Bits and Bytes

CMPU 224 – Computer Organization
Jason Waterman
What is a bit?

• All data stored in computer systems (hard drives, memory, SD cards, etc.) is stored as bits
• A bit represents one of two states, “on/off”, “true/false”, “1/0”
• How that bit is stored depends on the medium
  • Magnetic (hard drive, floppy disk)
  • Electronic (RAM, Registers)
  • Optical (CD / DVD / Punch Cards)

• Regardless of how it’s stored
  • A binary digit (or bit) takes on the value of either 0 or 1
• A bit is not very much data, so we usually group a bunch of bits together into logical groupings
Bytes and Words

• A byte is a group of 8 bits:
  • 01100110
  • 01100011
  • 00110010
  • 10101010

• How many unique bytes are there?
  • $2^8 = 256$, so a byte can represent up to 256 of some thing

• But bytes are still too small to be the basic size of data for a computer

• For the past 20 years or so the basic word size of most computers was 32 bits; today newer machines have a word size of 64 bits

• 00101010011011010100100110011011011000010011000111110100
  • Looking at a string of 64 bits is somewhat overwhelming and not a great way of transmitting information
  • We’ll see a way to represent words more compactly later
Decimal (base 10) Number System

- Decimal is base 10
  - Numbers are represented by the symbols 0-9
  - A decimal number has a one’s place (10^0), a ten’s place (10^1), a hundred’s place (10^2), a thousand’s place (10^3) and so on

\[
\begin{align*}
10^3 & \quad 10^2 & \quad 10^1 & \quad 10^0 \\
1000 & \quad 100 & \quad 10 & \quad 1
\end{align*}
\]

\[
224
\]

\[
100 \cdot 2 + 10 \cdot 2 + 1 \cdot 4
\]
Binary (base 2) Number System

- Binary is base 2
  - Numbers are represented by the symbols 0 and 1
  - A binary number has a one’s place \(2^0\), a two’s place \(2^1\), a four’s place \(2^2\), an eight’s place \(2^3\), and so on

\[
\begin{array}{cccccc}
2^3 & 2^2 & 2^1 & 2^0 \\
0 & 1 & 0 & 1 \\
\hline
\end{array}
\]

\[4 + 1 = 5\]
Hexadecimal (base 16) Number System

- Hexadecimal (hex) is a base 16 representation
  - We use the letters A-F as the extra “digits” so we count:
    - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
    - 10 11 12 13 14 15
  - A hexadecimal number has a one’s place \((16^0)\), a sixteen’s place \((16^1)\), a two-hundred-fifty-six’s place \((16^2)\), and so on
    - 0x54B54CDB6DC263F4

- Each hexadecimal digit represents how many bits? 4

- How many hexadecimal digits are in a byte? 2

- How many hexadecimal digits to represent a 64-bit number? 16
Applying Concepts: Ternary (base 3) numbers

- What is the base 10 (decimal) representation of the following ternary number:
  - $1021_3$

\[
\begin{array}{cccc}
3 & 2 & 1 & 0 \\
\hline
3 & 3 & 3 & 3 \\
\hline
27 & 9 & 3 & 1 \\
\hline
10 & 2 & 1 \\
\end{array}
\]

\[
27 + 2 \cdot 3 + 1 = 34_{10}
\]
Reading Binary

• You should become comfortable reading binary numbers up to 255
  • It’s fun and will soon become second nature
  • All you need to do is add combinations of:
    • 1, 2, 4, 8, 16, 32, 64, and 128
• Just like decimal numbers the least significant digit (or least significant bit LSB) is on the right
  • in the number 123, the rightmost digit (3) is in the one’s place, etc.
  • Example: for the binary number 0101
    • There is a 1 in the 1’s place and a 1 in the 4’s place, so the value is $1 + 4 = 5$
Binary examples

• $1111 = 8 + 4 + 2 + 1 = 15$
• $1000 = 8 + 0 + 0 + 0 = 8$
• $1001 = 8 + 0 + 0 + 1 = 9$
• $11001001 = 128 + 64 + 0 + 0 + 8 + 0 + 0 + 1 = 201$
Practice Problems

• Convert each of these binary values to decimal

  • 11111111  \[ 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 2^8 - 1 = 255 \]

  • 11110000  \[ 128 + 64 + 32 + 16 = \]

