes use different data representations for each pixel
- A mixture of red, green, and blue form a specific color on the display
- Color depth describes amount of each red, green, and blue for a mixture on a pixel -- 8, 16, or 24 bits
- 24-bit display, each color has 256 different shades
- 16-bit display, each color has 5 or 6 bits of shades
- 8-bit display, each color has 2 or 3 bits of shades

---

<table>
<thead>
<tr>
<th>Bit-Depth</th>
<th>Number of Colors</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>monochrome</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>CGA</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>EGA</td>
</tr>
<tr>
<td>8</td>
<td>256</td>
<td>VGA</td>
</tr>
<tr>
<td>16</td>
<td>65,536</td>
<td>High Color, XGA</td>
</tr>
<tr>
<td>24</td>
<td>16,777,216</td>
<td>True Color, SVGA</td>
</tr>
<tr>
<td>32</td>
<td>16,777,216</td>
<td>True Color + Alpha Channel</td>
</tr>
</tbody>
</table>

---

A hardware palette allows an 8-bit display to display a specific color chosen from the colors of 24-bit display

---

Representing Colors on a Video Display

- A hardware palette allows an 8-bit display to display a specific color chosen from the colors of 24-bit display
Audio Information Representation

- Audible sounds are the result of vibrating air molecules quickly back and forth between 20 and 20,000 times per second (Hz).
- A computer is capable of generating a signal that repeatedly applies alternate logic 0 and 1 for a short period of time.
- It requires 5,000 bytes per second to generate 20 kHz sound.

Analog to Digital -- Sampling

- Sampling is done at regular intervals of time, often small fractions of a second.
- Like frequencies, sampling rates are measured in hertz.
- The precision in which a sample represents the actual amplitude of the waveform at the instant the sample is taken depends on the sample size or number of bits (also called bit depth) used in the binary representation of the amplitude value.
- An 8-bit sample can resolve 256 \((=2^8)\) different amplitude or voltage values -- 40,000 bytes/second.
- A 16-bit converter can resolve 65,536 \((=2^{16})\) values -- 80,000 bytes/second.
- Sound recorded on audio CDs is stored as 16-bit samples.
- When a sample is taken, the actual value is rounded to the nearest value that can be represented by the number of bits in a sample.
Audio Formats

- **MIDI**
  - Musical Instrument Digital Interface is not technically an audio format, but it has recently become predominant as one of the main methods for delivering audio over the Internet. This is due to the fact that the file size are tiny compared to any other audio formats. The beauty behind MIDI files is the fact that it only save the data on what notes the instrument should play rather than the whole complex structure of sound waves.

- **WAV**
  - This format has become the standard audio format for sound files on the Internet. Almost every browser has built-in WAV playback support. The default Windows WAV format is PCM, which is basically uncompressed sound data, and these files tend to be rather large. However, many forms of compressed WAV files are available.

- **MPEG (Layer 3)**
  - This is latest of MPEG audio coding. It achieves high-fidelity sound quality, with a significant reduction in file size. It can shrink down CD audio by a factor of 12, without losing any clarity and quality. The encoded file are small enough to be transmitted at today's Internet speeds, this is one of the main reasons why mp3's are attracting so many users in the Internet community.