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

**Bits and Bytes**
January 13, 2005

**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

![Diagram showing binary representation and electronic implementation](image)
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 0xFA1D37B₁₆
    - 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>
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 use 32 bits (4 bytes) words
  - Limits addresses to 4GB
  - Becoming too small for memory-intensive applications
- High-end systems use 64 bits (8 bytes) words
  - Potential address space ≈ 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></td>
<td>0001</td>
<td></td>
</tr>
<tr>
<td>Addr = 0008</td>
<td></td>
<td>0002</td>
<td></td>
</tr>
<tr>
<td>Addr = 0012</td>
<td></td>
<td>0003</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Addr = 0008</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>

- 15-213, S’05
## 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>8/16†</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

(†: Depends on compiler&OS, 128bit FP is done in software)
Byte Ordering

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

Conventions

- Suns, Macs (PPC) 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 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 (x86 Linux):

```plaintext
int a = 15213;
0x11fffffc8  0x6d
0x11fffffc9  0x3b
0x11fffffcba  0x00
0x11fffffcbb  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

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

<table>
<thead>
<tr>
<th>Alpha Address</th>
<th>Sun Address</th>
<th>x86 Linux Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hex: 1 F F F F F F C A 0</td>
<td>Hex: E F F F F F B 2 C</td>
<td>Hex: B F F F F F 8 D 4</td>
</tr>
<tr>
<td>Binary: 0001 1111 1111 1111 1111 1111 1100 1010 0000</td>
<td>Binary: 1110 1111 1111 1111 1111 1011 0010 1100</td>
<td>Binary: 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;

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
  - 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 and modern MacOS (‘\n’ = 0x0a = ^J)
    » Older MacOS (‘\r’ = 0x0d = ^M)
    » DOS and HTTP (‘\r\n’ = 0x0d0a = ^M^J)
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
  - Alphas, Suns, PPC 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 is 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
- x86 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

- \( A\lor B = 1 \) when either \( A=1 \) or \( B=1 \)

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

Exclusive-Or (Xor)

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

<table>
<thead>
<tr>
<th>( \land )</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>
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

Boolean Algebra

- \(\{0,1\}, \lor, \land, \neg, 0, 1\) forms a “Boolean algebra”
- Or is “sum” operation
- And is “product” operation
- \(\neg\) is “complement” operation (not additive inverse)
- 0 is identity for sum
- 1 is identity for product
Boolean Algebra \approx Integer Ring

- **Commutativity**
  \[ A \lor B = B \lor A \]
  \[ A \land B = B \land A \]
  \[ A + B = B + A \]
  \[ A \ast B = B \ast A \]

- **Associativity**
  \[ (A \lor B) \lor C = A \lor (B \lor C) \]
  \[ (A \land B) \land C = A \land (B \land C) \]
  \[ (A + B) + C = A + (B + C) \]
  \[ (A \ast B) \ast C = A \ast (B \ast C) \]

- **Product distributes over sum**
  \[ A \land (B \lor C) = (A \land B) \lor (A \land C) \]
  \[ A \ast (B + C) = A \ast B + B \ast C \]

- **Sum and product identities**
  \[ A \lor 0 = A \]
  \[ A \land 1 = A \]
  \[ A + 0 = A \]
  \[ A \ast 1 = A \]

- **Zero is product annihilator**
  \[ A \land 0 = 0 \]
  \[ A \ast 0 = 0 \]

- **Cancellation of negation**
  \[ \sim (\sim A) = A \]
  \[ -(-A) = A \]
Boolean Algebra ≠ Integer Ring

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

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

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

- 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*
  \[ A \lor \neg A \neq 0 \quad A + \neg A = 0 \]
Properties of & and ^

Boolean Ring

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

Property

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

DeMorgan’s Laws

- Express & in terms of l, and vice-versa
  - A & B = ~(~A l ~B)
    » A and B are true if and only if neither A nor B is false
  - A l 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) l (A & ~B)
  » Exactly one of A and B is true
- A ^ B = (A l B) & ~(A & B)
  » Either A is true, or B is true, but not both
General Boolean Algebras

Operate on Bit Vectors

- Operations applied bitwise

```
01101001 & 01010101 = 01000001
01101001 | 01010101 = 01111101
01101001 ^ 01010101 = 00111100
~ 01010101 = 10101010
```

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 \( j \in A \)

01101001 \quad \{0, 3, 5, 6\}

76543210

01010101 \quad \{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, unsigned
- 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:  \( x << y \)
- 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
  \[ A ^ A = 0 \]

```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>
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) & 033333333333)
         - ((n >> 2) & 011111111111);
    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