Bits and Bytes
Aug. 31, 2000

Topics
• Why bits?
• Representing information as bits
  – Binary/Hexadecimal
  – Byte representations
    – numbers
    – characters and strings
    – Instructions
• Bit-level manipulations
  – Boolean algebra
  – Expressing in C

Binary Representations

Base 2 Number Representation
• Represent $15213_{10}$ as $111011011011_{2}$
• 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

- $0.0V$
- $0.5V$
- $2.8V$
- $3.3V$

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
  – $1.5213 \times 10^{4}$

Implementing Electronically
• Hard to store
  – ENIAC (First electronic computer) used 10 vacuum tubes / digit
• Hard to transmit
  – Need high precision to encode 10 signal levels on single wire
• Messy to implement digital logic functions
  – Addition, multiplication, etc.

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
- Hexadecimal: 0₁₆ to FF₁₆
  - Base 16 number representation
  - Use characters '0' to '9' and 'A' to 'F'
  - Write FA1D37B₁₆ in C as 0xFA1D37B
    » Or 0xfa1d37b

<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>

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)
  - Potentially address ≈ 1.8 x 10¹⁹ 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>Addr = 0004</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>Addr = 0012</td>
<td>0003</td>
<td></td>
</tr>
</tbody>
</table>

Data Representations

Sizes of C Objects (in Bytes)

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Compaq Alpha</th>
<th>Typical 32-bit</th>
<th>Intel IA32</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>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

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

Conventions
- Alphas, PC's are “Little Endian” machines
  - Least significant byte has lowest address
- Sun’s, Mac’s are “Big Endian” machines
  - Least significant byte has highest address

Example
- Variable \( x \) has 4-byte representation 0x01234567
- Address given by &x is 0x100

Big Endian

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

Little Endian

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

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	0x%.2x\n", start+i, start[i]);
  printf("\n");
}
```

Printf directives:
- %p: Print pointer
- %x: Print Hexadecimal

Representing Integers

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

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

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

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

Alpha Address
Hex: 01 FF FF FF FC A 0
Binary: 0001 1111 1111 1111 1111 1110 1010 0000

Sun Address
Hex: EF FF FB 2C
Binary: 1110 1111 1111 1111 1011 0010 1100

Representing Floats

```c
Float F = 15213.0;
```

IEEE Single Precision Floating Point Representation
Hex: 4 6 6 D B 4 0 0
Binary: 0100 0110 0110 1101 1011 0100 0000 0000

Not same as integer representation, but consistent across machines

Representing Strings

Strings in C
- Represented by array of characters
- Each character encoded in ASCII format
  - Standard 7-bit encoding of character set
  - Other encodings exist, but uncommon
  - Character “0” has code 0x30
    - Digit / has code 0x30 + i
- String should be null-terminated
  - Final character = 0

Compatibility
- Byte ordering not an issue
  - Data are single byte quantities
- Text files generally platform independent
  - Except for different conventions of line termination character!

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!
Representing Instructions

```c
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 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
- | 0 1
  - 0 0 0
  - 1 0 1

#### OR
- A|B = 1 when either A=1 or B=1
- | 0 1
  - 0 0 1
  - 1 1 1

#### NOT
- ~A = 1 when A=0
- ~ 0 1
  - 0 1
  - 1 0

#### EXCLUSIVE-OR (XOR)
- A^B = 1 when either A=1 or B=1, but not both
- ^ 0 1
  - 0 0 1
  - 1 1 0

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
- \( A \& \neg B \) or \( \neg A \& B \)
- \( A \& \neg B \) or \( \neg A \& B \) = \( A \& B \)

Properties of and | Operations

#### INTEGER ARITHMETIC
- \( \langle Z, +, *, -, 0, 1 \rangle \) 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
- \( \langle \{0,1\}, \&, |, \neg, 0, 1 \rangle \) forms a “Boolean algebra”
- Or is “sum” operation
- And is “product” operation
  - is “complement” operation (not additive inverse)
  - 0 is identity for sum
  - 1 is identity for product
Properties of Rings & Boolean Algebras

Boolean Algebra

- **Commutativity**
  - $A \lor B = B \lor A$
  - $A \land B = B \land A$

- **Associativity**
  - $(A \lor B) \lor C = A \lor (B \lor C)$
  - $(A \land B) \land C = A \land (B \land C)$

- **Product distributes over sum**
  - $A \land (B \lor C) = (A \land B) \lor (A \land C)$

- **Sum and product identities**
  - $A \lor 0 = A$
  - $A \land 1 = A$

- **Zero is product annihilator**
  - $A \land 0 = 0$

- **Cancellation of negation**
  - $\neg (\neg A) = A$

Integer Ring

- $A + B = B + A$
- $A \cdot B = B \cdot A$

Ring $\neq$ Boolean Algebra

- **Boolean: Sum distributes over product**
  - $A \lor (B \land C) = (A \lor B) \land (A \lor C)$
  - $A \land (B + C) \neq (A + B) \cdot (B + C)$

