15-213
“The Class That Gives CMU Its Zip!”

Bits and Bytes
January 15, 2004

Topics
- Why bits?
- Representing information as bits
  - Binary/Hexadecimal
  - Byte representations
    » numbers
    » characters and strings
    » Instructions
- Bit-level manipulations
  - Boolean algebra
  - Expressing in C
Why Don’t Computers Use Base 10?

Base 10 Number Representation
- That’s why fingers are known as “digits”
- Natural representation for financial transactions
  - Floating point number cannot exactly represent $1.20$
- Even carries through in scientific notation
  - $15.213 \times 10^3$ (1.5213e4)

Implementing Electronically
- Hard to store
  - ENIAC (First electronic computer) used 10 vacuum tubes / digit
  - IBM 650 used 5+2 bits (1958, successor to IBM’s Personal Automatic Computer, PAC from 1956)
- Hard to transmit
  - Need high precision to encode 10 signal levels on single wire
- Messy to implement digital logic functions
  - Addition, multiplication, etc.
Binary Representations

Base 2 Number Representation
- Represent $15213_{10}$ as $11101101101101_2$
- Represent $1.20_{10}$ as $1.0011001100110011[0011]..._2$
- Represent $1.5213 \times 10^4$ as $1.1101101101101_2 \times 2^{13}$

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

![Binary Representation Diagram]
Byte-Oriented Memory Organization

Programs Refer to Virtual Addresses
- Conceptually very large array of bytes
- Actually implemented with hierarchy of different memory types
  - SRAM, DRAM, disk
  - Only allocate for regions actually used by program
- In Unix and Windows NT, 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
- Multiple mechanisms: static, stack, and heap
- In any case, all allocation within single virtual address space
Encoding Byte Values

**Byte = 8 bits**

- **Binary** 00000000₂ to 11111111₂
- **Decimal:** 0₁₀ to 255₁₀
  - First digit must not be 0 in C
- **Octal:** 000₈ to 0377₈
  - Use leading 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

<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>
Literary Hex

Common 8-byte hex filler:

- 0xdeadbeef
- Can you think of other 8-byte fillers?
Machine Words

Machine Has “Word Size”

- Nominal size of integer-valued data
  - Including addresses
- Most current machines are 32 bits (4 bytes)
  - Limits addresses to 4GB
  - Becoming too small for memory-intensive applications
- High-end systems are 64 bits (8 bytes)
  - Potential address space ≈ 1.8 X 10^{19} bytes
- 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>
<tr>
<td></td>
<td></td>
<td>0004</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0005</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0006</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0007</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0008</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0009</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0010</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0011</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0012</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0013</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0014</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0015</td>
<td></td>
</tr>
</tbody>
</table>
# Data Representations

## Sizes of C Objects (in Bytes)

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Alpha (RIP)</th>
<th>Typical 32-bit</th>
<th>Intel IA32</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long int</td>
<td>8</td>
<td>4</td>
<td>4</td>
</tr>
<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>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>8</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

» Or any other pointer
Byte Ordering

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

Conventions

- Sun’s, Mac’s are “Big Endian” machines
  - Least significant byte has highest address
- Alphas, PC’s are “Little Endian” machines
  - 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>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>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</td>
<td>cmpl $0x0,0x28 (%ebx)</td>
</tr>
</tbody>
</table>

