• Question: Big Endian
Bits

Electronic Implementation

- Easy to store with bistable elements
- Reliably transmitted on noisy and inaccurate wires

![Diagram showing voltage levels and bit representation](image)
Bits

- 101
- 5 in binary
- 5 if it’s an integer
- $+\infty$ if float
- 0x101 = 65 in dec
- A if it’s hex referring to ASCII
Bits + Context

- Integer
- Float
- Character
- Program instruction
- Memory Address

- Bytes
Background: Hexadecimal

- **0xA4**
  - 10100100 (binary)
  - 164 (decimal)

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

1. Convert 0x173A4C to binary

2. Convert 11 1100 1010 1101 1011 0011₂ to hex

3. Convert 7AF₁₆ to base-10

4. Convert 60₁₀ to base-16

5. Convert 600₁₀ to base-16
Byte-Oriented Memory Organization

Programs Refer to Virtual Addresses
- Conceptually very large array of bytes
- Actually implemented with hierarchy of different memory types
- System provides address space private to particular “process”
  - Program being executed
  - Program can clobber its own data, but not that of others

Compiler + Run-Time System Control Allocation
- Where different program objects should be stored
- All allocation within single virtual address space
Machine Words

Machine Has “Word Size”

- Nominal size of integer-valued data
  - Including addresses
- Most current machines use 32 bits (4 bytes) words
  - Limits addresses to 4GB
  - Becoming too small for memory-intensive applications
- High-end systems use 64 bits (8 bytes) words
  - Potential address space $\approx 1.8 \times 10^{19}$ bytes
  - x86-64 machines support 48-bit addresses: 256 Terabytes
- Machines support multiple data formats
  - Fractions or multiples of word size
  - Always integral number of bytes
Word-Oriented Memory Organization

Addresses Specify Byte Locations

- Address of first byte in word
- Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)
<table>
<thead>
<tr>
<th>C declaration</th>
<th>32-bit</th>
<th>64-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short int</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long int</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>long long int</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>char*</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>
Byte Ordering

- **Big Endian**: Sun, PPC, Internet
  - Least significant byte has highest address
- **Little Endian**: x86
  - Least significant byte has lowest address

**Example**

- Variable `x` has 4-byte representation `0x01234567`
- Address given by `&x` is `0x100`

<table>
<thead>
<tr>
<th>Big Endian</th>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>01</td>
<td>23</td>
<td>45</td>
<td>67</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Little Endian</th>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>67</td>
<td>45</td>
<td>23</td>
<td>01</td>
</tr>
</tbody>
</table>
Blame George Bool & Claude Shannon

- And
- Or
- Not
- Xor
General Boolean Algebras

Operate on Bit Vectors

- Operations applied bitwise

\[
\begin{align*}
01101001 & \quad 01101001 & \quad 01101001 \\
& \quad 01010101 & \quad | & \quad 01010101 & \quad ^ & \quad 01010101 & \quad \sim & \quad 01010101
\end{align*}
\]

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, char, unsigned
- View arguments as bit vectors
- Arguments applied bit-wise

Examples

- \(~0x41 \rightarrow \)
  \(~01000001_2 \rightarrow \)

- \(~0x00 \rightarrow \)
  \(~00000000_2 \rightarrow \)

- \(0x69 \& 0x55 \rightarrow \)
  \(01101001_2 \& 01010101_2 \rightarrow \)

- \(0x69 \mid 0x55 \rightarrow \)
  \(01101001_2 \mid 01010101_2 \rightarrow \)
Contrast: Logic Operations in C

Contrast to Logical Operators

- &&, ||, !
  - View 0 as “False”
  - Anything nonzero as “True”
  - Always return 0 or 1
  - Early termination

Examples

- !0x41  -->  0x00
- !0x00  -->  0x01
- !!0x41  -->  0x01
- 0x69  &&  0x55  -->  0x01
- 0x69  ||  0x55  -->  0x01