Bits, Bytes, and Integers

CS 485: Systems Programming
Fall 2015

Instructor:
James Griffioen

Adapted from slides by R. Bryant and D. O’Hallaron  (http://csapp.cs.cmu.edu/public/instructors.html)
Overview: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
Binary Representations

3.3V
2.8V
0.5V
0.0V

0 1 0
Bits, Bytes, and Octets

Preliminaries: Number Bases
- *Decimal* – base 10, the normal way to write numbers
- *Binary* – base 2, numbers consists of 1’s and 0’s
- *Hexadecimal*
  - Base 16 number representation
  - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
    - Write FA1D37B₁₆ in C as
      - 0xFA1D37B
      - 0xfa1d37b
  - Each hexadecimal letter corresponds to 4 (binary) bits.

**Byte = 8 bits (usually)**
- Byte is the term commonly used in the context of machines
- An 8 bit byte has values of
  - Binary 00000000₂ to 11111111₂
  - Decimal: 0₁₀ to 255₁₀
  - Hexadecimal 0₀₁₆ to FF₁₆ (see below)

Octets = 8 bits (always)
- Octet is the term commonly used in the context of networks

How many bytes does it take to store the number:
- 1001100011101₂
- 131D₁₆
- 4893₁₀

<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>
Aside: How big is a Mega?

- $10^6$?
  - 1,000,000

- $2^{20}$?
  - 1,048,576

Networks are based on clock rates in Hz, therefore a Megabit of bandwidth is $10^6$ bits of data.

Computer memory is based on powers of 2, therefore a Megabyte of memory is $2^{20}$ bytes of memory.

Kilo, Giga, and Tera are similar.
Byte-Oriented Memory Organization

- Programs refer to (Virtual) Addresses
  - Memory is conceptually a (very large) array of bytes
  - Each running program gets a part of the memory
    - Programs can access their own memory region, but usually cannot write to (and thus “clobber”) other program’s memory
  - Virtual memory is implemented with a hierarchy of different memory types – usually a two-level hierarchy of memory + disk

- Compiler + Operating System control the allocation of memory
  - Decide where different parts of a program go in the program’s memory space. (compiler)
  - Determine where different programs should be stored in memory. (OS)
Machine Words

Each machine has a “word size”

- Nominal size of integer-valued data
  - Typically determines address size.
- Standard machines use 32 bit = (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)
# Data Representations

<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>10/12</td>
<td>10/16</td>
</tr>
<tr>
<td>pointer</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>
Byte Ordering

- How should bytes within a 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>
</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>
</tbody>
</table>
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></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x100</td>
<td>0x101</td>
<td>0x102</td>
<td>0x103</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></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x100</td>
<td>0x101</td>
<td>0x102</td>
<td>0x103</td>
</tr>
<tr>
<td></td>
<td>67</td>
<td>45</td>
<td>23</td>
<td>01</td>
</tr>
</tbody>
</table>
Representing Integers

```c
int A = 15213;
long int C = 15213;
```

```c
int B = -15213;
```

