Extra Lecture: Number Systems

◆ Objectives - To understand:
  ◆ Base of number systems: decimal, binary, octal and hexadecimal
  ◆ Textual information stored as ASCII
  ◆ Binary addition/subtraction, multiplication
  ◆ Binary logical operations
  ◆ Unsigned and signed binary number systems
  ◆ Fixed point binary representations
  ◆ Floating point representations

◆ By the end of the lecture, you should be able to:
  ◆ Convert between numbers represented in different bases
  ◆ Convert between fixed point and floating point numbers
  ◆ Perform simple binary arithmetic and logical operations
  ◆ Read and interpret hexadecimal numbers with reasonable speed

---

Decimal number system

◆ We are familiar with decimal number representation. For example:

<table>
<thead>
<tr>
<th>Hundreds</th>
<th>Tens</th>
<th>Ones</th>
<th>Tenths</th>
<th>Hundredths</th>
</tr>
</thead>
<tbody>
<tr>
<td>10²</td>
<td>10¹</td>
<td>10⁰</td>
<td>10⁻¹</td>
<td>10⁻²</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>2</td>
<td>.1</td>
<td>5</td>
</tr>
</tbody>
</table>

◆ The value of this number is calculated as:

\[
4 \times 10^2 = 4 \times 100 = 400.
6 \times 10^1 = 6 \times 10 = 60.
2 \times 10^0 = 2 \times 1 = 2.
1 \times 10^{-1} = 1 \times 0.1 = 0.1.
5 \times 10^{-2} = 5 \times 0.01 = +0.05
\]

462.15

◆ In general, the relationship between the contribution of a digit, its position, and the base of the system is given by:

\[
\text{DIGIT} \times \text{BASE}^{\text{POSITION}}\]

◆ Usually, we restrict

\[0 \leq \text{DIGIT} \leq \text{BASE} - 1\]
The bases of a number system

- There is no reason why one should be restricted to using base-10 (decimal) numbers only.
- Digital computers use a binary number system where the base (or radix) is 2:

<table>
<thead>
<tr>
<th>Fours</th>
<th>Twos</th>
<th>Ones</th>
<th>Halves</th>
<th>Fourths</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^2$</td>
<td>$2^1$</td>
<td>$2^0$</td>
<td>$2^{-1}$</td>
<td>$2^{-2}$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- For example, the value of this binary number is:

\[
\begin{align*}
1 \cdot 2^2 &= 1 \cdot 4 = 4. \\
1 \cdot 2^1 &= 1 \cdot 2 = 2. \\
0 \cdot 2^0 &= 0 \cdot 1 = 0. \\
1 \cdot 2^{-1} &= 1 \cdot 0.5 = 0.5 \\
1 \cdot 2^{-2} &= 1 \cdot 0.25 = 0.25
\end{align*}
\]

\[ \text{Answer: } 1.011 \]

Converting decimal integers to binary

- Repeatedly divide the decimal number by 2 (the base of the binary system).
- Division by 2 will either give a remainder of 1 or 0.
- Collecting the remainders (LSB first!) gives the binary answer.
- Convert 11₁₀ into binary

\[
\begin{array}{c|c c c c}
2 & 11 \\
2 & 5 & r & 1 \\
2 & 2 & r & 1 \\
2 & 1 & r & 0 \\
2 & 0 & r & 1 \\
\end{array}
\]

\[ \text{Answer: } 1011 \]
Octal and hexadecimal number systems

<table>
<thead>
<tr>
<th>Binary</th>
<th>Octal</th>
<th>Decimal</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>10</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>1001</td>
<td>11</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>12</td>
<td>10</td>
<td>A</td>
</tr>
<tr>
<td>1011</td>
<td>13</td>
<td>11</td>
<td>B</td>
</tr>
<tr>
<td>1100</td>
<td>14</td>
<td>12</td>
<td>C</td>
</tr>
<tr>
<td>1101</td>
<td>15</td>
<td>13</td>
<td>D</td>
</tr>
<tr>
<td>1110</td>
<td>16</td>
<td>14</td>
<td>E</td>
</tr>
<tr>
<td>1111</td>
<td>17</td>
<td>15</td>
<td>F</td>
</tr>
</tbody>
</table>

- Run out of “normal” digit symbols

Nibbles, Bytes, Words

- Internal datapaths inside computers could be different width - for example 4-bit, 8-bit, 16-bit or 32-bit.
- For example: ARM processor uses 32-bit internal datapath
- WORD = 32-bit for ARM (byte and nibble are architecture independent)

<table>
<thead>
<tr>
<th>32</th>
<th>24</th>
<th>23</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- MSB
- LSB
- Nibble
- Byte
- Word
Hexadecimal representation

- Convenient to divide any size of binary numbers into nibbles
- Represent each nibble as hexadecimal – think of the human!
- Example:

```
0100 1101 0110 1011  1000  0011 0000 1111
4    D    6    B     8    3    0    F
```

- This is possible because 16 is a power of 2
- Converting from decimal to hexadecimal is the same as converting to binary, except, divide by 16 instead of 2:

```
16 | 237
  16 | 14 r 13
   0 r 14
```

Answer: \( \text{ED}_h \)

Representing Text in ASCII

- Textual information must also be stored as binary numbers.
- Each character is represented as a 7-bit number known as ASCII codes (American Standard Code for Information Interchange)
- For example, ‘A’ is represented by \( 41_h \) and ‘a’ by \( 61_h \)
Signed numbers

- So far, numbers are assumed to be unsigned (i.e. positive)
- How to represent signed numbers?
- Solution 1: **Sign-magnitude** - Use one bit to represent the **sign**, the remain bits to represent **magnitude**

