Topics of this Slideset

There are 10 kinds of people in the world: those who understand binary, and those who don’t!

- Why bits?
- Representing information as bits
  - Binary and hexadecimal
  - Byte representations: numbers, characters, strings, instructions
- Bit level manipulations
  - Boolean algebra
  - C constructs
Why Not 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

$$1.5213 \times 10^4$$

**Implementing Electronically**

- 10 different values are hard to store. ENIAC (First electronic computer) used 10 vacuum tubes / digit
- They’re 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 $1.0011001100110011[0011] \ldots_2$
- 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.
**Fact:** Whatever you plan to store on a computer ultimately has to be represented as a collection of bits.

That’s true whether it’s integers, reals, characters, strings, data structures, instructions, pictures, videos, etc.

In a sense the representation is *arbitrary*. The representation is just a *mapping from the domain onto a finite set of bit strings*.

But some representations are better than others. **Why would that be?** Hint: **what operations do you want to support?**
Fact: If you are going to represent any type in $k$ bits, you can only represent $2^k$ different values. There are exactly as many integers as floats on IA32.

Fact: The same bit string can represent an integer (signed or unsigned), float, character string, list of instructions, etc. depending on the context.
Programs Refer to Virtual Addresses

- Conceptually very large array of bytes.
- Actually implemented with hierarchy of different memory types.
  - SRAM, DRAM, disk.
  - Only allocate storage for regions actually used by program.
- In Unix and Windows NT, address space private to particular “process.”
  - Encapsulates the program being executed.
  - Program can clobber its own data, but not that of others.

Compiler and Run-Time System Control Allocation

- Where different program objects should be stored.
- Multiple storage mechanisms: static, stack, and heap.
- In any case, all allocation within single virtual address space.
**Encoding Byte Values**

**Byte = 8 bits**
Which can be represented in various forms:

- Binary: \(00000000_2\) to \(11111111_2\)
- Decimal: \(0_{10}\) to \(255_{10}\)
- 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 \(0xfa1d37b\)

<table>
<thead>
<tr>
<th>Hex</th>
<th>Dec</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>
Machines generally have a specific “word size.”

- It’s the nominal size of integer-valued data, including addresses.
- Most current machines run 64-bit software (8 bytes).
  - 32-bit software limits addresses to 4GB.
  - Becoming too small for memory-intensive applications.
- All x86 current hardware systems are 64 bits (8 bytes).
  - Potentially address around $1.8 \times 10^{19}$ bytes.
- Machines support multiple data formats.
  - Fractions or multiples of word size.
  - Always integral number of bytes.
- X86-hardware systems operate in 16, 32, and 64 bits modes.
  - Initially starts in 286 mode, which is 16-bit.
  - Under programmer control, 32- and 64-bit modes are enabled.
Word-Oriented Memory Organization

Addresses Specify Byte Locations

- Which is the address of the 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></td>
<td>0000</td>
</tr>
<tr>
<td>Addr: 0004</td>
<td></td>
<td></td>
<td>0001</td>
</tr>
<tr>
<td>Addr: 0008</td>
<td></td>
<td></td>
<td>0002</td>
</tr>
<tr>
<td>Addr: 0012</td>
<td></td>
<td></td>
<td>0003</td>
</tr>
<tr>
<td></td>
<td>Addr: 0000</td>
<td></td>
<td>0004</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0005</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0006</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0007</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0008</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0009</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0011</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0012</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0013</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0014</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0015</td>
</tr>
</tbody>
</table>
### Sizes of C Objects (in Bytes)

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Alpha</th>
<th>Intel IA32</th>
<th>AMD 64</th>
</tr>
</thead>
<tbody>
<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>8</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>8</td>
</tr>
<tr>
<td>other pointer</td>
<td>8</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>
How should bytes within multi-byte word be ordered in memory?

