Bits, Bytes, and Integers

Topics
- Representing information as bits
- Bit-level manipulations
  - Boolean algebra
  - Expressing in C
- Representations of Integers
  - Basic properties and operations
  - Implications for C

Binary Representations

Base 2 Number Representation
- Represent 15213\textsubscript{10} as 11101101101101\textsubscript{2}
- Represent 1.20\textsubscript{10} as 1.0011001100110011[0011]...\textsubscript{2}
- Represent 1.5213 \times 10^4 as 1.1101101101101\textsubscript{2} X 2^{13}

Electronic Implementation
- Easy to store with bistable elements
- Reliably transmitted on noisy and inaccurate wires
Encoding Byte Values

Byte = 8 bits
- Binary 00000000₂ to 11111111₂
- Decimal: 0₁₀ to 255₁₀
  - First digit must not be 0 in C
- Hexadecimal 00₁₆ to FF₁₆
  - Base 16 number representation
  - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
  - Write FA1D37B₁₆ in C as 0xFA1D37B
    - Or 0xfa1d37b

<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>

Byte-Oriented Memory Organization

<table>
<thead>
<tr>
<th>00...0</th>
<th>FF...F</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 ~ 1.8 X 10^18 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)
Data Representations

Sizes of C Objects (in Bytes)

- C Data Type  Typical 32-bit  Intel IA32  x86-64
  - char       1         1         1
  - short      2         2         2
  - int        4         4         4
  - long       4         4         8
  - long long  8         8         8
  - float      4         4         4
  - double     8         8         8
  - long double  8      10/12   10/16
  - char *     4         4         8

» Or any other pointer

Byte Ordering

How should bytes within multi-byte word be ordered in memory?

Conventions

- Big Endian: Sun, PPC Mac, Internet
  - Least significant byte has highest address
- Little Endian: x86
  - Least significant byte has lowest address
Byte Ordering Example

Big Endian
- Least significant byte has highest address

Little Endian
- Least significant byte has lowest address

Example
- Variable \( x \) has 4-byte representation 0x01234567
- Address given by \&x is 0x100

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Code</th>
<th>Assembly Rendition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>01 23 45 67</td>
<td></td>
</tr>
<tr>
<td>0x101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x102</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x103</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Code</th>
<th>Assembly Rendition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>67 45 23 01</td>
<td></td>
</tr>
<tr>
<td>0x101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x102</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x103</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Reading Byte-Reversed Listings

Disassembly
- Text representation of binary machine code
- Generated by program that reads the machine code

Example Fragment

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Code</th>
<th>Assembly Rendition</th>
</tr>
</thead>
<tbody>
<tr>
<td>8048365:</td>
<td>5b</td>
<td>pop %ebx</td>
</tr>
<tr>
<td>8048366:</td>
<td>81 c3 ab 12 00 00</td>
<td>add $0x12ab,%ebx</td>
</tr>
<tr>
<td>804836c:</td>
<td>83 bb 28 00 00 00 00</td>
<td>cmpl $0x0,0x28(%ebx)</td>
</tr>
</tbody>
</table>

Deciphering Numbers
- Value: 0x12ab
- Pad to 32 bits: 0x000012ab
- Split into bytes: 00 00 12 ab
- Reverse: ab 12 00 00
Examining Data Representations

Code to Print Byte Representation of Data

- Casting pointer to unsigned char * creates byte array

```c
typedef unsigned char *pointer;

void show_bytes(pointer start, int len)
{
    int i;
    for (i = 0; i < len; i++)
        printf("0x%p\t0x%.2x\n", start+i, start[i]);
    printf("\n");
}
```

Printf directives:
-%p: Print pointer
-%x: Print Hexadecimal

show_bytes Execution Example

```c
int a = 15213;
printf("int a = 15213;\n");
show_bytes((pointer) &a, sizeof(int));
```

Result (Linux):

```c
int a = 15213;
0x11ffffcb8 0x6d
0x11ffffcb9 0x3b
0x11ffffcba 0x00
0x11ffffcbb 0x00
```
Representing Integers

\[
\begin{align*}
\text{int } A &= 15213; \\
\text{int } B &= -15213; \\
\text{long int } C &= 15213;
\end{align*}
\]

Decimal: 15213

Binary: 0011 1011 0110 1101

Hex: 3 B 6 D

---

Two’s complement representation (Covered later)

Representing Pointers

\[
\begin{align*}
\text{int } B &= -15213; \\
\text{int } *P &= \& B;
\end{align*}
\]

Different compilers & machines assign different locations to objects
Representing Strings

Strings in C

- Represented by array of characters
- Each character encoded in ASCII format
  - Standard 7-bit encoding of character set
  - Character “0” has code 0x30
    - Digit / has code 0x30+/
- String should be null-terminated
  - Final character = 0

Compatibility
- Byte ordering not an issue

Boolean Algebra

Developed by George Boole in 19th Century

- Algebraic representation of logic
  - Encode “True” as 1 and “False” as 0

And

- \( A \& B = 1 \) when both \( A=1 \) and \( B=1 \)

<table>
<thead>
<tr>
<th>( B )</th>
<th>( &amp; )</th>
<th>( A )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Or

- \( A \sqcup B = 1 \) when either \( A=1 \) or \( B=1 \)

<table>
<thead>
<tr>
<th>( B )</th>
<th>( \sqcup )</th>
<th>( A )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Not

- \( \neg A = 1 \) when \( A=0 \)

<table>
<thead>
<tr>
<th>( A )</th>
<th>( \neg )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Exclusive-Or (Xor)

- \( A \oplus B = 1 \) when either \( A=1 \) or \( B=1 \), but not both

<table>
<thead>
<tr>
<th>( B )</th>
<th>( \oplus )</th>
<th>( A )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
**Application of Boolean Algebra**

Applied to Digital Systems by Claude Shannon

- 1937 MIT Master’s Thesis
- Reason about networks of relay switches
  - Encode closed switch as 1, open switch as 0

Connection when

\[ A \& \sim B \lor \sim A \& B \]

\[ = A \oplus B \]

All of you should be familiar with this topic (ECE 301/331)

---

**General Boolean Algebras**

Operate on Bit Vectors

- Operations applied bitwise

\[
\begin{array}{cccc}
01101001 & 01101001 & 01101001 \\
\& 01010101 & 01010101 & ^{01010101} & \sim 01010101 \\
01000001 & 01111101 & 00111100 & 10101010 \\
\end{array}
\]

All of the Properties of Boolean Algebra Apply

---
Representing & Manipulating Sets

Representation
- Width \( w \) bit vector represents subsets of \( \{0, \ldots, w-1\} \)
- \( a_j = 1 \) if \( j \in A \)

\[
\begin{align*}
01101001 & \quad \{0, 3, 5, 6\} \\
76543210 & \\
01010101 & \quad \{0, 2, 4, 6\} \\
76543210 &
\end{align*}
\]

Operations
- \& Intersection \( 01000001 \quad \{0, 6\} \)
- | Union \( 01111101 \quad \{0, 2, 3, 4, 5, 6\} \)
- ^ Symmetric difference \( 00111100 \quad \{2, 3, 4, 5\} \)
- ~ Complement \( 10101010 \quad \{1, 3, 5, 7\} \)

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 (Char data type)
- \( ^{\sim}0x41 \rightarrow 0xBE \)
  \(-01000001_{\text{2}} \rightarrow 10111110_{\text{2}} \)
- \( ^{\sim}0x00 \rightarrow 0xFF \)
  \(-00000000_{\text{2}} \rightarrow 11111111_{\text{2}} \)
- \( 0x69 \& 0x55 \rightarrow 0x41 \)
  \(01101001_{\text{2}} \& 01010101_{\text{2}} \rightarrow 01000001_{\text{2}} \)
- \( 0x69 \mid 0x55 \rightarrow 0x7D \)
  \(01101001_{\text{2}} \mid 01010101_{\text{2}} \rightarrow 01111101_{\text{2}} \)
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 (char data type)

- `!0x41 --> 0x00`
- `!0x00 --> 0x01`
- `!!0x41 --> 0x01`
- `0x69 && 0x55 --> 0x01`
- `0x69 || 0x55 --> 0x01`
- `p && *p` (avoids null pointer access)

Shift Operations

Left Shift: \( x \ll y \)

- Shift bit-vector \( x \) left \( y \) positions
  - Throw away extra bits on left
  - Fill with 0’s on right

Right Shift: \( x \gg y \)

- Shift bit-vector \( x \) right \( y \) positions
  - Throw away extra bits on right
- Logical shift
  - Fill with 0’s on left
- Arithmetic shift
  - Replicate most significant bit on right

Undefined Behavior

- Shift amount < 0 or ≥ word size
Encoding Integers

Unsigned

\[ B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i \]

Two’s Complement

\[ B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i \]

- C short 2 bytes long

\[
\begin{array}{|c|c|c|c|}
\hline
\text{Decimal} & \text{Hex} & \text{Binary} \\
\hline
x & 15213 & 00111011 01101101 \\
\hline
y & -15213 & 11000100 10010011 \\
\hline
\end{array}
\]

Sign Bit

- For 2’s complement, most significant bit indicates sign
  - 0 for nonnegative
  - 1 for negative

Encoding Example (Cont.)

\[
\begin{array}{|c|c|c|}
\hline
\text{Weight} & 15213 & -15213 \\
\hline
1 & 1 & 1 & 1 & 1 \\
2 & 0 & 0 & 1 & 2 \\
4 & 1 & 4 & 0 & 0 \\
8 & 1 & 8 & 0 & 0 \\
16 & 0 & 0 & 1 & 16 \\
32 & 1 & 32 & 0 & 0 \\
64 & 1 & 64 & 0 & 0 \\
128 & 0 & 0 & 1 & 128 \\
256 & 1 & 256 & 0 & 0 \\
512 & 1 & 512 & 0 & 0 \\
1024 & 0 & 0 & 1 & 1024 \\
2048 & 1 & 2048 & 0 & 0 \\
4096 & 1 & 4096 & 0 & 0 \\
8192 & 1 & 8192 & 0 & 0 \\
16384 & 0 & 0 & 1 & 16384 \\
-32768 & 0 & 0 & 1 & -32768 \\
\hline
\text{Sum} & 15213 & -15213 \\
\hline
\end{array}
\]
Readings

Today’s lecture -- Chapter 2.1 (Information Storage)

Next class – Chapter 2.2 (Integer Representations)