Deciphering Numbers

- Value: 0x12ab
- Pad to 4 bytes: 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;
0x11ffffffcb8 0x6d
0x11ffffffcb9 0x3b
0x11ffffffcba 0x00
0x11ffffffcbb 0x00
```
 Representing Integers 

\[
\text{int } A = 15213; \\
\text{int } B = -15213; \\
\text{long int } C = 15213; \\
\]

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

- Two’s complement representation (Covered next lecture)
Representing Pointers

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

### Alpha Address

<table>
<thead>
<tr>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1FFFFFCAC0</td>
<td>0001 1111 1111 1111 1111 1111 1100 1010 0000</td>
</tr>
</tbody>
</table>

### Sun Address

<table>
<thead>
<tr>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFFFFFFB2C</td>
<td>1110 1111 1111 1111 1111 1111 1011 0010 1100</td>
</tr>
</tbody>
</table>

### Linux Address

<table>
<thead>
<tr>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFFF8D4</td>
<td>1011 1111 1111 1111 1111 1000 1101 0100</td>
</tr>
</tbody>
</table>

Different compilers & machines assign different locations to objects.
Representing Floats

Float \( F = 15213.0; \)

<table>
<thead>
<tr>
<th>Linux/Alpha ( F )</th>
<th>Sun ( F )</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>46</td>
</tr>
<tr>
<td>B4</td>
<td>6D</td>
</tr>
<tr>
<td>6D</td>
<td>B4</td>
</tr>
<tr>
<td>46</td>
<td>00</td>
</tr>
</tbody>
</table>

IEEE Single Precision Floating Point Representation

Hex: \[ \begin{array}{cccccccc}
4 & 6 & 6 & D & B & 4 & 0 & 0 \\
\end{array} \]

Binary: \[ \begin{array}{cccccccccccc}
0100 & 0110 & 0110 & 1101 & 1011 & 0100 & 0000 & 0000 \\
\end{array} \]

15213: \[ \begin{array}{cccccccc}
1110 & 1101 & 1011 & 01 \\
\end{array} \]

Not same as integer representation, but consistent across machines

Can see some relation to integer representation, but not obvious
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
- Text files generally platform independent
  - Except for different conventions of line termination character(s)!
    - Unix (‘\n’ = 0x0a = ^J)
    - Mac (‘\r’ = 0x0d = ^M)
    - DOS and HTTP (‘\r\n’ = 0x0d0a = ^M^J)

