Binary Information

3735928559
(convert to hex!)
Binary information

• Topics
  – Base conversions (bin/dec/hex)
  – Data sizes
  – Byte ordering
  – Character and program encodings
  – Bitwise operations
Core theme

What does this mean?

IOO
Core theme

Information = Bits + Context
Why binary?

• Computers store information in binary encodings
  – 1 bit is the simplest form of information (on / off)
  – Minimizes storage and transmission errors
• To store more complicated information, use more bits
  – However, we need context to understand them
  – Data encodings provide context
  – For the next two weeks, we will study encodings
  – First, let’s become comfortable working with binary
Base conversions

- **Binary encoding** is base-2: bit $i$ represents the value $2^i$
  - Bits typically written from most to least significant (i.e., $2^3 \ 2^2 \ 2^1 \ 2^0$)

  1 \[= 1 = 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = [0001] \]

  5 \[= 4 + 1 = 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = [0101] \]

  11 \[= 8 + 2 + 1 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = [1011] \]

  15 \[= 8 + 4 + 2 + 1 = 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = [1111] \]

**Binary to decimal:**
Add up all the powers of two (memorize powers of two to make this go faster!)

**Decimal to binary:**
Find highest power of two and subtract to find the remainder
Repeat above until the remainder is zero
Every power of two become 1; all other bits are 0
Remainder system

• Quick method for decimal → binary conversions
  – Repeatedly divide decimal number by two until zero, keeping track of remainders (either 0 or 1)
  – Read in reverse to get binary equivalent

\[
\begin{align*}
11 & \\
5 & r 1 \\
2 & r 1 \quad \Rightarrow \quad 1011 \quad (8 + 2 + 1) \\
1 & r 0 \\
0 & r 1
\end{align*}
\]
What is the decimal number 25 when represented in binary?

25
12 r 1
6 r 0 => 11001 (16 + 8 + 1)
3 r 0
1 r 1
0 r 1
• **Hexadecimal** encoding is base-16 (usually prefixed with “0x”)
  
  - Converting between hex and binary is easy
    - Each digit represents 4 bits; just substitute digit-by-digit or in groups of four!
    - You should memorize (at least some of) these equivalences

<table>
<thead>
<tr>
<th>Dec</th>
<th>Bin</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>
</tbody>
</table>
Base conversions

- \( \text{0x4CA} \leftrightarrow 0100 \ 1100 \ 1010 \)
- \( \text{0x5F0} \leftrightarrow 0101 \ 1111 \ 0000 \)

<table>
<thead>
<tr>
<th>Dec</th>
<th>Bin</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>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dec</th>
<th>Bin</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<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>
Fundamental data sizes

- **1 byte** = 2 hex digits (= 2 nibbles!) = 8 bits

  \[
  \begin{array}{cccc}
  2^7 & 2^6 & 2^5 & 2^4 \\
  128 & 64 & 32 & 16 \\
  \hline
  2^3 & 2^2 & 2^1 & 2^0 \\
  8 & 4 & 2 & 1 \\
  \end{array}
  \]

  1 byte: 1 hex digit (Y) 1 hex digit (Z)

  Value of byte 0xYZ is \(16 \cdot Y + Z\)

- **Machine word** = size of an address
  - (i.e., the size of a pointer in C)
  - Early computers used 16-bit addresses
    - Could address \(2^{16}\) bytes = 64 KB
  - Now 32-bit (4 bytes) or 64-bit (8 bytes)
    - Can address 4GB or 16 EB

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Bin</th>
<th>Dec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kilo</td>
<td>(2^{10})</td>
<td>(~10^3)</td>
</tr>
<tr>
<td>Mega</td>
<td>(2^{20})</td>
<td>(~10^6)</td>
</tr>
<tr>
<td>Giga</td>
<td>(2^{30})</td>
<td>(~10^9)</td>
</tr>
<tr>
<td>Tera</td>
<td>(2^{40})</td>
<td>(~10^{12})</td>
</tr>
<tr>
<td>Peta</td>
<td>(2^{50})</td>
<td>(~10^{15})</td>
</tr>
<tr>
<td>Exa</td>
<td>(2^{60})</td>
<td>(~10^{18})</td>
</tr>
</tbody>
</table>
Byte ordering

- **Big endian**: store **higher** place values at lower addresses
  - Most-significant byte (MSB) to least-significant byte (LSB)
  - Similar to standard way to write hex (implied with “0x” prefix)
- **Little endian**: store **lower** place values at lower addresses
  - Least-significant byte (LSB) to most-significant byte (MSB)
  - Default byte ordering on most Intel-based machines

\[
\begin{array}{c|c|c|c|c}
\text{low} & \text{high} \\
\text{addr} & \text{addr} \\
\hline
0x11223344 \text{ in big endian:} & 11 & 22 & 33 & 44 \\
0x11223344 \text{ in little endian:} & 44 & 33 & 22 & 11 \\
\end{array}
\]
Byte ordering examples

- **Big endian**: most significant byte first (MSB to LSB)
- **Little endian**: least significant byte first (LSB to MSB)

