Overview of Computer Science

CSC 101 — Summer 2011
Representing Discrete Data
Number Systems / Representation
Lecture 5 — July 12, 2011

Announcements

• Lab #1 Today
• Lab #2 on Thursday
  – Prelab is posted online
• Writing Assignment #2 is due on Thursday
  – Tim-Berners Lee Video (TED)
  – You will post this on your CSC101 blog

Objectives

• Digitizing discrete information
  – Text
  – Simple numbers
• Decimal, binary and hexadecimal numbering systems
• Integer and Floating Point Representation
Digital Information

- Digital computers process digital information
- Digital information is discrete; however, natural forms of information are analog and continuous
- The process of converting information to a digital form is called digitization
- Both discrete and analog information may be digitized
  - Information that is already discrete (numbers, text characters, etc.) is easily represented in a digital form
  - Analog information must be converted in some way

Digitizing Analog Information

- Analog information is continuous (non-discrete)
  - Must be transformed into a discrete form for digitizing
- Analog information is digitized in two steps:
  1. Sampling: Discrete samples are chosen to represent the continuous data
  2. Quantizing: Each sample is assigned a particular number
- Text and numbers are discrete information
  - Digitization is simply a matter of conversion from one discrete form to another

Digitizing Discrete Information

- Simple numbers are discrete information
  - If we can write it down exactly, it has an exact, distinct value
  - We can choose whatever symbols we want to use to write it
  - These symbols all have an order or collating sequence that gives them meaning
    - 1 < 2 < 3 < 4 < 5 < 6 < 7 < 8 < 9
    - i < ii < iii < iv < v < vi < vii < viii < ix
Digitizing Discrete Information

- To digitize simple numbers (integers)
  - Could just write down the decimal digits
  - But, storing numbers in digital computers works best when the symbols used are only the digits 0 and 1.
    - Because a computer’s memory is made up of bits
      - Remember that we call it binary data when we use just two digits
- One way to digitize simple numbers using binary data
  - Convert the decimal number to its base-2 equivalent
    - (e.g. $183_{10} = 10110111_2$)
  - We can then store numbers as binary data just by setting some bits to either on or off to represent the binary digits
- To digitize large or small numbers or fractions
  - More complex binary representations are used
  - Floating point representation

Digitizing Discrete Information

- Inside a computer, all information consists of bit patterns
  - Remember that each bit in a computer’s memory is just a tiny ‘switch’ that is either “on” or “off”
  - Any kind of information can be represented by choosing the right pattern of bits
  - For base-2 numbers, the choice of bit pattern is obvious:
    - Each bit represents one digit of a binary number
    - If the bit is “off”, it’s a zero
    - If the bit is “on”, it’s a one
  - But, we live in a world of decimal (base-10) numbers
    - Need a method to represent decimal numbers using bits
    - Easiest solution: use binary numbers to represent decimal numbers

Decimal vs. Binary Numbers

- Decimal is a base-10 positional numbering system
- Each digit is a 0 – 9 and represents a multiple of a power of ten
**Decimal vs. Binary Numbers**

- **Decimal** is a base-10 positional numbering system
- Each digit is a 0 – 9 and represents a multiple of a power of ten
- Example: 183

\[
\begin{align*}
0 \times 10^3 &= 0 \\
1 \times 10^2 &= 100 \\
8 \times 10^1 &= 80 \\
3 \times 10^0 &= 3 \\
\hline
&= 183
\end{align*}
\]

---

**Decimal vs. Binary Numbers**

- **Binary** is a base-2 positional numbering system
- Each digit is a 0 or 1 and represents a multiple of a power of two
- Example: 183

\[
\begin{align*}
1 \times 2^7 &= 128 \\
0 \times 2^6 &= 0 \\
1 \times 2^5 &= 32 \\
1 \times 2^4 &= 16 \\
0 \times 2^3 &= 0 \\
1 \times 2^2 &= 4 \\
1 \times 2^1 &= 2 \\
1 \times 2^0 &= 1 \\
\hline
&= 183_{10} = 10110111_2
\end{align*}
\]

