Byte Ordering

Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
Memory Model

▪ Physical memory
  • DRAM chips can read/write 4, 8, 16 bits
  • DRAM modules can read/write 64 bits

▪ Programmer’s view of memory
  • Conceptually, a very large array of bytes
  • Stored-program computers: keeps program codes and data in memory
  • Running programs share the physical memory
  • OS handles memory allocation and management
Machine Words

- Each computer has a “word size”
  - Nominal size of integer-valued data
    - Including addresses (= pointer size)
  - Until recently, most machines used 32-bit (4-byte) words
    - Limits addresses to 4 GB
    - Becoming too small for memory-intensive applications
  - Increasingly, machines have 64-bit (8-byte) word size
    - Potential address space $\approx 18.4 \times 10^{18}$ bytes (18 EB)
    - x86-64 machines support 48-bit addresses: 256 TB
  - Machines support multiple data formats
    - Fractions or multiples of word size
    - Always integral number of bytes
Word-level Memory Access

- Addresses specify byte locations
  - Address of first byte in word
  - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)
  - Usually, addresses should be aligned to the word boundary
# Data Types in C

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Typical 32-bit</th>
<th>Typical 64-bit</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>8</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>-</td>
<td>-</td>
<td>10/16</td>
</tr>
<tr>
<td>pointer</td>
<td>4</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>
Byte Ordering

▪ How are the bytes within a multi-byte word ordered in memory?

▪ Conventions
  • Big endian: Sun, PowerPC Mac, Internet
  • Little endian: Intel x86, ARM running Android & iOS

▪ Note:
  • Alpha and PowerPC can run in either mode, with the byte ordering convention determined when the chip is powered up
  • Problem when the binary data is communicated over a network between different machines
Big vs. Little Endian

- **Big endian**
  - Least significant byte has highest address

- **Little endian**
  - Least significant byte has lowest address
Example 1

- **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>cmp1 $0x0,0x28(%ebx)</td>
</tr>
</tbody>
</table>

- **Deciphering numbers:**
  
  Value: 0x12ab
  Pad to 32 bits: 0x0000012ab
  Split into bytes: 00 00 12 ab
  Reverse: ab 12 00 00
Example 2

- What is the output of this program?
  - Solaris/SPARC:  
  - Linux/x86: 

```c
#include <stdio.h>

union {
    int i;
    unsigned char c[4];
} u;

int main () {
    u.i = 0x12345678;
    printf ("%x %x %x %x\n",
            u.c[0], u.c[1], u.c[2], u.c[3]);
}
```
Representing Integers

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

\[
\begin{array}{c}
\text{IA32, x86-64 A} \\
\begin{array}{c}
6D \\
3B \\
00 \\
00
\end{array} \\
\text{Sun A} \\
\begin{array}{c}
00 \\
00 \\
3B \\
6D
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\text{IA32, x86-64 B} \\
\begin{array}{c}
93 \\
C4 \\
FF \\
FF
\end{array} \\
\text{Sun B} \\
\begin{array}{c}
FF \\
FF \\
C4 \\
93
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\text{IA32 C} \\
\begin{array}{c}
6D \\
3B \\
00 \\
00
\end{array} \\
\text{x86-64 C} \\
\begin{array}{c}
00 \\
00 \\
00 \\
00
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\text{Sun C} \\
\begin{array}{c}
00 \\
00 \\
3B \\
6D
\end{array}
\end{array}
\]

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

Two's complement representation
Representing Pointers

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

Different compilers & machines assign different locations to objects
Even get different results each time run program
**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 = 0x00

- **Compatibility**
  - Byte ordering not an issue
Summary

- It's all about bits & bytes
  - Numbers, programs, text, …

- Different machines follow different conventions
  - Word size
  - Byte ordering
  - Representations (integer, floating-point)

- When programming, be aware of
  - Type casting & mixed signed/unsigned expressions
  - Overflow
  - Error propagation
  - Byte ordering