Conventions

- Sun, PowerPC MacIntosh computers are “big endian” machines: least significant byte has highest address.
- Alpha, Intel MacIntosh, PC's are “little endian” machines: least significant byte has lowest address.
- ARM processor offer support for big endian, but mainly they are used in their default, little endian configuration.
- There are many (hundreds) of microcontrollers so check before you start programming!
**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>Address:</th>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value:</td>
<td>01</td>
<td>23</td>
<td>45</td>
<td>67</td>
</tr>
</tbody>
</table>

**Big Endian:**

<table>
<thead>
<tr>
<th>Address:</th>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value:</td>
<td>67</td>
<td>45</td>
<td>23</td>
<td>01</td>
</tr>
</tbody>
</table>

**Little Endian:**
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</th>
<th>Code</th>
<th>Assembly Rendition</th>
</tr>
</thead>
<tbody>
<tr>
<td>8048365:</td>
<td>5b</td>
<td></td>
<td>pop %ebx</td>
</tr>
<tr>
<td>8048366:</td>
<td>81 c3 ab 12 00 00</td>
<td></td>
<td>add $0x12ab,%ebx</td>
</tr>
<tr>
<td>804836c:</td>
<td>83 bb 28 00 00 00 00</td>
<td></td>
<td>cmpl $0x0,0x28(%ebx)</td>
</tr>
</tbody>
</table>

Deciphering Numbers: Consider the value 0x12ab in the second line of code:

- Pad to 4 bytes: 0x000012ab
- Split into bytes: 00 00 12 ab
- Reverse: ab 12 00 00
Code to Print Byte Representations of Data

Casting a pointer to unsigned char * creates a byte array.

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):**

```
0x11ffffffcb8 0x6d
0x11ffffffcb9 0x3b
0x11ffffffcba 0x00
0x11ffffffcbb 0x00
```
Representing Integers

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

\[ 15213_{10} = 0011101101101101_2 = 3B6D_{16} \]

<table>
<thead>
<tr>
<th></th>
<th>Linux</th>
<th>Alpha</th>
<th>Sun</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>6D 3B 00 00</td>
<td>6D 3B 00 00</td>
<td>00 00 3B 6D</td>
</tr>
<tr>
<td>B</td>
<td>93 C4 FF FF</td>
<td>93 C4 FF FF</td>
<td>FF FF C4 93</td>
</tr>
<tr>
<td>C</td>
<td>6D 3B 00 00</td>
<td>6D 3B 00 00 00 00 00 00 00 00 00</td>
<td>00 00 3B 6D</td>
</tr>
</tbody>
</table>

We’ll cover the representation of negatives shortly.
Representing Pointers

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

**Linux Address:**
Hex: BFFFF8D4
Binary: 1011111111111111111010100
In memory: D4 F8 FF BF

**Sun Address:**
Hex: EFFFFFFB2C
Binary: 1110111111111111111010110
In Memory: EF FF FB 2C

**Alpha Address:**
Hex: 1FFFFFFCA0
Binary: 00011111111111111111110010100000
In Memory: A0 FC FF FF 01 00 00 00

*Different compilers and machines assign different locations.*
All modern machines implement the IEEE Floating Point standard. This means that it is consistent across all machines.

```
float F = 15213.0;
```

Hex: 466DB400
Binary: 010001100110110110110100000000000000
In Memory (Linux/Alpha): 00 B4 6D 46
In Memory (Sun): 46 6D B4 00

Note that it’s not the same as the int representation, but you can see that the int is in there, if you know where to look.
Representing Strings

Strings in C

- Strings are represented by an array of characters.
- Each character is encoded in ASCII format.
  - Standard 7-bit encoding of character set.
  - Other encodings exist, but are less common.
  - Character 0 has code 0x30. Digit i has code 0x30+i.
- Strings should be null-terminated. That is, the final character has ASCII code 0.

Compatibility

- Byte ordering not an issue since the data are single byte quantities.
- Text files are generally platform independent, except for different conventions of line termination character(s).
Machine Level Code Representation

Encode Program as Sequence of Instructions

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

Remember: Programs are byte sequences too!
int sum(int x, int y) {
    return x + y;
}

For this example, Alpha and Sun use two 4-byte instructions. They use differing numbers of instructions in other cases.

PC uses 7 instructions with lengths 1, 2, and 3 bytes. Windows and Linux are not fully compatible.

Different machines typically use different instructions and encodings.

Instruction sequence for sum program:

Alpha: 00 00 30 42 01 80 FA 68
Sun: 81 C3 E0 08 90 02 00 09
PC: 55 89 E5 8B 45 OC 03 45 08 89 EC 5D C3
Developed by George Boole in the 19th century, Boolean algebra is the algebraic representation of logic. We encode “True” as 1 and “False” as 0.