  • 00001111  \[ 8 + 4 + 2 + 1 = 15 \]

  • 01010101  \[ 64 + 16 + 4 + 1 = 85 \]
Converting from decimal to binary

• Example: $42_{10}$
  • $32$ is the largest power of two number $\leq 42$ so we know we have a one in the $32$’s column. Subtract $32$ from $42$, leaving $10$
  • $16 > 10$, so we have a zero in the $16$’s column
  • $8 \leq 10$, so we have a one in the $8$’s column. Subtract $8$ from $10$, leaving $2$
  • $4 > 2$, so we have a zero in the $4$’s column.
  • $2 \geq 2$, so we have a one in the $2$’s column. Subtract $2$ from $2$, leaving $0$
  • $1 > 0$, so we have a zero in the $1$’s column
  • Putting it all together $42_{10} = 101010_2$
Practice Converting from decimal to binary

- **Example: 75**
  
  \[ \begin{array}{cccccc}
  64 & 32 & 16 & 8 & 4 & 2 & 1 \\
  \hline
  75 & & & & & & \\
  \end{array} \]

  \[ \begin{array}{ccccccc}
  & 1 & 0 & 0 & 1 & 0 & 1 \end{array} \]

- **Example: 224**
  
  \[ \begin{array}{cccccc}
  128 & 64 & 32 & 16 & 8 & 4 & 2 & 1 \\
  \hline
  224 & & & & & & & \\
  \end{array} \]

  \[ \begin{array}{ccccccc}
  & 1 & 1 & 0 & 0 & 0 & 0 \end{array} \]

  (Don’t Forget To Add The Zeros!)
Converting to and from Binary and Hex

• Mapping to and from binary and hex is more straightforward than the other conversions we looked at

• Binary to Hex:
  • Group the bits in sets of four
  • Convert each set of 4 bits to a hex digit
  • Example:
    • 10001101010101000101
    • 1000 1101
    • 0101 0100
    • 0101
    • 8 D 5 4 5
    • 0x8D545

• Hex to Binary
  • Convert each hex digit into 4 bits
  • 0xDEADBEEF
  • 1101 1110 1010 1101 1011 1110 1110 1111
Practice Problems (Answers Next Slide)

• Convert 0xFACE to binary

• Convert 011101101001110001₂ to hexadecimal
Practice Problems

• Convert 0xFACE to binary

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

• Convert 011101101001110001₂ to hexadecimal

0x1D A71
# Decimal / Binary / Hex Cheat Sheet

<table>
<thead>
<tr>
<th>Decimal</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>
Boolean Algebra

• Developed by George Boole in 19th Century
  • Algebraic representation of logic
  • Encode “True” as 1 and “False” as 0

<table>
<thead>
<tr>
<th>And</th>
<th>Or</th>
</tr>
</thead>
<tbody>
<tr>
<td>A&amp;B = 1 when both A=1 and B=1</td>
<td>A</td>
</tr>
<tr>
<td>&amp;</td>
<td>0 1</td>
</tr>
<tr>
<td>0 0</td>
<td>0 0 1</td>
</tr>
<tr>
<td>1 0</td>
<td>1 1 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Not</th>
<th>Exclusive-Or (Xor)</th>
</tr>
</thead>
<tbody>
<tr>
<td>~A = 1 when A=0</td>
<td>A^B = 1 when either A=1 or B=1, but not both</td>
</tr>
<tr>
<td>~</td>
<td>0 1</td>
</tr>
<tr>
<td>0 1</td>
<td>0 0 1</td>
</tr>
<tr>
<td>1 0</td>
<td>1 1 0</td>
</tr>
</tbody>
</table>
### In General

- **Operate on Bit Vectors**
  - Operations applied bitwise

<table>
<thead>
<tr>
<th>AND</th>
<th>OR</th>
<th>XOR</th>
<th>NOT</th>
</tr>
</thead>
<tbody>
<tr>
<td>01101001 &amp; 01010101</td>
<td>01101001</td>
<td>01101001 ^ 01010101</td>
<td>~ 01010101</td>
</tr>
</tbody>
</table>

- **All of the properties of Boolean Algebra apply**
Bit-Level Operations in C

- Operations &, |, ~, ^ available in C
  - Apply to any “integral” data type
  - long, int, short, and char
  - View arguments as bit vectors
  - Arguments applied bit-wise

