Bits, Bytes, and Integers

15-213/18-243: Introduction to Computer Systems
2nd Lecture, 13 January 2010

Instructors:
Bill Nace and Gregory Kesden

The course that gives CMU its “Zip”!
Last Time: Course Overview

Course Theme:

Abstraction Is Good But Don’t Forget Reality

5 Great Realities
- Ints are not Integers, Floats are not Reals
- You’ve Got to Know Assembly
- Memory Matters
- There’s more to performance than asymptotic complexity
- Computers do more than execute programs

Administrative / Logistics details
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 $111011011011012$
  - Represent $1.20_{10}$ as $1.0011001100110011[0011]...2$
  - Represent $1.5213 \times 10^4$ (decimal) as $1.11011011011011012 \times 2^{13}$

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

![Graph showing binary voltage levels](image-url)
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 $FA1D37B_{16}$ in C as $0xFA1D37B$
      - Or $0x$fa1d37b
      - Or $0Xfa1d37b$ /* don’t do this */
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 ≈ $1.8 \times 10^{19}$ 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

### 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>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 \( 0x100 \)
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</td>
<td>cmp %ebx $0x000028(%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;
0x11fffffcb8  0x6d
0x11fffffcb9  0x3b
0x11fffffcb9  0x00
0x11fffffcb9  0x00
```
Representing Integers

```plaintext
int A = 15213;
```

```plaintext
int B = -15213;
```

```plaintext
long int C = 15213;
```

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

Two’s complement representation (Covered later)
Representing Pointers

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

Different compilers & machines assign different locations to objects

<table>
<thead>
<tr>
<th>Sun</th>
<th>IA32</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>EF</td>
<td>D4</td>
<td>0C</td>
</tr>
<tr>
<td>FF</td>
<td>F8</td>
<td>89</td>
</tr>
<tr>
<td>FB</td>
<td>FF</td>
<td>EC</td>
</tr>
<tr>
<td>2C</td>
<td>BF</td>
<td>FF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7F</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00</td>
</tr>
</tbody>
</table>
### 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] = "18243";
```
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
- \( A \mid B = 1 \) 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
- \( \sim A = 1 \) when \( A=0 \)

<table>
<thead>
<tr>
<th>( \sim )</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></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&~B | ~A&B
General Boolean Algebras

- Operate on Bit Vectors
  - Operations applied bitwise

\[
\begin{array}{ccc}
01101001 & 01101001 & 01101001 \\
& 01010101 & | 01010101 & ^ 01010101 & \sim 01010101
\end{array}
\]

- All of the Properties of Boolean Algebra Apply
Representing & Manipulating Sets

- **Representation**
  - Width $w$ bit vector represents subsets of $\{0, ..., w-1\}$
  - $a_j = 1$ if $j \in A$
    - $01101001 \ { 0, 3, 5, 6 }$
    - $01010101 \ { 0, 2, 4, 6 }$

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

- **Right Shift:** \( x >> y \)
  - 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

<table>
<thead>
<tr>
<th>Argument ( x )</th>
<th>( 01100010 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( &lt;&lt; 3 )</td>
<td>( 00010000 )</td>
</tr>
<tr>
<td>( \text{Log.} &gt;&gt; 2 )</td>
<td>( 00011000 )</td>
</tr>
<tr>
<td>( \text{Arith.} &gt;&gt; 2 )</td>
<td>( 00011000 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument ( x )</th>
<th>( 10100010 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( &lt;&lt; 3 )</td>
<td>( 00010000 )</td>
</tr>
<tr>
<td>( \text{Log.} &gt;&gt; 2 )</td>
<td>( 00101000 )</td>
</tr>
<tr>
<td>( \text{Arith.} &gt;&gt; 2 )</td>
<td>( 11101000 )</td>
</tr>
</tbody>
</table>

**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

Initialization

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

- $x < 0 \Rightarrow (x*2 < 0)$
- $ux >= 0$
- $x & 7 == 7 \Rightarrow (x<<30 < 0)$
- $ux > -1$
- $x > y \Rightarrow -x < -y$
- $x * x >= 0$
- $x > 0 && y > 0 \Rightarrow x + y > 0$
- $x >= 0 \Rightarrow -x <= 0$
- $x <= 0 \Rightarrow -x >= 0$
- $(x|-x)>>31 == -1$
- $ux >> 3 == ux/8$
- $x >> 3 == x/8$
- $x & (x-1) != 0$
Encoding Integers

**Unsigned**

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

**Two’s Complement**

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

- For 2 byte long short
  - `short int x = 15213;`
  - `short int y = -15213;`

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

**short int x = 15213:** 00111011 01101101  
**short int 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>

<table>
<thead>
<tr>
<th>Sum</th>
<th>15213</th>
<th>-15213</th>
</tr>
</thead>
<tbody>
<tr>
<td>15213</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>-15213</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>