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

Bits and Bytes
Aug. 29, 2002

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
  - $1.5213 \times 10^4$

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.
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 $1.1101101101101101_2 \times 2^{13}$

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

0.0V 0.5V 2.8V 3.3V
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: 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>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 \( \approx 1.8 \times 10^{19} \) 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 = 0008</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>Addr = 0004</td>
<td>Addr =</td>
<td>0001</td>
<td>0001</td>
</tr>
<tr>
<td>Addr = 0008</td>
<td>Addr =</td>
<td>0002</td>
<td>0002</td>
</tr>
<tr>
<td>Addr = 0012</td>
<td>Addr =</td>
<td>0003</td>
<td>0003</td>
</tr>
</tbody>
</table>

- 64-bit Words

| Addr = 0000 |
| 0004         |
| 0008         |
| 0012         |
| 0016         |
| 0020         |
| 0024         |
| 0028         |
| 0032         |
| 0036         |
| 0040         |
| 0044         |
| 0048         |
| 0052         |
| 0056         |
| 0060         |
| 0064         |
| 0068         |
| 0072         |
| 0076         |
| 0080         |
| 0084         |
| 0088         |
| 0092         |
| 0096         |
| 00A0         |
| 00A4         |
| 00A8         |
| 00AC         |
| 00B0         |
| 00B4         |
| 00B8         |
| 00BC         |
| 00C0         |
| 00C4         |
| 00C8         |
| 00CC         |
| 00D0         |
| 00D4         |
| 00D8         |
| 00DC         |
| 00E0         |
| 00E4         |
| 00E8         |
| 00EC         |
| 00F0         |
| 00F4         |
| 00F8         |
| 00FC         |

15-213, F'02
## 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

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

<table>
<thead>
<tr>
<th>Little Endian</th>
<th>0x100 0x101 0x102 0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>67 45 23 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)
{
    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
show_bytes Execution Example

```c
int a = 15213;
printf("int a = 15213;\n");
show_bytes((pointer) &a, sizeof(int));
```

Result (Linux):

```plaintext
int a = 15213;
0x11ffffffcb8 0x6d
0x11ffffffcb9 0x3b
0x11ffffffcba 0x00
0x11ffffffcbb 0x00
```
Representing Integers

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

<table>
<thead>
<tr>
<th>Decimal: 15213</th>
<th>Binary: 0011 1011 0110 1101</th>
<th>Hex: 3 B 6 D</th>
</tr>
</thead>
</table>

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

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

### Alpha Address

- **Hex:** 1 F F F F F F C A 0
- **Binary:** 0001 1111 1111 1111 1111 1111 1100 1010 0000

### Sun Address

- **Hex:** E F F F F F B 2 C
- **Binary:** 1110 1111 1111 1111 1111 1111 1011 0010 1100

### Linux Address

- **Hex:** B F F F F 8 D 4
- **Binary:** 1011 1111 1111 1111 1111 1000 1101 0100

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

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
15213: 1110 1101 1011 01

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
  - Other encodings exist, but uncommon
  - 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
  - Data are single byte quantities
- Text files generally platform independent
  - Except for different conventions of line termination character(s)!

```c
char S[6] = "15213";
```
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

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>&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

- $A | B = 1$ 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>1</td>
</tr>
</tbody>
</table>

Exclusive-Or (Xor)

- $A \oplus B = 1$ when either $A=1$ or $B=1$, but not both

<table>
<thead>
<tr>
<th>$\oplus$</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

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

= \( A \wedge B \)
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

- \{0,1\}, |, &, \sim, 0, 1\} 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**

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

- **Sum and product identities**

  \[
  \begin{align*}
  A \lor 0 & = A \\
  A & = 1 \\
  A \cdot 1 & = A
  \end{align*}
  \]

- **Zero is product annihilator**

  \[
  \begin{align*}
  A \land 0 & = 0 \\
  A \cdot 0 & = 0
  \end{align*}
  \]

- **Cancellation of negation**

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

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

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

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

- **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

\[ \langle \{0,1\}, \land, \lor, I, 0, 1 \rangle \]

- Identical to integers mod 2
- \( I \) is identity operation: \( I(A) = A \)
  \[ A \land A = 0 \]

**Property**

**Boolean Ring**

- Commutative sum
  \[ A \lor B = B \lor A \]
- Commutative product
  \[ A \land B = B \land A \]
- Associative sum
  \[ (A \lor B) \lor C = A \lor (B \lor C) \]
- Associative product
  \[ (A \land B) \land C = A \land (B \land C) \]
- Prod. over sum
  \[ A \land (B \lor C) = (A \land B) \lor (B \land C) \]
- 0 is sum identity
  \[ A \lor 0 = A \]
- 1 is prod. identity
  \[ A \land 1 = A \]
- 0 is product annihilator
  \[ A \land 0 = 0 \]
- Additive inverse
  \[ A \lor A = 0 \]
Relations Between Operations

DeMorgan’s Laws

Express & in terms of |, and vice-versa

| A & B  =  ~(~A | ~B) |

» A and B are true if and only if neither A nor B is false

| A | B  =  ~(~A & ~B) |

» A or B are true if and only if A and B are not both false

Exclusive-Or using Inclusive Or

| A ^ B  =  (~A & B) | (A & ~B) |

» Exactly one of A and B is true

| A ^ B  =  (A | B) & ~(A & B) |

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

Operate on Bit Vectors

- Operations applied bitwise

\[
\begin{align*}
01101001 & \quad 01101001 & \quad 01101001 \\
\& 01010101 & \quad | 01010101 & \quad ^ 01010101 & \quad \sim 01010101 \\
01000001 & \quad 01111101 & \quad 00111100 & \quad 10101010
\end{align*}
\]

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

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
- View arguments as bit vectors
- Arguments applied bit-wise

Examples (Char data type)

- ~0x41 --> 0xBE
  ~01000001₂ --> 10111110₂
- ~0x00 --> 0xFF
  ~00000000₂ --> 11111111₂
- 0x69 & 0x55 --> 0x41
  01101001₂ & 01010101₂ --> 01000001₂
- 0x69 | 0x55 --> 0x7D
  01101001₂ | 01010101₂ --> 01111101₂
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: \( x \ll y \)
- Shift bit-vector \( x \) left \( y \) positions
  - Throw away extra bits on left
  - Fill with 0’s on right

Right Shift: \( x \gg y \)
- 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>( \ll 3 )</td>
<td>00010000</td>
</tr>
<tr>
<td>Log. ( \gg 2 )</td>
<td>00011000</td>
</tr>
<tr>
<td>Arith. ( \gg 2 )</td>
<td>00011000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument ( x )</th>
<th>10100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \ll 3 )</td>
<td>00010000</td>
</tr>
<tr>
<td>Log. ( \gg 2 )</td>
<td>00101000</td>
</tr>
<tr>
<td>Arith. ( \gg 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
  \( A \land A = 0 \)

```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^B</td>
<td>B</td>
</tr>
<tr>
<td>2</td>
<td>A^B</td>
<td>(A^B)^B = A</td>
</tr>
<tr>
<td>3</td>
<td>(A^B)^A = B</td>
<td>A</td>
</tr>
<tr>
<td>End</td>
<td>B</td>
<td>A</td>
</tr>
</tbody>
</table>
Main Points

It’s All About Bits & Bytes
- Numbers
- Programs
- Text

Different Machines Follow Different Conventions
- Word size
- Byte ordering
- Representations

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