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

Two’s complement representation (Covered later)
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("%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;
0x11ffffffcb8 0x6d
0x11ffffffcb9 0x3b
0x11ffffffcba 0x00
0x11ffffffcbb 0x00
```
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 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
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
    - How do you know when you are at the end?
      - String are null-terminated
        - Final character = 0

- **Compatibility**
  - Byte ordering not an issue
Strings vs. Buffers

- Strings and Buffers can easily be confused
- They look alike in a C program
  - Example string definition: `char example_string[200]`
  - Example buffer definition: `char example_buffer[200]`

- A String
  - Is typically used to store “printable” phrases or sentences.
  - Uses a null character (‘\0’) to indicate the end of a string (implying that ‘\0’ cannot occur within a string).

- A Buffer
  - Is a term that is not explicitly defined by C, but is often used in the context of networking and operating system code.
  - Is an array of bytes used to store any binary values, not just printable ones.
  - Does not end with a null character (‘\0’) because the null character might be stored in one of the bytes of the buffer.
  - Requires an additional variable to hold the “current length” of the buffer (i.e., to tell how many bytes of the buffer currently have data values in them.)
Overview: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
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>A &amp; B</td>
<td>A</td>
</tr>
<tr>
<td>0 1</td>
<td>1 0 1</td>
</tr>
<tr>
<td>0 0 0</td>
<td>0 0 1</td>
</tr>
<tr>
<td>1 0 1</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>A ^ B</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>1 0</td>
<td>0 0 1</td>
</tr>
<tr>
<td>1 0</td>
<td>1 1 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&~B | ~A&B

= A^B
General Boolean Algebras

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

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

- **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 \( j \in A \)

  - 01101001 \( \{0, 3, 5, 6\} \)
  - 76543210

  - 01010101 \( \{0, 2, 4, 6\} \)
  - 76543210

- **Operations**
  - &  Intersection  01000001 \( \{0, 6\} \)
  - |  Union  01111101 \( \{0, 2, 3, 4, 5, 6\} \)
  - ^  Symmetric difference  00111100 \( \{2, 3, 4, 5\} \)
  - ~  Complement  10101010 \( \{1, 3, 5, 7\} \)
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)**
  - \(~0x41 \rightarrow 0xBE\)
    - \(~01000001_2 \rightarrow 10111110_2\)
  - \(~0x00 \rightarrow 0xFF\)
    - \(~00000000_2 \rightarrow 11111111_2\)
  - \(0x69 \& 0x55 \rightarrow 0x41\)
    - \(01101001_2 \& 01010101_2 \rightarrow 01000001_2\)
  - \(0x69 \mid 0x55 \rightarrow 0x7D\)
    - \(01101001_2 \mid 01010101_2 \rightarrow 01111101_2\)
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 ➔ 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
Overview: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
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

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>15213</td>
<td>3B 6D 00111011 01101101</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>C4 93 11000100 10010011</td>
</tr>
</tbody>
</table>

Sign Bit

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

**Sum**

- \( 15213 \)
- \( -15213 \)
Numeric Ranges

- **Unsigned Values**
  - $UMin = 0$
    - 000...0
  - $UMax = 2^w - 1$
    - 111...1

- **Two's Complement Values**
  - $TMin = -2^{w-1}$
    - 100...0
  - $TMax = 2^{w-1} - 1$
    - 011...1

- **Other Values**
  - Minus 1
    - 111...1

Values for $W = 16$

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>65535</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>Tmax</td>
<td>32767</td>
<td>7F FF</td>
<td>01111111 11111111</td>
</tr>
<tr>
<td>Tmin</td>
<td>-32768</td>
<td>80 00</td>
<td>10000000 00000000</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>
Values for Different Word Sizes

<table>
<thead>
<tr>
<th>W</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>255</td>
<td>65,535</td>
<td>4,294,967,295</td>
<td>18,446,744,073,709,551,615</td>
</tr>
<tr>
<td>Tmax</td>
<td>127</td>
<td>32,767</td>
<td>2,147,483,647</td>
<td>9,223,372,036,854,775,807</td>
</tr>
<tr>
<td>Tmin</td>
<td>-128</td>
<td>-32,768</td>
<td>-2,147,483,648</td>
<td>-9,223,372,036,854,775,808</td>
</tr>
</tbody>
</table>

- **Observations**
  - $|Tmin| = Tmax + 1$
    - Asymmetric range
  - $U_{Max} = 2 \times Tmax + 1$

- **C Programming**
  - `#include <limits.h>`
  - Declares constants, e.g.,
    - `ULONG_MAX`
    - `LONG_MAX`
    - `LONG_MIN`
  - Values platform specific
Unsigned & Signed Numeric Values

<table>
<thead>
<tr>
<th>$\chi$</th>
<th>$B2U(\chi)$</th>
<th>$B2T(\chi)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>-8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td>-7</td>
</tr>
<tr>
<td>1010</td>
<td>10</td>
<td>-6</td>
</tr>
<tr>
<td>1011</td>
<td>11</td>
<td>-5</td>
</tr>
<tr>
<td>1100</td>
<td>12</td>
<td>-4</td>
</tr>
<tr>
<td>1101</td>
<td>13</td>
<td>-3</td>
</tr>
<tr>
<td>1110</td>
<td>14</td>
<td>-2</td>
</tr>
<tr>
<td>1111</td>
<td>15</td>
<td>-1</td>
</tr>
</tbody>
</table>

- **Equivalence**
  - Same encodings for nonnegative values

- **Uniqueness**
  - Every bit pattern represents unique integer value
  - Each representable integer has unique bit encoding

- **Can Invert Mappings**
  - $U2B(\chi) = B2U^{-1}(x)$
    - Bit pattern for unsigned integer
  - $T2B(\chi) = B2T^{-1}(x)$
    - Bit pattern for two’s comp integer