```
char S[6] = "15213";
```
Machine-Level Code Representation

Encode Program as Sequence of Instructions

- Each simple operation
  - Arithmetic operation
  - Read or write memory
  - Conditional branch
- Instructions encoded as bytes
  - Alpha’s, Sun’s, Mac’s use 4 byte instructions
    » Reduced Instruction Set Computer (RISC)
  - PC’s use variable length instructions
    » Complex Instruction Set Computer (CISC)
- Different instruction types and encodings for different machines
  - Most code not binary compatible

Programs are Byte Sequences Too!
int sum(int x, int y)
{
    return x+y;
}

- For this example, Alpha & Sun use two 4-byte instructions
  - Use differing numbers of instructions in other cases
- PC uses 7 instructions with lengths 1, 2, and 3 bytes
  - Same for NT and for Linux
  - NT / Linux not fully binary compatible

Different machines use totally different instructions and encodings
# 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></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\midB = 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>0</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

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

= \( A \wedge \overline{B} \)
Integer Algebra

Integer Arithmetic

- \langle \mathbb{Z}, +, *, -, 0, 1 \rangle forms a “ring”
- Addition is “sum” operation
- Multiplication is “product” operation
- – is additive inverse
- 0 is identity for sum
- 1 is identity for product
Boolean Algebra

- $\langle \{0,1\}, |, &, \sim, 0, 1 \rangle$ forms a “Boolean algebra”
- Or is “sum” operation
- And is “product” operation
- $\sim$ is “complement” operation (not additive inverse)
- 0 is identity for sum
- 1 is identity for product
Boolean Algebra \( \approx \) Integer Ring

- **Commutativity**
  
  \[
  \begin{align*}
  A \lor B &= B \lor A \\
  A \land B &= B \land A
  \end{align*}
  \]

  \[
  \begin{align*}
  A + B &= B + A \\
  A \cdot B &= B \cdot A
  \end{align*}
  \]

- **Associativity**
  
  \[
  \begin{align*}
  (A \lor B) \lor C &= A \lor (B \lor C) \\
  (A \land B) \land C &= A \land (B \land C)
  \end{align*}
  \]

  \[
  \begin{align*}
  (A + B) + C &= A + (B + C) \\
  (A \cdot B) \cdot C &= A \cdot (B \cdot C)
  \end{align*}
  \]

- **Product distributes over sum**
  
  \[
  \begin{align*}
  A \land (B \lor C) &= (A \land B) \lor (A \land C) \\
  A \cdot (B + C) &= A \cdot B + A \cdot C
  \end{align*}
  \]

- **Sum and product identities**
  
  \[
  \begin{align*}
  A \lor 0 &= A \\
  A \land 1 &= A \\
  A + 0 &= A \\
  A \cdot 1 &= A
  \end{align*}
  \]

- **Zero is product annihilator**
  
  \[
  \begin{align*}
  A \land 0 &= 0 \\
  A \cdot 0 &= 0
  \end{align*}
  \]

- **Cancellation of negation**
  
  \[
  \begin{align*}
  \neg (\neg A) &= A \\
  -(\neg A) &= A
  \end{align*}
  \]
Boolean Algebra ≠ Integer Ring

- **Boolean: Sum distributes over product**
  \[ A \lor (B \land C) = (A \lor B) \land (A \lor C) \quad A + (B \land C) \neq (A + B) \land (B + C) \]

- **Boolean: Idempotency**
  \[ A \lor A = A \quad A + A \neq A \]
  “A is true” or “A is true” = “A is true”
  \[ A \land A = A \quad A \land A \neq A \]

- **Boolean: Absorption**
  \[ A \lor (A \land B) = A \quad A + (A \land B) \neq A \]
  “A is true” or “A is true and B is true” = “A is true”
  \[ A \land (A \lor B) = A \quad A \land (A + B) \neq A \]

- **Boolean: Laws of Complements**
  \[ A \lor \neg A = 1 \quad A + \neg A \neq 1 \]
  “A is true” or “A is false”

- **Ring: Every element has additive inverse**
  \[ A \lor \neg A \neq 0 \quad A + \neg A = 0 \]
## Boolean Ring

- \( \{0,1\}, ^\land, ^\lor, 0, 1 \)
- Identical to integers mod 2
- \( 1 \) is identity operation: \( 1 (A) = A \)
  \[ A ^\land A = 0 \]

### Property

<table>
<thead>
<tr>
<th>Property</th>
<th>Boolean Ring</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commutative sum</td>
<td>( A ^\land B = B ^\land A )</td>
</tr>
<tr>
<td>Commutative product</td>
<td>( A &amp; B = B &amp; A )</td>
</tr>
<tr>
<td>Associative sum</td>
<td>( (A ^\land B) ^\lor C = A ^\land (B ^\lor C) )</td>
</tr>
<tr>
<td>Associative product</td>
<td>( (A &amp; B) &amp; C = A &amp; (B &amp; C) )</td>
</tr>
<tr>
<td>Prod. over sum</td>
<td>( A &amp; (B ^\lor C) = (A &amp; B) ^\land (B &amp; C) )</td>
</tr>
<tr>
<td>0 is sum identity</td>
<td>( A ^\land 0 = A )</td>
</tr>
<tr>
<td>1 is prod. identity</td>
<td>( A &amp; 1 = A )</td>
</tr>
<tr>
<td>0 is product annihilator</td>
<td>( A &amp; 0 = 0 )</td>
</tr>
<tr>
<td>Additive inverse</td>
<td>( A ^\land A = 0 )</td>
</tr>
</tbody>
</table>
Relations Between Operations

DeMorgan’s Laws

- Express \& in terms of |, and vice-versa
  - \[ A \& B = \neg (\neg A \mid \neg B) \]
    - A and B are true if and only if neither A nor B is false
  - \[ A \mid B = \neg (\neg A \& \neg B) \]
    - A or B are true if and only if A and B are not both false

Exclusive-Or using Inclusive Or

- \[ A \uparrow B = (\neg A \& B) \mid (A \& \neg B) \]
  - Exactly one of A and B is true
- \[ A \uparrow B = (A \mid B) \& \neg (A \& B) \]
  - Either A is true, or B is true, but not both
General Boolean Algebras

Operate on Bit Vectors

- Operations applied bitwise

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

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 \( j \in A \)