---

**Decimal-to-Binary Conversion**

1. Write down the powers of two
2. Starting with the largest power of two, subtract it if possible
3. If you can subtract it, write down a 1; if you can’t, write down a 0
4. Continue until you’re done

\[
\begin{align*}
128 &= 2^7 \\
64 &= 2^6 \\
32 &= 2^5 \\
16 &= 2^4 \\
8 &= 2^3 \\
4 &= 2^2 \\
2 &= 2^1 \\
1 &= 2^0
\end{align*}
\]
Binary-to-Decimal Conversion

1. Write down the powers of two
   - 128 = 2^7
   - 64 = 2^6
   - 32 = 2^5
   - 16 = 2^4
   - 8 = 2^3
   - 4 = 2^2
   - 2 = 2^1
   - 1 = 2^0
2. For each digit that is a 1, add in the corresponding power of two
3. For each digit that is a 0, skip that power of two
4. Continue until you’re done

Other Bases

- **Decimal** is base 10, using 10 digits (0–9)
  - Each digit represents a multiple of a power of 10
- **Binary** is base 2, using 2 digits (0, 1)
  - Each digit represents a multiple of a power of 2
- Numbers can be represented in any base, not just base 10 or base 2
  - Sometimes other bases are more convenient
- **Octal** is base 8, using 8 digits (0–7)
  - Each digit represents a multiple of a power of 8
- **Hexadecimal** is base 16, using 16 digits (0–9 and A–F)
  - Each digit represents a multiple of a power of 16
  - Hex is a compact and convenient way to write down bit sequences

<table>
<thead>
<tr>
<th>Decimal (Dec)</th>
<th>Hex (Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>A</td>
</tr>
<tr>
<td>11</td>
<td>B</td>
</tr>
<tr>
<td>12</td>
<td>C</td>
</tr>
<tr>
<td>13</td>
<td>D</td>
</tr>
<tr>
<td>14</td>
<td>E</td>
</tr>
<tr>
<td>15</td>
<td>F</td>
</tr>
</tbody>
</table>

Using hex for only need a single digit to encode any four bits from 0 to 15, when you get above 15 you start using letters
**Hexadecimal**

- **Hexadecimal** is a base-16 positional numbering system
- Each digit is a 0–9 or A–F and represents a multiple of a power of sixteen
- Example: $183_{10} = B7_{16}$

\[
\begin{align*}
183 \times 256 &= 0 \cdot 16^2 \\
11 \times 16 &= 176 \cdot 16^1 \\
7 \times 1 &= 7 \cdot 16^0 \\
183_{10} &= B7_{16}
\end{align*}
\]

**Dec** | **Hex**
---|---
0 | 0
1 | 1
2 | 2
3 | 3
4 | 4
5 | 5
6 | 6
7 | 7
8 | 8
9 | 9
A | 10
B | 11
C | 12
D | 13
E | 14
F | 15

**Hexadecimal**

- Converting between decimal and hexadecimal is similar to binary conversion
  - Uses multiplication and division
- The real benefit of hex:
  - It is very easy to convert between binary and hex
  - Hex provides a convenient way to write down binary bit sequences

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

**Hexadecimal**

- Hex uses 16 digits (0–9 and A–F)
- With four bits, we can represent exactly 16 values
- So, the 16 hex digits can exactly represent groups of four bits

<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>
**Hexadecimal**

- We can simplify by using hex to represent the bit patterns
  - Instead of writing
    
    \[0100\ 0001\ 0100\ 0001\ 0101\ 0100\ 0101\ 0011\]
  - we could have written
    
    \[43\ 41\ 54\ 53\]

Another example:

\[0100\ 0011\ 0100\ 0001\ 0101\ 0100\ 0101\ 0011\]

