Representing Information

“Bit Juggling”

- Representing information using bits
- Number representations
- Some other bits

Announcements

TA Bud Vasile’s office hours:
Monday 2-3:30pm @ SN 034
Tuesday 2-3:30pm @ SN 034
From Last Time

1) Computers are tools for processing information.
2) Information resolves uncertainty.
3) Information is measured in bits.
4) A fact that narrows $N$ possible choices down to $M$ provides
   $$\log_2(N/M)$$ bits of information.

This time we’ll explore how information is represented in computers and discuss specific examples.

Encoding

- Encoding describes the process of assigning representations to information
- Choosing an appropriate and efficient encoding is a real engineering challenge
- Impacts design at many levels
  - Mechanism (devices, # of components used)
  - Efficiency (bits used)
  - Reliability (noise)
  - Security (encryption)
Fixed-Length Encodings

If all choices are equally likely (or we have no reason to expect otherwise), then a fixed-length code is often used. Such a code will use at least enough bits to represent the information content.

- Decimal digits 10 = \{0,1,2,3,4,5,6,7,8,9\}
- 4-bit BCD (binary code decimal)
  \[ \log_2(10 / 1) = 3.322 < 4 \text{ bits} \]

- \approx 84 English characters = \{A-Z (26), a-z (26), 0-9 (11), punctuation (8), math (9), financial (4)\}
- 7-bit ASCII (American Standard Code for Information Interchange)
  \[ \log_2(84 / 1) = 6.392 < 7 \text{ bits} \]

Encoding Positive Integers

It is straightforward to encode positive integers as a sequence of bits. Each bit is assigned a weight. Ordered from right to left, these weights are increasing powers of 2. The value of an n-bit number encoded in this fashion is given by the following formula:

\[ n-1 \sum_{i=0} b_i \cdot 2^i \]

\[ 011111010000 \]

\[ \begin{align*}
  2^4 &= 16 \\
  + 2^6 &= 64 \\
  + 2^7 &= 128 \\
  + 2^8 &= 256 \\
  + 2^9 &= 512 \\
  + 2^{10} &= 1024 \\
  &= 2000_{10}
\end{align*} \]
Some Tricks with Bits

- You are going to have to get accustomed to working in binary. Specifically for Comp 120, but it will be helpful throughout your career as a computer scientist.
- Here are some helpful guides

1. Memorize the first 10 powers of 2

\[
\begin{array}{lcl}
2^0 &=& 1 \\
2^1 &=& 2 \\
2^2 &=& 4 \\
2^3 &=& 8 \\
2^4 &=& 16 \\
2^5 &=& 32 \\
2^6 &=& 64 \\
2^7 &=& 128 \\
2^8 &=& 256 \\
2^9 &=& 512
\end{array}
\]

More Tricks with Bits

- You are going to have to get accustomed to working in binary. Specifically for Comp 120, but it will be helpful throughout your career as a computer scientist.
- Here are some helpful guides

2. Memorize the prefixes for powers of 2 that are multiples of 10

\[
\begin{array}{lcl}
2^{10} &=& \text{Kilo (1024)} \\
2^{20} &=& \text{Mega (1024*1024)} \\
2^{30} &=& \text{Giga (1024*1024*1024)} \\
2^{40} &=& \text{Tera (1024*1024*1024*1024)} \\
2^{50} &=& \text{Peta (1024*1024*1024*1024*1024)} \\
2^{60} &=& \text{Exa (1024*1024*1024*1024*1024*1024*)}
\end{array}
\]
Even More Tricks with Bits

- You are going to have to get accustomed to working in binary. Specifically for Comp 120, but it will be helpful throughout your career as a computer scientist.
- Here are some helpful guides

```
01000111000000000110000000000101000
```

3. When you convert a binary number to decimal, first break it down into clusters of 10 bits.
4. Then compute the value of the leftmost remaining bits (1) find the appropriate prefix (GIGA) (Often this is sufficient)
5. Compute the value of and add in each remaining 10-bit cluster

Other Helpful Clusterings

Oftentimes we will find it convenient to cluster groups of bits together for a more compact representation. The clustering of 3 bits is called Octal. Octal is not that common today.

```
<table>
<thead>
<tr>
<th>Octal - base 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 - 0</td>
</tr>
<tr>
<td>001 - 1</td>
</tr>
<tr>
<td>010 - 2</td>
</tr>
<tr>
<td>011 - 3</td>
</tr>
<tr>
<td>100 - 4</td>
</tr>
<tr>
<td>101 - 5</td>
</tr>
<tr>
<td>110 - 6</td>
</tr>
<tr>
<td>111 - 7</td>
</tr>
</tbody>
</table>
```

```
0*8^0 = 0
+ 2*8^1 = 16
+ 7*8^2 = 448
+ 3*8^3 = 1536
```

```
03720
```

```
= 2000_{10}
```

```
\sum_{i=0}^{n-1} 8^i d_i
```

