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

Bits and Bytes
Aug. 26, 1999

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 $111011011011012$
- Represent $1.20_{10}$ as $1.0011001100110011[0011]...2$
- 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

0.0V — 0.5V — 2.8V — 3.3V

- Straightforward implementation of arithmetic functions
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)
# 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>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long int</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>char *</td>
<td>8</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 $0x1234567$
• 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></td>
<td>01</td>
<td>23</td>
<td>45</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>
Examining Data Representations

Code to Print Byte Representation of Data

- Cast 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:

```
int a = 15213;
0x11ffffffffcb8  0x6d
0x11ffffffffcb9  0x3b
0x11ffffffffcba  0x00
0x11ffffffffcbb  0x00
```
Representing Integers

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

Decimal: 15213
Binary: 0011 1011 0110 1101
Hex: 3B6D

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

\begin{align*}
\text{int } B &= -15213; \\
\text{int } \ast P &= \&B;
\end{align*}

**Alpha Address**

<table>
<thead>
<tr>
<th>Hex</th>
<th>1 F F F F F F C A 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>0001 1111 1111 1111 1111 1111 1100 1010 0000</td>
</tr>
</tbody>
</table>

**Sun Address**

<table>
<thead>
<tr>
<th>Hex</th>
<th>E F F F F B 2 C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>1110 1111 1111 1111 1111 1011 0010 1100</td>
</tr>
</tbody>
</table>

Different compilers & machines assign different locations to objects
Representing Floats

Float $F = 15213.0$;

IEEE Single Precision Floating Point Representation

<table>
<thead>
<tr>
<th>Hex:</th>
<th>4 6 6 D B 4 0 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary:</td>
<td>0100 0110 0110 1101 1011 0100 0000 0000</td>
</tr>
<tr>
<td>15213:</td>
<td>1110 1101 1011 01</td>
</tr>
</tbody>
</table>

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 \(i\) 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!

char S[6] = "15213";

Alpha S | Sun S
---|---
31 | 31
35 | 35
32 | 32
31 | 31
33 | 33
00 | 00
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!
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 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

- \( A \mid B = 1 \) 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>1</td>
</tr>
</tbody>
</table>

Exclusive-Or (Xor)

- \( A \wedge B = 1 \) 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>

= class02.ppt
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

\[ A \& \neg B \mid \neg A \& B = A \oplus B \]
Properties of & and | Operations

Integer Arithmetic

- $\langle \mathbb{Z}, +, \ast, -, 0, 1 \rangle$ 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
### Properties of Rings & Boolean Algebras

<table>
<thead>
<tr>
<th>Boolean Algebra</th>
<th>Integer Ring</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Commutativity</strong></td>
<td></td>
</tr>
<tr>
<td>(A \lor B = B \lor A)</td>
<td>(A + B = B + A)</td>
</tr>
<tr>
<td>(A \land B = B \land A)</td>
<td>(A \times B = B \times A)</td>
</tr>
<tr>
<td><strong>Associativity</strong></td>
<td></td>
</tr>
<tr>
<td>((A \lor B) \lor C = A \lor (B \lor C))</td>
<td>((A + B) + C = A + (B + C))</td>
</tr>
<tr>
<td>((A \land B) \land C = A \land (B \land C))</td>
<td>((A \times B) \times C = A \times (B \times C))</td>
</tr>
<tr>
<td><strong>Product distributes over sum</strong></td>
<td></td>
</tr>
<tr>
<td>(A \land (B \lor C) = (A \land B) \lor (A \land C))</td>
<td>(A \times (B + C) = A \times B + B \times C)</td>
</tr>
<tr>
<td><strong>Sum and product identities</strong></td>
<td></td>
</tr>
<tr>
<td>(A \lor 0 = A)</td>
<td>(A + 0 = A)</td>
</tr>
<tr>
<td>(A \land 1 = A)</td>
<td>(A \times 1 = A)</td>
</tr>
<tr>
<td><strong>Zero is product annihilator</strong></td>
<td></td>
</tr>
<tr>
<td>(A \land 0 = 0)</td>
<td>(A \times 0 = 0)</td>
</tr>
<tr>
<td><strong>Cancellation of negation</strong></td>
<td></td>
</tr>
<tr>
<td>(\neg (\neg A) = A)</td>
<td>(\neg (\neg A) = A)</td>
</tr>
</tbody>
</table>
Ring $\neq$ Boolean Algebra

**Boolean Algebra**

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

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

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

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

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

Boolean Ring

• \( \langle \{0,1\}, ^\^, &, I, 0, 1 \rangle \)

• Identical to integers mod 2

• I is identity operation: \( I (A) = A \)

\[ A ^\^ A = 0 \]

Property | Boolean Ring
--- | ---
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 | \( A \& (B ^\^ C) = (A \& B) ^\^ (B \& C) \)
0 is sum identity | \( A ^\^ 0 = A \)
1 is prod. identity | \( A \& 1 = A \)
0 is product annihilator | \( A \& 0 = 0 \)
Additive inverse | \( A ^\^ A = 0 \)
Relations Between Operations

DeMorgan’s Laws

- Express \& in terms of |, and vice-versa

\[
A \land B = \neg(\neg A \lor \neg B)
\]

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

\[
A \lor B = \neg(\neg A \land \neg B)
\]

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

**Exclusive-Or using Inclusive Or**

\[
A \oplus B = (\neg A \land B) \lor (A \land \neg B)
\]

» Exactly one of A and B is true

\[
A \oplus B = (A \lor B) \land \neg(A \land B)
\]

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

Operate on Bit Vectors

- Operations applied bitwise

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

Representation of Sets

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

\[
\begin{array}{ccc}
-01101001 & \{0, 3, 5, 6\} \\
-01010101 & \{0, 2, 4, 6\}
\end{array}
\]

- & Intersection \quad 01000001 \quad \{0, 6\}
- | Union \quad 01111101 \quad \{0, 2, 3, 4, 5, 6\}
- ^ Symmetric difference \quad 00111100 \quad \{2, 3, 4, 5\}
- ~ Complement \quad 10101010 \quad \{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)

- $\sim 0x41 \rightarrow 0xBE$
  $\sim 01000001_2 \rightarrow 10111110_2$

- $\sim 0x00 \rightarrow 0xFF$
  $\sim 00000000_2 \rightarrow 11111111_2$

- $0x69 \& 0x55 \rightarrow 0x41$
  $01101001_2 \& 01010101_2 \rightarrow 01000001_2$

- $0x69 \mid 0x55 \rightarrow 0x7D$
  $01101001_2 \mid 01010101_2 \rightarrow 01111101_2$
Contrast: Logic Operations in C

Contrast to Logical Operators

- &&, ||, !
  - View 0 as “False”
  - Anything nonzero as “True”
  - Always return 0 or 1

Examples (char data type)

- !0x41 --> 0x00
- !0x00 --> 0x01
- !!0x41 --> 0x01

- 0x69 && 0x55 --> 0x01
- 0x69 || 0x55 --> 0x01
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
# 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>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>