<table>
<thead>
<tr>
<th>Dec</th>
<th>Binary</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1000</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>1101</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1010</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>1000</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>1010</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>1101</td>
<td>3</td>
</tr>
</tbody>
</table>

Can be represented as

- #88008E
- #FF4920
- #ABEE38
- #AA2920
- #5228BE

Colors in Hexadecimal

- Colors can be represented in hexadecimal
  - In HTML, one way to specify an arbitrary color is by indicating exact amounts of the three primary colors (red, green, blue – “RGB”)
  - **This shade of ORANGE**
    - consists of a mixture of lots of red, some green, and a little blue
    - The amount of each primary color, specified as a number between 0 and 255, would be
      - Red = 255, Green = 142, Blue = 10
  - The binary bit patterns that represent those numbers is
    - 255 = 11111111, 142 = 10001110, 10 = 00001010
  - \(<\text{font color="#FFAE0A">}\) is easier
Digitizing Discrete Information

• Simple numbers are easy to digitize because they represent discrete information.
• Text is also discrete information.
  – Individual letters, punctuation, etc.
• Text letters can be digitized by converting to a binary form.
  – But, what pattern of 1s and 0s is best for representing characters?
  – An early method was to represent letters with “dots”, the way a scoreboard does:

```
   00000000 00000000 00000000 00000000
   00000000 00000000 00000000 00000000
   00000000 00000000 00000000 00000000
```

  – But, this is very inefficient and depends upon the two-dimensional arrangement of the bits.

Digitizing Discrete Information

• Simple numbers are easy to digitize because they represent discrete information.
• Text is also discrete information.
  – Individual letters, punctuation, etc.
• Text letters can be digitized by converting to a binary form.
  – A better method: choose a standard code for the conversion
    – ASCII (American Standard Code for Information Interchange)
      – Each character is represented using 7 bits (one byte) – 128 possible characters
    – UNICODE (A universal code for all characters in all languages on all platforms)
      – Each character is represented using two, three or four bytes – billions of possible characters
  – Each letter or punctuation is assigned a unique bit pattern, based on the chosen code.
    – An example using ASCII:
      - “A” = 01000001, “B” = 01000010, “C” = 01000011, ...
      - “CATS” = 01000011 01000001 01010100 01010011

