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

Bits and Bytes
September 2, 2004

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 15213₁₀ as 11101101101₁₀
- Represent 1.2₀₁₀ as 1.0011001100110011[0011]...₂
- Represent 1.5213 X 10⁴ as 1.1101101101₁₀ X 2¹³

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

---

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 $255_{10}$
  - First digit must not be 0 in C
- Octal: $000_8$ to $0377_8$
  - Use leading 0 in C
- 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>0000000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>00000001</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>00000010</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>00000011</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>00000100</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>00000101</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>00000110</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>00000111</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>00001000</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>00001001</td>
</tr>
<tr>
<td>A</td>
<td>10</td>
<td>00010100</td>
</tr>
<tr>
<td>B</td>
<td>11</td>
<td>00010111</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>00011000</td>
</tr>
<tr>
<td>D</td>
<td>13</td>
<td>00011001</td>
</tr>
<tr>
<td>E</td>
<td>14</td>
<td>00011110</td>
</tr>
<tr>
<td>F</td>
<td>15</td>
<td>00100001</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 = $1.8 \times 10^{19}$ 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)

- C Data Type | Alpha (RIP) | Typical 32-bit | Intel IA32
- unsigned    | 4          | 4             | 4
- int         | 4          | 4             | 4
- long int    | 8          | 4             | 4
- char        | 1          | 1             | 1
- short       | 2          | 2             | 2
- float       | 4          | 4             | 4
- double      | 8          | 8             | 8
- long double | 8/16†      | 8             | 10/12
- char *      | 8          | 4             | 4

† 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

- 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>cmp 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 (Linux):

```c
int a = 15213;
0x1fffffff 0x6d
0x1fffffff 0x3b
0x1fffffff 0x00
0x1fffffff 0x00
```
Representing Integers

\[
\text{int } A = 15213; \\
\text{int } B = -15213; \\
\text{long int } C = 15213;
\]

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

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

Two's complement representation (Covered next lecture)

Representing Pointers

\[
\text{int } B = -15213; \\
\text{int } *P = &B;
\]

<table>
<thead>
<tr>
<th></th>
<th>Alpha P</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A0</td>
<td>FC</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>01</td>
</tr>
<tr>
<td></td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

\[
\text{Alpha Address} \\
\text{Hex: } 1 F F F F F F C A 0 \\
\text{Binary: } 0001 1111 1111 1111 1111 1111 1111 1110 1010
\]

<table>
<thead>
<tr>
<th></th>
<th>Sun P</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>EF</td>
<td>FF</td>
<td>FF</td>
<td>FB</td>
<td>2C</td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{Sun Address} \\
\text{Hex: } E F F F F B 2 C \\
\text{Binary: } 1110 1111 1111 1111 1111 1011 0010
\]

<table>
<thead>
<tr>
<th></th>
<th>Linux P</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>D4</td>
<td>F8</td>
<td>FF</td>
<td>BF</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{Linux Address} \\
\text{Hex: } B F F F F F 8 D 4 \\
\text{Binary: } 1011 1111 1111 1111 1111 1000 1101 0100
\]

Different compilers & machines assign different locations to objects
Representing Floats

Float F = 15213.0;

<table>
<thead>
<tr>
<th>Linux/Alpha F</th>
<th>Sun F</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>46</td>
</tr>
<tr>
<td>B4</td>
<td>6D</td>
</tr>
<tr>
<td>6D</td>
<td>B4</td>
</tr>
<tr>
<td>46</td>
<td>00</td>
</tr>
</tbody>
</table>

IEEE Single Precision Floating Point Representation

Hex: 4 6 6 D B 4 0 0
Binary: 0100 0110 0110 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

char S[6] = "15213";

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 (\n = 0x0a = \n)  
    » Mac (\r = 0x0d = \r)  
    » DOS and HTTP (\r\n = 0x0d0a = \r\n)
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 is not binary compatible

Programs are Byte Sequences Too!