**And:** \( A \land B = 1 \) when both \( A = 1 \) and \( B = 1 \).

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Or:** \( A \lor B = 1 \) when either \( A = 1 \) or \( B = 1 \).

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

**Not:** \( \neg A = 1 \) when \( A = 0 \).

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

**Xor:** \( A \oplus B = 1 \) 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></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
In a 1937 MIT Master’s Thesis, Claude Shannon showed that Boolean algebra would be a great way to model digital networks.

At that time, the networks were relay switches. But today, all combinational circuits can be described in terms of Boolean “gates.”
Mathematical Rings

- A *ring* is an algebraic structure.
- It includes a finite set of elements and some operators with certain properties.
- A ring has a finite number of elements, a *sum* operation, a *product* operation, additive inverses, and identity elements.
- The addition and product ops must be associative and commutative.
- The product operation should distribute over addition.

Integer Arithmetic

- $\langle \mathbb{Z}, +, \ast, , 0, 1 \rangle$ forms a ring.
- Addition is the sum operation.
- Multiplication is the product operation.
- Minus returns the additive inverse
- 0 is the identity for sum.
- 1 is identity for product.
\[ \langle \{0, 1\}, |, \&, \sim, 0, 1 \rangle \] forms a Boolean algebra.

- Or is the sum operation.
- And is the product operation.
- \( \sim \) is the “complement” operation (not additive inverse).
- 0 is the identity for sum.
- 1 is the identity for product.

Note that a Boolean algebra is not the same as a ring, though every Boolean algebra gives rise to a ring if you let \( \sim \) be the product operator.
Booleans Algebra like Integer Ring

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

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

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:
\[
\begin{align*}
\neg (\neg A) &= A \\
-(-A) &= A
\end{align*}
\]
Boolean Algebra vs. Integer Ring

**Boolean:** Sum distributes over product
\[ A \| (B \& C) = (A \| B) \& (A \| C) \quad A + (B \ast C) \neq (A + B) \ast (A + C) \]

**Boolean:** Idempotency
\[ A \| A = A \quad A + A \neq A \\
A \& A = A \quad A \ast A \neq A \]

**Boolean:** Absorption
\[ A \| (A \& B) = A \quad A + (A \ast B) \neq A \\
A \& (A \| B) = A \quad A \ast (A + B) \neq A \]

**Boolean:** Laws of Complements
\[ A \| \sim A = 1 \quad A + A \neq 1 \]

**Ring:** Every element has additive inverse
\[ A \| A \neq 0 \quad A + A = 0 \]
Properties of \& and ^

- \(<\{0, 1\}, ^, 0, 1\> \text{ forms a } Boolean \text{ ring.}
- \text{This is isomorphic to the integers mod 2.}
- \text{I} \text{ is the identity operation: } I(A) = A.

Commutative sum: \( A \& B = B \& A \)

Commutative product: \( A \& B = B \& A \)

Associative sum: \( (A \& B) \& C = A \& (B \& C) \)

Associative product: \( (A \& B) \& C = A \& (B \& C) \)

Prod. over sum: \( A \& (B \& C) = (A \& B) \& (A \& C) \)

0 is sum identity: \( A \& 0 = A \)

1 is prod. identity: \( A \& 1 = A \)

0 is product annihilator: \( A \& 0 = 0 \)

Additive inverse: \( A \& A = 0 \)
DeMorgan’s Laws
Express & in terms of |, and vice-versa:

\[ A \land B = \sim (\sim A \mid \sim B) \]
\[ A \lor B = \sim (\sim A \land \sim B) \]

Exclusive-Or using Inclusive Or:

\[ A \oplus B = (\sim A \land B) \mid (A \land \sim B) \]
\[ A \oplus B = (A \lor B) \land \sim (A \land B) \]
We can also operate on bit vectors (bitwise). All of the properties of Boolean algebra apply:

\[
\begin{array}{ccc}
01101001 & 01101001 & 01101001 \\
\& 01010101 & | 01010101 & ^ 01010101 & ~ 01010101 \\
\hline
01000001 & 01111101 & 00111100 & 10101010
\end{array}
\]
Representing Sets