The ASCII Code

<table>
<thead>
<tr>
<th>Character</th>
<th>ASCII Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>01000001</td>
</tr>
<tr>
<td>B</td>
<td>01000010</td>
</tr>
<tr>
<td>C</td>
<td>01000011</td>
</tr>
<tr>
<td>D</td>
<td>01000100</td>
</tr>
<tr>
<td>E</td>
<td>01000101</td>
</tr>
<tr>
<td>F</td>
<td>01000110</td>
</tr>
<tr>
<td>G</td>
<td>01000111</td>
</tr>
<tr>
<td>H</td>
<td>01001000</td>
</tr>
<tr>
<td>I</td>
<td>01001001</td>
</tr>
<tr>
<td>J</td>
<td>01001010</td>
</tr>
<tr>
<td>K</td>
<td>01001011</td>
</tr>
<tr>
<td>L</td>
<td>01001100</td>
</tr>
<tr>
<td>M</td>
<td>01001101</td>
</tr>
<tr>
<td>N</td>
<td>01001110</td>
</tr>
<tr>
<td>O</td>
<td>01001111</td>
</tr>
<tr>
<td>P</td>
<td>01010000</td>
</tr>
<tr>
<td>Q</td>
<td>01010001</td>
</tr>
<tr>
<td>R</td>
<td>01010010</td>
</tr>
<tr>
<td>S</td>
<td>01010011</td>
</tr>
<tr>
<td>T</td>
<td>01010100</td>
</tr>
<tr>
<td>U</td>
<td>01010101</td>
</tr>
<tr>
<td>V</td>
<td>01010110</td>
</tr>
<tr>
<td>W</td>
<td>01010111</td>
</tr>
<tr>
<td>X</td>
<td>01011000</td>
</tr>
<tr>
<td>Y</td>
<td>01011001</td>
</tr>
<tr>
<td>Z</td>
<td>01011010</td>
</tr>
<tr>
<td>0</td>
<td>01100000</td>
</tr>
<tr>
<td>1</td>
<td>01100001</td>
</tr>
<tr>
<td>2</td>
<td>01100010</td>
</tr>
<tr>
<td>3</td>
<td>01100011</td>
</tr>
<tr>
<td>4</td>
<td>01100100</td>
</tr>
<tr>
<td>5</td>
<td>01100101</td>
</tr>
<tr>
<td>6</td>
<td>01100110</td>
</tr>
<tr>
<td>7</td>
<td>01100111</td>
</tr>
<tr>
<td>8</td>
<td>01101000</td>
</tr>
<tr>
<td>9</td>
<td>01101001</td>
</tr>
<tr>
<td>10</td>
<td>01101010</td>
</tr>
<tr>
<td>11</td>
<td>01101011</td>
</tr>
<tr>
<td>12</td>
<td>01101100</td>
</tr>
<tr>
<td>13</td>
<td>01101101</td>
</tr>
<tr>
<td>14</td>
<td>01101110</td>
</tr>
<tr>
<td>15</td>
<td>01101111</td>
</tr>
<tr>
<td>16</td>
<td>01110000</td>
</tr>
<tr>
<td>17</td>
<td>01110001</td>
</tr>
<tr>
<td>18</td>
<td>01110010</td>
</tr>
<tr>
<td>19</td>
<td>01110011</td>
</tr>
<tr>
<td>20</td>
<td>01110100</td>
</tr>
<tr>
<td>21</td>
<td>01110101</td>
</tr>
<tr>
<td>22</td>
<td>01110110</td>
</tr>
<tr>
<td>23</td>
<td>01110111</td>
</tr>
<tr>
<td>24</td>
<td>01111000</td>
</tr>
<tr>
<td>25</td>
<td>01111001</td>
</tr>
<tr>
<td>26</td>
<td>01111010</td>
</tr>
<tr>
<td>27</td>
<td>01111011</td>
</tr>
<tr>
<td>28</td>
<td>01111100</td>
</tr>
<tr>
<td>29</td>
<td>01111101</td>
</tr>
<tr>
<td>30</td>
<td>01111110</td>
</tr>
<tr>
<td>31</td>
<td>01111111</td>
</tr>
</tbody>
</table>
Digitizing Discrete Information

- Digitized data is just stored as binary bits
  - No specific identification indicating what type of data
  - The same bit patterns could be
    - Text (in ASCII code)
      - 01000011 01000001 01010100 01010011 = “CATS”
    - Simple integers (represented as base-2 numbers)
      - 01000011 01000001 01010100 01010011 = 67,65,84,83
    - More complicated numbers (using floating-point representation)
      - 01000011 01000001 01010100 01010011 = “0.0000755102935”
    - Part of an image or sound
      - 01000011 01000001 01010100 01010011 = four pixels
    - Etc.
  - The software knows what type of data it is looking for

Integers and Real Numbers

- Integers
  - All whole numbers – negative, positive, and zero
  - An infinite, discrete set
  - Can be represented digitally only within a limited range
    - The digital representation is just the equivalent binary number
    - Range is dictated by the number of binary digits (bits) used
  - But, can be represented digitally with exact precision
    - The binary number is exactly the same as the original integer

- Real numbers
  - All numbers – negative, positive, and zero – including fractions
  - An infinite and continuous set (the entire number line)
  - Represented digitally only within a limited range
  - Represented digitally only with a limited precision
    - Not all numbers can be represented exactly
    - Decimal example: \( \frac{1}{2} = 0.5 \) (exactly)
      - but \( \frac{1}{3} = 0.\overline{3} \)
    - Binary example:
      - \( 0.8_{10} = 0.1100110110011001001100..._{2} \)
**Integer Storage**

- Typical personal computers use 16 bits (2 bytes) to store integers.
- First bit is used to represent the sign (– or +) …leaving 15 bits for the rest of the number.
- The largest 15-bit binary number is fifteen “1” bits: 
  \[ \pm 111111111111111 \times 2^{15} = \pm 32,767_{10} \]
- So, every integer from –32,767 to +32,767 can be stored with exact precision.
- If a calculation results in a number outside this range, we say that an integer overflow has occurred.