Seems natural to me!
### One Last Clustering

Clusters of 4 bits are used most frequently. This representation is called hexadecimal. The hexadecimal digits include 0-9, and A-F, and each digit position represents a power of 16.

\[
\sum_{i=0}^{n-1} d_i 16^i
\]

**Hexadecimal - base 16**

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hexadecimal</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>

Clusters of 4 bits are used most frequently. This representation is called hexadecimal. The hexadecimal digits include 0-9, and A-F, and each digit position represents a power of 16.

\[
0 \times 16^0 = 0 + 13 \times 16^1 = 208 + 7 \times 16^2 = 1792 = 2000_{10}
\]

**Example: ASCII table**

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Oct</th>
<th>Hex</th>
<th>ASCII Code</th>
<th>ASCII Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>00</td>
<td>0</td>
<td>Space</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>01</td>
<td>1</td>
<td>Shift out</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td>02</td>
<td>2</td>
<td>Shift in</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td>03</td>
<td>3</td>
<td>Data link escape</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>04</td>
<td>4</td>
<td>Device control 1</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>05</td>
<td>5</td>
<td>Device control 2</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>06</td>
<td>6</td>
<td>Device control 3</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>07</td>
<td>7</td>
<td>Device control 4</td>
</tr>
<tr>
<td>8</td>
<td>100</td>
<td>08</td>
<td>8</td>
<td>Negative acknowledge</td>
</tr>
<tr>
<td>9</td>
<td>101</td>
<td>09</td>
<td>9</td>
<td>Synchr 1</td>
</tr>
<tr>
<td>10</td>
<td>110</td>
<td>A</td>
<td>A</td>
<td>Synchr 2</td>
</tr>
<tr>
<td>11</td>
<td>111</td>
<td>B</td>
<td>B</td>
<td>Synchr 3</td>
</tr>
<tr>
<td>12</td>
<td>0000</td>
<td>C</td>
<td>C</td>
<td>[</td>
</tr>
<tr>
<td>13</td>
<td>0001</td>
<td>D</td>
<td>D</td>
<td>]</td>
</tr>
<tr>
<td>14</td>
<td>0010</td>
<td>E</td>
<td>E</td>
<td>\</td>
</tr>
<tr>
<td>15</td>
<td>0011</td>
<td>F</td>
<td>F</td>
<td>/</td>
</tr>
</tbody>
</table>

---

Marc Pollefeys  1/18/2005  00:51
Signed-Number Representations

There are also schemes for representing signed integers with bits. One obvious method is to encode the sign of the integer using one bit. Conventionally, the most significant bit is used for the sign. This encoding for signed integers is called the SIGNED MAGNITUDE representation.

\[ v = -1^b \sum_{i=0}^{n-1} 2^i b_i \]

\[ \begin{array}{c}
\text{Signed Number} \\
\hline
-2000 \\
2000
\end{array} \]

Even though this approach seems straightforward, it is not used that frequently in practice (with one important exception).

One more trick...

Sum of powers of two

\[ \sum_{i=0}^{n-1} 2^i = 2^n - 1 \]

Complement

\[ \begin{array}{c}
\text{Complement of 47} \\
\hline
2048 \\
47
\end{array} \]

\[ \begin{array}{c}
\text{Complement of 2000} \\
\hline
2048 \\
2000
\end{array} \]
2's Complement Integers

The 2's complement representation for signed integers is the most commonly used signed-integer representation. It is a simple modification of unsigned integers where the most significant bit represents a negative power of 2.

8-bit 2's complement example:

\[ \begin{align*}
11010110 = -2^7 + 2^6 + 2^4 + 2^2 + 2^1 \\
&= -128 + 64 + 16 + 4 + 2 = -42
\end{align*} \]

Why 2's Complement?

If we use a two's complement representation for signed integers, the same binary addition mod \(2^n\) procedure will work for adding positive and negative numbers (don't need separate subtraction rules). The same procedure will also handle unsigned numbers!

When using signed magnitude representations, adding a negative value really means to subtract a positive value. However, in 2's magnitude, adding is adding regardless of sign. In fact, you NEVER need to subtract when you use a 2's complement representation.