Representation
A width \( w \) bit vector may represent subsets of \( \{0, \ldots, w-1\} \).
\( a_i = 1 \) iff \( j \in A \)

Bit vector A:

\[
\begin{array}{cccc}
0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\
7 & 6 & 5 & 4 & 3 & 2 & 1 & 0
\end{array}
\]
represents \( \{0, 3, 5, 6\} \)

Bit vector B:

\[
\begin{array}{cccc}
0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
7 & 6 & 5 & 4 & 3 & 2 & 1 & 0
\end{array}
\]
represents \( \{0, 2, 4, 6\} \)

What bit operations on these set representations correspond to: intersection, union, complement?
### Operations:
Given the two sets above, perform these bitwise ops to obtain:

<table>
<thead>
<tr>
<th>Set operation</th>
<th>Boolean op</th>
<th>Result</th>
<th>Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intersection</td>
<td>A &amp; B</td>
<td>01000001</td>
<td>{0, 6}</td>
</tr>
<tr>
<td>Union</td>
<td>A</td>
<td>B</td>
<td>01111101</td>
</tr>
<tr>
<td>Symmetric difference</td>
<td>A ^ B</td>
<td>00111100</td>
<td>{2, 3, 4, 5}</td>
</tr>
<tr>
<td>Complement</td>
<td>~A</td>
<td>10010110</td>
<td>{1, 2, 4, 7}</td>
</tr>
</tbody>
</table>
The operations &, |, ~, ^ are all available in C.

- Apply to any integral data type: long, int, short, char.
- View the arguments as bit vectors.
- Operations are applied bit-wise to the argument(s).

**Examples:** (char data type)

- \( \sim 0x41 \rightarrow 0xBE \)
- \( \sim 01000001_2 \rightarrow 10111110_2 \)
- \( \sim 0x00 \rightarrow 0xFF \)
- \( \sim 00000000_2 \rightarrow 11111111_2 \)
- \( 0x69 \& 0x55 \rightarrow 0x41 \)
- \( 01101001_2 \& 01010101_2 \rightarrow 01000011_2 \)
- \( 0x69|0x55 \rightarrow 0x7D \)
- \( 01101001_2|01010101_2 \rightarrow 01111101_2 \)
Contrast to Logical Operators in C

Remember the operators: &&, ||, !.

- View 0 as “False.”
- View anything nonzero as “True.”
- Always return 0 or 1.
- Allow for early termination.

**Examples:**

- `!0x41` → `0x00`
- `!0x00` → `0x01`
- `!!0x41` → `0x01`
- `!!0x69 && 0x55` → `0x01`
- `!!0x69 || 0x55` → `0x01`

Can use `p && *p` to avoid null pointer access. *How and why?*
**Left Shift:** $x << y$
Shift bit vector $x$ left by $y$ positions
- Throw away extra bits on the left.
- Fill with 0’s on the right.

**Right Shift:** $x >> y$
Shift bit vector $x$ right by $y$ positions.
- Throw away extra bits on the right.
- **Logical shift:** Fill with 0’s on the left.
- **Arithmetic shift:** Replicate with most significant bit on the left.

Arithmetic shift is useful with two’s complement integer representation.
### Shift Operations

<table>
<thead>
<tr>
<th>Argument ×</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 ×</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>
Bitwise XOR is a form of addition, with the extra property that each 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 */
}
```

<table>
<thead>
<tr>
<th></th>
<th>*x</th>
<th>*y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Begin</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>A ( \oplus ) B</td>
<td>B</td>
</tr>
<tr>
<td>2</td>
<td>A ( \oplus ) B</td>
<td>(A ( \oplus ) B) ( \oplus ) B = A</td>
</tr>
<tr>
<td>3</td>
<td>(A ( \oplus ) B) ( \oplus ) A = B</td>
<td>B</td>
</tr>
<tr>
<td>End</td>
<td>A</td>
<td>B</td>
</tr>
</tbody>
</table>

Is there ever a case where this code fails?
Main Points

It’s all about bits and bytes.
- Numbers
- Programs
- Text

Different machines follow different conventions.
- Word size
- Byte ordering
- Representations

Boolean algebra is the mathematical basis.
- Basic form encodes “False” as 0 and “True” as 1.
- General form is like bit-level operations in C; good for representing and manipulating sets.