<table>
<thead>
<tr>
<th></th>
<th>7</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Advantage: easy human reasoning – it’s what we use
- Problem: addition and subtraction require quite complex circuits

Two’s complement

- Solution 2: **Two’s complement** – represent a negative number \( x \) by the number \( 2^N + x \), in an \( n \)-bit representation:
- Example: Encode -27 in 8 bit two’s complement –

\[
256 - 27 = 229 \\
229_{10} = 1110 0101_b
\]

- So long as we only want to represent numbers with magnitude less than \( 2^{N-1} \), the MSB is still the **sign bit**
- Example: Encode 27 in 8 bit two’s complement –

\[
27_{10} = 0001 1011_b
\]
Two’s complement

- Another view of two’s complement is to represent a negative number by taking its magnitude, inverting all bits and adding one:

<table>
<thead>
<tr>
<th>Positive number</th>
<th>+27 = 0001 1011&lt;sub&gt;b&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invert all bits</td>
<td>1110 0100&lt;sub&gt;b&lt;/sub&gt;</td>
</tr>
<tr>
<td>Add 1</td>
<td>-27  = 1110 0101&lt;sub&gt;b&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

- Why is this the same?
- Inverting all bits is the same as subtracting the number from 11….1:

<table>
<thead>
<tr>
<th>“All ones”</th>
<th>1111 1111&lt;sub&gt;b&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Positive number</td>
<td>+27 = 0001 1011&lt;sub&gt;b&lt;/sub&gt;</td>
</tr>
<tr>
<td>Subtraction</td>
<td>1110 0100&lt;sub&gt;b&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

- So inverting and then adding one is the same as subtracting from 11….1 + 1 = 100….0 = 2<sup>N</sup>

<table>
<thead>
<tr>
<th>“All ones”</th>
<th>1111 1111&lt;sub&gt;b&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add 1</td>
<td>0000 0001&lt;sub&gt;b&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

Two’s complement

- A third (and final!) way to view two’s complement is that the weight of position <i>i</i> is 2<sup>i</sup> except the MSB, which has negative weight

\[
x = -b_{N-1}2^{N-1} + b_{N-2}2^{N-2} + \cdots + b_12^1 + b_02^0
\]

-27  = 1110 0101<sub>b</sub>  = -128 + 64 + 32 + 4 + 1

- Why is this the same?
- If we interpreted this as an unsigned number, it would be

\[
y = b_{N-1}2^{N-1} + b_{N-2}2^{N-2} + \cdots + b_12^1 + b_02^0
\]

- If <i>x</i> is negative, sign bit <i>b_{N-1}</i> is 1, so difference is <i>y – x = 2<sup>N</sup></i>, i.e. <i>y = 2<sup>N</sup> + x</i>
Why 2’s complement representation?

- If we represent signed numbers in 2’s complement form, subtraction is the same as addition to the (2’s complemented) number (if we ignore any carry out)

\[
\begin{array}{c|c}
27 & 0001 1011_b \\
-17 & 0001 0001_b \\
\hline
+10 & 0000 1010_b \\
\end{array}
\]

\[
\begin{array}{c|c}
+27 & 0001 1011_b \\
+(-17) & 1110 1111_b \\
\hline
+10 & 0000 1010_b \\
\end{array}
\]

- Note that the range for 8-bit unsigned and signed numbers are different.
  - 8-bit unsigned: 0 …… +255
  - 8-bit 2’s complement signed number: -128 …… +127

Sign Extension

- How to translate an 8-bit 2’s complement number to a 16-bit 2’s complement number?

This operation is known as **sign extension**.
- Result is the same: trivially so for positive numbers
**Sign Extension**

- For negative numbers, consider a 1-bit sign extension.

\[
\begin{array}{c|c|c}
2^7 & 2^6 & 2^0 \\
-2^7 & 2^6 & 2^0 \\
\end{array}
\]

\[
\begin{array}{c}
\text{difference} \\
-256 \\
\end{array}
\]

\[
\begin{array}{c}
\text{difference} \\
128-(-128) = 256 \\
\end{array}
\]

Total difference = 0

i.e. same number!

---

**Fixed point representation**

- So far, we have concentrated on integer representation with the fractional part.
- There is an implicit binary point to the right:

\[
\begin{array}{c|c|c}
S & N-1 \\
\end{array}
\]

- In general, the binary point can be in the middle of the word (or off the end!)

\[
\begin{array}{c|c|c}
S & N-1 \\
\end{array}
\]
Idea of floating point representation

- Although fixed point representation can cope with numbers with fractions, the range of values that can represented is still limited.
- Alternative: use the equivalent of “scientific notation”, but in binary:

\[
\text{number} = \text{sign} \times \text{mantissa} \times 2^e
\]

- For example:

10.5 in fixed point \[1010.1_2\]
Move binary point to left \[1.0101_2 \times 2^3\]
10.5 = \[1.3125 \times 8\]

IEEE-754 standard floating point

- 32-bit single precision floating point:

<table>
<thead>
<tr>
<th>single precision</th>
<th>31</th>
<th>30</th>
<th>23</th>
<th>22</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>8-bit exp</td>
<td>23-bit frac</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[x = -1^s \times 2^{\text{exp}-127} \times 1.\text{frac}\]

\[1.175 \times 10^{-38} < |x| < 1.7 \times 10^{38}\]

- MSB is sign-bit (same as fixed point)
- 8-bit exponent in bias-127 integer format (i.e. store 127+exponent)
- 23-bit to represent only the fractional part of the mantissa. The MSB of the mantissa is ALWAYS ‘1’, therefore it is not stored