<table>
<thead>
<tr>
<th>Hex Value</th>
<th>Big Endian</th>
<th>Little Endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x11223344</td>
<td>11 22 33 44</td>
<td>44 33 22 11</td>
</tr>
</tbody>
</table>

- Decimal: 1
  - 16-bit big endian: 00000000 00000001 (hex: 00 01)
  - 16-bit little endian: 00000001 00000000 (hex: 01 00)

- Decimal: 19 (16+2+1)
  - 16-bit big endian: 00000000 00010011 (hex: 00 13)
  - 16-bit little endian: 00010011 00000000 (hex: 13 00)

- Decimal: 256
  - 16-bit big endian: 00000001 00000000 (hex: 01 00)
  - 16-bit little endian: 00000000 00000001 (hex: 00 01)
What is the byte in the highest address when hexadecimal number 0x8345 is stored in little-endian ordering?

- A) 0x83
- B) 0x45
- C) 0x34
- D) 0x85
- E) There is not enough information to tell.
Character encodings

- **ASCII** ("American Standard Code for Information Interchange")
  - 1-byte code developed in 1960s
  - Limited support for non-English characters
- **Unicode**
  - Multi-byte code developed in 1990s
  - "All the characters for all the writing systems of the world"
  - Over 136,000 characters in latest standard
  - Fixed-width (**UTF-16** and **UTF-32**) and variable-width (**UTF-8**)
Program encodings

- Machine code
  - Binary encoding of **opcodes** and operands
  - Specific to a particular CPU architecture (e.g., x86_64)

```
int add (int num1, int num2)
{
    return num1 + num2;
}
```

```
0000000000400606 <add>:

400606:  55  push  %rbp
400607:  48 89 e5  mov  %rsp,%rbp
40060a:  89 7d fc  mov  %edi,-0x4(%rbp)
40060d:  89 75 f8  mov  %esi,-0x8(%rbp)
400610:  8b 55 fc  mov  -0x4(%rbp),%edx
400613:  8b 45 f8  mov  -0x8(%rbp),%eax
400616:  01 d0  add  %edx,%eax
400618:  5d  pop  %rbp
400619:  c3  retq
```
Bitwise operations

- **Basic bitwise operations**
  - & (and)  | (or)  ^ (xor)
- **Not boolean algebra!**
  - && (and)  || (or)  ! (not)
  - 0 (false)  non-zero (true)
- **Important properties:**
  - \(x \& 0 = 0\)
  - \(x \& 1 = x\)
  - \(x | 0 = x\)
  - \(x | 1 = 1\)
  - \(x ^ 0 = x\)
  - \(x ^ 1 = \neg x\)
  - \(x ^ x = 0\)

- **Commutative:**
  - \(x \& y = y \& x\)
  - \(x | y = y | x\)
  - \(x ^ y = y ^ x\)

- **Associative:**
  - \((x \& y) \& z = x \& (y \& z)\)
  - \((x | y) | z = x | (y | z)\)
  - \((x ^ y) ^ z = x ^ (y ^ z)\)

- **Distributive:**
  - \(x \& (y | z) = (x \& y) | (x \& z)\)
  - \(x | (y \& z) = (x | y) \& (x | z)\)
Bitwise operations

- **Bitwise complement (~)** - “flip the bits”
  - \(~0000 = 1111\) \((\sim 0 = 1)\) \( \sim 1010 = 0101\) \((\sim 0xA = 0x5)\)

- **Left shift (<<) and right shift (>>)**
  - Equivalent to multiplying (<<) or dividing (>>) by two
  - Left shift: \(0110 << 1 = 1100\) \(1 << 3 = 8\)
  - Logical right shift (fill zeroes): \(1100 >> 2 = 0011\)
  - Arithmetic right shift (fill most sig. bit): \(1100 >> 2 = 1111\)
    \(0100 >> 2 = 0001\)

**On stu:**
- `int`: \(0f000000 >> 8 = 0000f000\) (arithmetic, for signed integers)
- `int`: \(ff000000 >> 8 = fff00000\)
- `uint`: \(0f000000 >> 8 = 0000f000\) (logical, for unsigned integers)
- `uint`: \(ff000000 >> 8 = 000ff0000\)
Masking

- Bitwise operations can extract parts of a binary value
  - This is referred to as masking; specify a bit pattern mask to indicate which bits you want
    - Helpful fact: 0xF is all 1’s in binary!
  - Use a bitwise AND (&) with the mask to extract the bits
  - Use a bitwise complement (~) to invert a mask
  - Example: To extract the lower-order 16 bits of a larger value \( v \), use \( v \& \ 0xFFFF \)

\[
\begin{align*}
0xDEADBEEF \ & 0xFFFF & = 0x0000BEEF & = \ 0xBEEF \\
0xDEADBEEF \ & 0x0000FFFF & = 0x0000BEEF & = \ 0xBEEF \\
0xDEADBEEF \ & 0xFFFF0000 & = 0xDEAD0000 \\
0xDEADBEEF \ & \sim 0xFFFF & = 0xDEAD0000 \\
0xDEADBEEF \ & \sim 0x0000FFFF & = 0xDEAD0000
\end{align*}
\]