- Examples (for a char data type)
  - ~0x41  ->  0xBE
    - ~01000001  ->  10111110
  - 0x69  &  0x55  ->  0x41
    - 01101001  &  01010101  ->  01000001
  - 0x69  |  0x55  ->  0x7D
    - 01101001  |  01010101  ->  01111101
Try some Problems

• ~0x35  ->

• 0xD2 & 0x0F  ->

• 0x6A | 0xF0  ->
Try some Problems

• ~0x35  →  
  ~0011 0101  →  1100 1010  →  0xCA

• 0xD2 & 0x0F  →  
  1101 0010 & 0000 1111  →  0x202
  we “cleared” the upper 4 bits of 0xD2 to zero

• 0x6A | 0xF0  →  
  0110 1010 | 1111 0000  →  1111 1010  
  1111 1010  →  0xFA
  we “set” the upper 4 bits of 0x6A to one
Helpful Boolean Identities

• Where A is a Boolean value (either 0 or 1)
  • \( A \lor 0 = A \)
  • \( A \land 1 = A \)
  • \( A \lor A = A \)
  • \( A \land A = A \)
  • \( A \lor \neg A = 1 \)
  • \( 1 \lor A = 1 \)
  • \( A \land \neg A = 0 \)
  • \( 0 \land A = 0 \)
  • \( A \oplus A = 0 \)
Helpful Boolean Identities

• Where A and B are Boolean values (either 0 or 1)
  • Commutative Law:
    • \( A \lor B \equiv B \lor A \)
    • \( A \land B \equiv B \land A \)
  • Associative Law:
    • \( A \lor (B \lor C) \equiv (A \lor B) \lor C \)
    • \( A \land (B \land C) \equiv (A \land B) \land C \)
  • Distributive Law:
    • \( A \land (B \lor C) \equiv A \land B \lor A \land C \)
    • \( A \lor B \land C \equiv (A \lor B) \land (A \lor C) \)
  • De Morgan’s Theorem:
    • \( \neg (A \land B) \equiv \neg A \lor \neg B \)
    • \( \neg (A \lor B) \equiv \neg A \land \neg B \)
Logic Operations in C

• Contrast to logical operators
  • &&, ||, !
  • View 0 as “False”
  • Anything nonzero as “True”
  • Always returns 0 or 1
  • Early termination

• Examples (char data type)
  • !0x41 -> 0x00
  • !0x00 -> 0x01
  • !!0x41 -> 0x01
  • 0x69 && 0x55 -> 0x01
  • 0x69 || 0x55 -> 0x01
Word-Oriented Memory Organization

• Address Specify Byte locations
  • Address of the first byte in word
  • Addresses of successive words differ by 4 (32-bit) or 8 (64-bit) bytes
Byte Ordering

• So, how are the bytes within a multi-byte word ordered in memory

• Conventions
  • Big Endian: Sun, PowerPC Macs, Internet
    • Least significant byte has the highest address
  • Little Endian: x86, ARM processors running Android, iOS, and Windows
    • Least significant byte has the lowest address

<table>
<thead>
<tr>
<th>Addr</th>
<th>32-bit Words</th>
<th>64-bit Words</th>
<th>Bytes</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td></td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>0004</td>
<td>0</td>
<td>0</td>
<td>0001</td>
<td>0001</td>
</tr>
<tr>
<td>0008</td>
<td>0</td>
<td>0</td>
<td>0002</td>
<td>0002</td>
</tr>
<tr>
<td>0012</td>
<td></td>
<td>0</td>
<td>0003</td>
<td>0003</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td>0004</td>
<td>0004</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td>0005</td>
<td>0005</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td>0006</td>
<td>0006</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td>0007</td>
<td>0007</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0008</td>
<td>0008</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td>0009</td>
<td>0009</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td>0010</td>
<td>0010</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td>0011</td>
<td>0011</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td>0012</td>
<td>0012</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td>0013</td>
<td>0013</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td>0014</td>
<td>0014</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td>0015</td>
<td>0015</td>
</tr>
</tbody>
</table>
Byte Ordering Example

• Example
  • 4-byte value (say an integer) of 0x01234567 is located at memory address 0x100
  • This value exists in memory locations 0x100, 0x101, 0x102, 0x103

![Byte Ordering Diagram]

- Big Endian
- Little Endian