Example:

\[ \begin{align*}
55_{10} &= 00110111_2 \\
+ 10_{10} &= 00001010_2 \\
65_{10} &= 01000001_2
\end{align*} \]

\[ \begin{align*}
55_{10} &= 00110111_2 \\
-10_{10} &= 11110110_2 \\
45_{10} &= 100101101_2
\end{align*} \]
2's Complement Tricks

- Negation – changing the sign of a number
  - First complement every bit (i.e. $1 \rightarrow 0$, $0 \rightarrow 1$)
  - Add 1
  - Example: $20 = 00010100$, $-20 = 11101011 + 1 = 11101100$

- Sign-Extension – aligning different sized 2's complement integers

16-bit version of 42 = 0000 0000 0010 1010
8-bit version of -2 = 1111 1111 1111 1110

signed-number vs. 2's complement

<table>
<thead>
<tr>
<th>b,b,b,b&lt;sub&gt;n&lt;/sub&gt;</th>
<th>Unsigned</th>
<th>Sign and Magnitude</th>
<th>Two's complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>15</td>
<td>-7</td>
<td>-1</td>
</tr>
<tr>
<td>1110</td>
<td>14</td>
<td>-6</td>
<td>-2</td>
</tr>
<tr>
<td>1101</td>
<td>13</td>
<td>-5</td>
<td>-3</td>
</tr>
<tr>
<td>1100</td>
<td>12</td>
<td>-4</td>
<td>-4</td>
</tr>
<tr>
<td>1011</td>
<td>11</td>
<td>-3</td>
<td>-5</td>
</tr>
<tr>
<td>1010</td>
<td>10</td>
<td>-2</td>
<td>-6</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td>-1</td>
<td>-7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>-0</td>
<td>-8</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Fixed-Point Numbers

By moving the implicit location of the “binary” point, we can represent signed fractions too. This has no effect on how operations are performed, assuming that the operands are properly aligned.

\[
\begin{array}{cccccccc}
-2^3 & 2^2 & 2^1 & 2^0 & 2^{-1} & 2^{-2} & 2^{-3} & 2^{-4} \\
\end{array}
\]

\[
1101.0110 = -2^3 + 2^2 + 2^0 + 2^{-2} + 2^{-3} = -8 + 4 + 1 + 0.25 + 0.125 = -2.625
\]

OR

\[
1101.0110 = -42 * 2^{-4} = -42/16 = -2.625
\]

Repeated Binary Fractions

Not all fractions can be represented exactly using a finite representation. You’ve seen this before in decimal notation where the fraction 1/3 (among others) requires an infinite number of digits to represent (0.3333...).

In Binary, a great many fractions that you’ve grown attached to require an infinite number of bits to represent exactly.

EX: 
1 / 10 = 0.1_{10} = .00011_2 
1 / 5 = 0.2_{10} = .00111_2 = 0.333_16

Marc Pollefey 1/18/2005 00:51
Bias Notation

There is yet one more way to represent signed integers, which is surprisingly simple. It involves subtracting a fixed constant from a given unsigned number. This representation is called “Bias Notation”.

\[ v = \sum_{i=0}^{n-1} 2^i b_i - \text{Bias} \]

\[
\begin{array}{cccccccc}
2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\
1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\
\end{array}
\]

EX: (Bias = 127)

\[
\begin{align*}
6 \times 1 &= 6 \\
13 \times 16 &= 208 \\
\end{align*}
\]

Why? Monotonicity

Floating Point Numbers

Another way to represent numbers is to use a notation similar to Scientific Notation. This format can be used to represent numbers with fractions (3.90 x 10^-4), very small numbers (1.60 x 10^-19), and large numbers (6.02 x 10^23). This notation uses two fields to represent each number. The first part represents a normalized fraction (called the significand), and the second part represents the exponent (i.e. the position of the “floating” binary point).

\[ \text{Normalized Fraction} \times 2^{\text{Exponent}} \]

“bits of accuracy”  “dynamic range”
Lecture Notes

IEEE 754 Format

- Single precision format

\[ v = -1^S \times 1.\text{Significand} \times 2^{\text{Exponent} - 127} \]

- Double precision format

\[ v = -1^S \times 1.\text{Significand} \times 2^{\text{Exponent} - 1023} \]

Summary

1) Selecting the encoding of information has important implications on how this information can be processed, and how much space it requires.

2) Computer arithmetic is constrained by finite representations, this has advantages (it allows for complement arithmetic) and disadvantages (it allows for overflows, numbers too big or small to be represented).

3) Bit patterns can be interpreted in an endless number of ways, however important standards do exist
   - Two’s complement
   - IEEE 754 floating point