- **Boolean: Idempotency**
  - $A \lor A = A$
  - $A \land A = A$

- **Boolean: Absorption**
  - $A \lor (A \land B) = A$
  - $A \land (A \lor B) = A$

- **Boolean: Laws of Complements**
  - $A \lor \neg A = 1$
  - $A \land \neg A = 0$

- **Ring: Every element has additive inverse**
  - $A \lor \neg A = 0$
  - $A \land \neg A = 0$

Properties of $\&$ and $\wedge$

Boolean Ring

- $\langle \{0,1\}, \land, \lor, \ 0, \ 1 \rangle$
- Identical to integers mod 2
- $I$ is identity operation: $I(A) = A$
- $A \wedge A = 0$

Property

- **Commutative sum**
  - $A \land B = B \land A$

- **Commutative product**
  - $A \cdot B = B \cdot A$

- **Associative sum**
  - $(A \land B) \land C = A \land (B \land C)$

- **Associative product**
  - $(A \cdot B) \cdot C = A \cdot (B \cdot C)$

- **Prod. over sum**
  - $A \land (B \lor C) = (A \land B) \lor (A \land C)$

- **0 is sum identity**
  - $A \lor 0 = A$

- **1 is prod. identity**
  - $A \cdot 1 = A$

- **0 is product annihilator**
  - $A \land 0 = 0$

- **Additive inverse**
  - $A \lor \neg A = 0$

Relations Between Operations

DeMorgan’s Laws

- Express $\&$ in terms of $\lor$, and vice-versa
  - $A \land B = \neg(\neg A \lor \neg B)$
  - $A \lor B = \neg(\neg A \land \neg B)$

- A and B are true if and only if neither A nor B is false
  - $A \land B = \neg(\neg A \lor \neg B)$

- Either A is true, or B is true, but not both

Exclusive-Or using Inclusive Or

- $A \land B = (\neg A \lor B) \land (A \land \neg B)$

- Exactly one of A and B is true
  - $A \lor B = (A \land B) \lor (\neg A \land \neg B)$

- Either A is true, or B is true, but not both
### General Boolean Algebras

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

<table>
<thead>
<tr>
<th>Bit Vector</th>
<th>Subset</th>
</tr>
</thead>
<tbody>
<tr>
<td>01110101</td>
<td>{0, 3, 5, 6}</td>
</tr>
<tr>
<td>01010101</td>
<td>{0, 2, 4, 6}</td>
</tr>
</tbody>
</table>

**Representation of Sets**
- Width \( w \) bit vector represents subsets of \( \{0, \ldots, w - 1\} \)
- \( a_j = 1 \) if \( j \in A \)

<table>
<thead>
<tr>
<th>Bit Vector</th>
<th>Subset</th>
</tr>
</thead>
<tbody>
<tr>
<td>001000001</td>
<td>{0, 6}</td>
</tr>
<tr>
<td>01111100</td>
<td>{2, 3, 4, 5}</td>
</tr>
<tr>
<td>10101010</td>
<td>{1, 3, 5, 7}</td>
</tr>
</tbody>
</table>

### Bit-Level Operations in C

**Operations &, |, ~, ^ Available in C**
- Apply to any “integral” data type
  - long, int, short, char
- View arguments as bit vectors
- Arguments applied bit-wise

**Examples (Char data type)**
- \(~0x41\) --> 0xBE
- \(~01000001\) --> 10111110
- \(~0x00\) --> 0xFF
- \(~00000000\) --> 11111111

- \(0x69 \& 0x55\) --> 0x41
- \(01101001 \& 01010101\) --> 01000001
- \(0x69 | 0x55\) --> 0x7D
- \(01101001 | 01010101\) --> 01111111

### 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

<table>
<thead>
<tr>
<th>Argument ( x )</th>
<th>( 01100010 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( &lt;&lt; 3 )</td>
<td>00010000</td>
</tr>
</tbody>
</table>

**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>( 10100010 )</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>01110000</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 */  
}
```

<table>
<thead>
<tr>
<th>Step</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^B</td>
<td>B</td>
</tr>
<tr>
<td>2</td>
<td>A^B</td>
<td>(A^B)^B = A^[(B^B) = A^0 = A]</td>
</tr>
<tr>
<td>3</td>
<td>(A^B)^A = (B^A)^A = B^(A^A) = B^0 = B</td>
<td>A</td>
</tr>
<tr>
<td>End</td>
<td>B</td>
<td>A</td>
</tr>
</tbody>
</table>