```plaintext
01101001 \{ 0, 3, 5, 6 \}
76543210

01010101 \{ 0, 2, 4, 6 \}
76543210
```

Operations

- & Intersection \( \{ 0, 6 \} \)
- | Union \( \{ 0, 2, 3, 4, 5, 6 \} \)
- ^ Symmetric difference \( \{ 2, 3, 4, 5 \} \)
- ~ Complement \( \{ 1, 3, 5, 7 \} \)
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)
- \(~0\times41\) --> \(0\timesBE\)
  \(~01000001_2\) --> \(10111110_2\)
- \(~0\times00\) --> \(0\timesFF\)
  \(~00000000_2\) --> \(11111111_2\)
- \(0\times69 \& 0\times55\) --> \(0\times41\)
  \(01101001_2 \& 01010101_2\) --> \(01000001_2\)
- \(0\times69 \mid 0\times55\) --> \(0\times7D\)
  \(01101001_2 \mid 01010101_2\) --> \(01111101_2\)
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)

- `!0\x41  -->  0\x00`
- `!0\x00  -->  0\x01`
- `!!0\x41  -->  0\x01`

- `0\x69  &&  0\x55  -->  0\x01`
- `0\x69  ||  0\x55  -->  0\x01`
- `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
  - Useful with two’s complement integer representation

<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>Log. &gt;&gt; 2</td>
<td>( 00011000 )</td>
</tr>
<tr>
<td>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>Log. &gt;&gt; 2</td>
<td>( 00101000 )</td>
</tr>
<tr>
<td>Arith. &gt;&gt; 2</td>
<td>( 11101000 )</td>
</tr>
</tbody>
</table>
Cool Stuff with Xor

- Bitwise Xor is form of addition
- With extra property that every value is its own additive inverse
  \[ A \oplus A = 0 \]

```c
void funny(int *x, int *y)
{
    *x = *x ^ *y;    /* #1 */
    *y = *x ^ *y;    /* #2 */
    *x = *x ^ *y;    /* #3 */
}
```
More Fun with Bitvectors

Bit-board representation of chess position:

```c
unsigned long long blk_king, whit_king, whit_rook_mv2, ...;

wht_king = 0x000000000000001000ull;
blk_king  = 0x0004000000000000ull;
wht_rook_mv2 = 0x10ef101010101010ull;
...

/*
 * Is black king under attach from
 * white rook ?
*/
if (blk_king & whit_rook_mv2)
    printf("Yes\n");
```
More Bitvector Magic

Count the number of 1’s in a word

MIT Hackmem 169:

```c
int bitcount(unsigned int n)
{
    unsigned int tmp;

    tmp = n - ((n >> 1) & 0333333333333)
         - ((n >> 2) & 0111111111111);
    return ((tmp + (tmp >> 3)) & 030707070707)%63;
}
```
Some Other Uses for Bitvectors

Representation of small sets

Representation of polynomials:
- Important for error correcting codes
- Arithmetic over finite fields, say GF(2^n)
- Example 0x15213 : \(x^{16} + x^{14} + x^{12} + x^9 + x^4 + x + 1\)

Representation of graphs
- A ‘1’ represents the presence of an edge

Representation of bitmap images, icons, cursors, ...
- Exclusive-or cursor patent

Representation of Boolean expressions and logic circuits
Summary of the Main Points

It’s All About Bits & Bytes

- Numbers
- Programs
- Text

Different Machines Follow Different Conventions for

- Word size
- Byte ordering
- Representations

Boolean Algebra is the Mathematical Basis

- Basic form encodes “false” as 0, “true” as 1
- General form like bit-level operations in C
  - Good for representing & manipulating sets