**Real Number Storage**

- Typical personal computers use 32 bits (4 bytes) to store real numbers.
- Real numbers are stored in floating-point representation:
  - 1-bit sign (– or +) for the number.
  - 8-bit exponent (one bit used for the – or + sign of the exponent).
  - 23-bit mantissa (just the digits without a decimal point).
    - No digits to the right of the decimal point.
  - A base-10 example of exponent and mantissa:
    - \( 318.39 = 31839. \times 10^{-2} \)
    - In binary: \( 318.39 = 10011111001100011101 \times 2^{-2} \)

**Exponents and Mantissas**

- For each of these numbers, what would be the exponent and what would be the mantissa?

  - \( 3.14159 = +314159. \times 10^{-3} \)
  - \( 0.000057048 = +57048. \times 10^{-9} \)
  - \( -708,939.805 = -708939805. \times 10^{-3} \)
  - \( 6.022 \times 10^{-23} = +6022. \times 10^{-20} \)
Real Number Storage

32-bit number

+ / – Exponent Mantissa

Example: 318.39 = +31839. × 10⁻²

sign: + => 0

exponent: -2 => 10000111

mantissa: 31839 => 1001111100110011110101

So, the floating-point representation is:

01000011110011110011001110101

• Limited range (due to 8-bit exponent):
  - Largest real number (– or +) that can be stored is about 2¹²⁸ (about 10⁴⁰)
  - Smallest real number (closest to zero) that can be stored is about 2⁻¹²⁸ (about 10⁻⁴⁰)

• Limited precision (due to the 23-bit mantissa):
  - Many numbers have a very long or infinitely long fractional representation
    e.g. \( \frac{1}{3} = 0.33333333333333333... \)
    or \( \pi = 3.14159265358979323... \)
    or \( 0.8_{10} = 0.1100110011001100110011001100110011001100110011001100..._2 \)
  - But we don’t have an infinite number of bits in the mantissa – only 23 bits
Real Number Storage Limitations

- Three types of numbers can’t be correctly represented by a floating-point representation:
  1. Numbers out-of-range because their absolute value is too large (overflow error)
  2. Numbers out-of-range because their absolute value is too small (too close to zero) (underflow error)
  3. Numbers which require more than 23 binary digits for an accurate representation (limited precision)

Risks in Numerical Computing

- Almost all computer calculations have some precision error (roundoff error)
- Must be anticipated and accommodated
  - Most software products deal with it somehow
  - Potential for catastrophic results

Risks in Numerical Computing

- Examples of some catastrophic failures
  - Ariane 5 inaugural launch (1996)
    - Failed due to an overflow error
  - Patriot missile in the First Gulf War (1991)
    - Can intercept SCUD missiles
    - Failed due to a precision error
- Further details at:
  - [http://cs.furman.edu/digitaldomain/themes/risks/risks_numeric.htm](http://cs.furman.edu/digitaldomain/themes/risks/risks_numeric.htm)
  - [http://www.ima.umn.edu/~arnold/disasters/patriot.html](http://www.ima.umn.edu/~arnold/disasters/patriot.html)
Round-off Error & the Patriot Missile

The Patriot system used floating-point number representation to represent tenths of seconds, with only 23 bits available.

The floating-point value \(0.00011001100110011001100110011001100110011001100110011\) used to represent 0.1 sec. works out to 0.0999999046326… (not exactly 0.10).

This slight error compounded over time, as shown below: