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 \(15213_{10}\) as \(11101101101101_2\)
- Represent \(1.20_{10}\) as \(1.0011001100110011[0011]\ldots_2\)
- Represent \(1.5213 \times 10^4\) as \(1.11011011011012 \times 2^{13}\)

Electronic Implementation

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

Byte = 8 bits

- Binary: \(00000000_2\) to \(11111111_2\)
- Decimal: \(0_{10}\) to \(255_{10}\)
  - First digit must not be 0 in C
- Hexadecimal: \(00_{16}\) to \(FF_{16}\)
  - Base 16 number representation
  - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
  - Write \(\text{FA1D37B}_{16}\) in C as \(0x\text{FA1D37B}\)
    » 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

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>32-bit Words</th>
<th>64-bit Words</th>
<th>Bytes</th>
<th>Addr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr = 0000</td>
<td>Addr = 0000</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>Addr = 0004</td>
<td></td>
<td>0001</td>
<td></td>
</tr>
<tr>
<td>Addr = 0008</td>
<td>Addr = 0008</td>
<td>0002</td>
<td></td>
</tr>
<tr>
<td>Addr = 0012</td>
<td></td>
<td>0003</td>
<td></td>
</tr>
</tbody>
</table>
# Data Representations

## Sizes of C Objects (in Bytes)

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Typical 32-bit</th>
<th>Intel IA32</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>long long</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>8</td>
<td>8</td>
<td>10/12</td>
</tr>
<tr>
<td>char *</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

» 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>Big Endian</th>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>67</td>
<td>45</td>
<td>23</td>
<td>01</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):**

```
int a = 15213;
0x11fffffcb8 0x6d
0x11fffffcb9 0x3b
0x11fffffcca 0x00
0x11fffffcbc 0x00
```
Representing Integers

int A = 15213;
int B = -15213;
long int C = 15213;

Decimal: 15213
Binary: 0011 1011 0110 1101
Hex: 3 B 6 D

Two’s complement representation (Covered later)
Representing Pointers

```plaintext
int B = -15213;
int *P = &B;
```

<table>
<thead>
<tr>
<th></th>
<th>Sun P</th>
<th>IA32 P</th>
<th>x86-64 P</th>
</tr>
</thead>
<tbody>
<tr>
<td>2C</td>
<td>EF</td>
<td>D4</td>
<td>0C</td>
</tr>
<tr>
<td>FB</td>
<td>FF</td>
<td>F8</td>
<td>89</td>
</tr>
<tr>
<td>FF</td>
<td>FB</td>
<td>FF</td>
<td>EC</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>FF</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>FF</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>7F</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>00</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>00</td>
</tr>
</tbody>
</table>

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 \( i \) has code 0x30+i
- String should be null-terminated
  - Final character = 0

**Compatibility**

- Byte ordering not an issue

```c
char S[6] = "15213";
```
**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>&amp;</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

### Or

- **A|B = 1 when either A=1 or B=1**

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

### Not

- **~A = 1 when A=0**

<table>
<thead>
<tr>
<th>~</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

### Exclusive-Or (Xor)

- **A^B = 1 when either A=1 or B=1, but not both**

<table>
<thead>
<tr>
<th>^</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</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

\[ A \& \sim B \mid \sim A \& B \]

= \( A \wedge B \)
General Boolean Algebras

Operate on Bit Vectors

- Operations applied bitwise

\[
\begin{array}{ccc}
01101001 & 01101001 & 01101001 \\
\& 01010101 & | 01010101 & ^ 01010101 \\
01000001 & 01111101 & 00111100 \\
\end{array}
\]

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 $j \in A$

<table>
<thead>
<tr>
<th>Bit Vector</th>
<th>Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>01101001</td>
<td>{0, 3, 5, 6}</td>
</tr>
<tr>
<td>01010101</td>
<td>{0, 2, 4, 6}</td>
</tr>
</tbody>
</table>

Operations

- **&** Intersection
  - Bit Vector: 01000001
  - Set: \{0, 6\}
- **|** Union
  - Bit Vector: 01111101
  - Set: \{0, 2, 3, 4, 5, 6\}
- **^** Symmetric difference
  - Bit Vector: 00111100
  - Set: \{2, 3, 4, 5\}
- **~** Complement
  - Bit Vector: 10101010
  - Set: \{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)

- \( \sim 0x41 \rightarrow 0xBE \)
  \( \sim 01000001_2 \rightarrow 10111110_2 \)
- \( \sim 0x00 \rightarrow 0xFF \)
  \( \sim 00000000_2 \rightarrow 11111111_2 \)
- \( 0x69 \& 0x55 \rightarrow 0x41 \)
  \( 01101001_2 \& 01010101_2 \rightarrow 01000001_2 \)
- \( 0x69 \mid 0x55 \rightarrow 0x7D \)
  \( 01101001_2 \mid 01010101_2 \rightarrow 01111101_2 \)
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 \rightarrow 0x00
- \!0x00 \rightarrow 0x01
- \!!0x41 \rightarrow 0x01
- 0x69 \&\& 0x55 \rightarrow 0x01
- 0x69 || 0x55 \rightarrow 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 \( \geq \) word size
Integer C Puzzles

- Assume 32-bit word size, two’s complement integers
- For each of the following C expressions, either:
  - Argue that is true for all argument values
  - Give example where not true

- $x < 0 \implies (x \times 2) < 0$
- $ux \geq 0$
- $x \& 7 == 7 \implies (x \ll 30) < 0$
- $ux > -1$
- $x > y \implies -x < -y$
- $x \times x \geq 0$
- $x > 0 \&\& y > 0 \implies x + y > 0$
- $x \geq 0 \implies -x \leq 0$
- $x \leq 0 \implies -x \geq 0$
- $(x | -x) >> 31 == -1$
- $ux >> 3 == ux/8$
- $x >> 3 == x/8$
- $x \& (x-1) != 0$

Initialization

```c
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```
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

\[
\begin{array}{|c|c|c|}
\hline
\text{Decimal} & \text{Hex} & \text{Binary} \\
\hline
x & 15213 & 3B 6D 00111011 01101101 \\
\hline
y & -15213 & C4 93 11000100 10010011 \\
\hline
\end{array}
\]

For 2’s complement, most significant bit indicates sign

- 0 for nonnegative
- 1 for negative
### Encoding Example (Cont.)

x = 15213: 00111011 01101101  
y = -15213: 11000100 10010011

<table>
<thead>
<tr>
<th>Weight</th>
<th>15213</th>
<th>-15213</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>64</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>128</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>256</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>512</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1024</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2048</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4096</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8192</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>16384</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>-32768</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Sum 15213 -15213