---

Representing Instructions

```c
int sum(int x, int y) {
    return x+y;
}
```

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha sum</td>
<td>Sun sum</td>
<td>PC sum</td>
</tr>
<tr>
<td>00</td>
<td>81</td>
<td>55</td>
</tr>
<tr>
<td>00</td>
<td>C3</td>
<td>89</td>
</tr>
<tr>
<td>30</td>
<td>E0</td>
<td>E5</td>
</tr>
<tr>
<td>42</td>
<td>08</td>
<td>8B</td>
</tr>
<tr>
<td>01</td>
<td>90</td>
<td>45</td>
</tr>
<tr>
<td>80</td>
<td>02</td>
<td>0C</td>
</tr>
<tr>
<td>FA</td>
<td>00</td>
<td>03</td>
</tr>
<tr>
<td>6B</td>
<td>09</td>
<td>45</td>
</tr>
</tbody>
</table>

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**

<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**

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

**Not**

<table>
<thead>
<tr>
<th>~A</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)**

<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>0</td>
<td>1</td>
<td>0</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

```
A & B
```

Connection when

```
A & ~B | ~A & B
```

= A^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**

**Boolean Algebra**
- \(\langle \{0,1\}, \|, \&, \neg, 0, 1 \rangle\) 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
- \( A \oplus B = B \oplus A \)
- \( A \times B = B \times A \)

### Associativity
- \( (A \oplus B) \oplus C = A \oplus (B \oplus C) \)
- \( (A \times B) \times C = A \times (B \times C) \)

### Product distributes over sum
- \( A \oplus (B \times C) = (A \oplus B) \times (A \oplus C) \)
- \( A \times (B + C) = A \times B + A \times C \)

### Sum and product identities
- \( A \oplus 0 = A \)
- \( A \times 1 = A \)
- \( A \times 0 = 0 \)

### Zero is product annihilator
- \( A \times 0 = 0 \)

### Cancellation of negation
- \( \neg (\neg A) = A \)
- \( - (\neg A) = A \)

---

## Boolean Algebra \( \neq \) Integer Ring

### Boolean: Sum distributes over product
- \( A \oplus (B \times C) = (A \oplus B) \times (A \times C) \)
- \( A \times (B + C) = (A + B) \times (A + C) \)

### Boolean: Idempotency
- \( A \oplus A = A \)
- \( A \times A = A \)

- \( \text{“A is true” or “A is true” = “A is true”} \)

- \( A \oplus A = A \)

### Boolean: Absorption
- \( A \oplus (A \times B) = A \)
- \( A \times (A \oplus B) = A \)

### Boolean: Laws of Complements
- \( \neg \neg A = A \)
- \( A \times \neg A = 0 \)
- \( A \oplus \neg A = 1 \)

### Ring: Every element has additive inverse
- \( A \times 0 = A \times \neg A = 0 \)
- \( A + \neg A = 1 \)
Boolean Ring

Properties of & and ^

- (0,1), ^, &, 1, 0, 1
- Identical to integers mod 2
- 1 is identity operation: 1 (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) ^ (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

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

<table>
<thead>
<tr>
<th>01101001</th>
<th>01101001</th>
<th>01101001</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp; 01010101</td>
<td></td>
<td>^ 01010101</td>
</tr>
<tr>
<td>01000001</td>
<td>01111101</td>
<td>00111100</td>
</tr>
</tbody>
</table>

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 \(j \in A\)
  
<table>
<thead>
<tr>
<th>01101001</th>
<th>{0, 3, 5, 6}</th>
</tr>
</thead>
<tbody>
<tr>
<td>01010101</td>
<td>{0, 2, 4, 6}</td>
</tr>
</tbody>
</table>

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
  01101001₁ & 01010101₂ --> 0100001₁
- 0x69 | 0x55 --> 0x7D
  01101001₁ | 01010101₂ --> 0111111₁

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

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>( x )</th>
<th>( y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Begin</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>A^B</td>
</tr>
<tr>
<td>2</td>
<td>A^B</td>
</tr>
<tr>
<td>3</td>
<td>(A^B)^A = B</td>
</tr>
<tr>
<td>End</td>
<td>B</td>
</tr>
</tbody>
</table>
More Fun with Bitvectors

Bit-board representation of chess position:

```
unsigned long long blk_king, wht_king, wht_rook_mv2, ...
```

```
8   0 1 2
7   
6   
5   
4   
3   
2   
1   a b c d e f g h
```

```
whl_king = 0x000000000001000ull;
blk_king = 0x000400000000000ull;
wht_rook_mv2 = 0x10ef1010101010ull;
...
/*
 * Is black king under attach from
 * white rook ?
 */
if (blk_king & wht_rook_mv2)
    printf("Yes\n");
```

---

More Bitvector Magic

Count the number of 1’s in a word

MIT Hackmem 169:

```
int bitcount(unsigned int n)
{
    unsigned int tmp;
    tmp = n - ((n >> 1) & 03333333333333)
         - ((n >> 2) & 0111111111111111);
    return ((tmp + (tmp >> 3)) & 0307070707070707) % 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^3 + 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