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

Bits and Bytes
January 15, 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
  - $1.213 \times 10^1$ (1.213e4)

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, PACE 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 $111011011011_{2}$
- Represent $1.20_{10}$ as $1.0011001100111[0011]_{2}$
- Represent $1.5213 \times 10^4$ as $1.110111011101_{2} \times 2^{13}$

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 to 11111111
  - Decimal: 0 to 255
  - First digit must not be 0 in C
- Octal: 000 to 0377
  - Use leading 0 in C
- Hexadecimal 00 to FF
  - Base 16 number representation
  - Use characters '0' to '9' and 'A' to 'F'
  - Write FA1D37B in C as 0xFA1D37B
  - Or 0xFA1d37b

Literary Hex

Common 8-byte hex filler:
- Oxdeadbeef
- 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 are 32 bits (4 bytes)
  - Limits addresses to 4GB
  - Becoming too small for memory-intensive applications
- High-end systems are 64 bits (8 bytes)
  - Potential address space > 1.8 X 10^20 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 8 10/12
- char*: 8 4 4
- 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
- Alpha’s, 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 4x is 0x100

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

Little Endian: 0x100 0x101 0x102 0x103
<table>
<thead>
<tr>
<th></th>
<th>67</th>
<th>45</th>
<th>23</th>
<th>01</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>0x048365</td>
<td>0b</td>
<td>pop wdx</td>
</tr>
<tr>
<td>0x048366</td>
<td>81 e3 ab 12 00 00</td>
<td>add 0x12ab, wbx</td>
</tr>
<tr>
<td>0x04836c</td>
<td>83 bb 28 00 00 00 00 00</td>
<td>cmpl 0x0, 0x28 (wax)</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

```c
typedef unsigned char *pointer;
void show_bytes(pointer start, int len)
{
    int i;
    for (i = 0; i < len; i++)
        printf(“%02x\t", start+i, start[i]);
    printf(“\n”);
}
```

Print 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;
0x1f0000 0x5d9f 0x1f0000
0x5d9f 0x1f0000 0x5d9f
```

Representing Integers

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>15213</td>
<td>0011 1011 0110 1101</td>
<td>7B 6D</td>
</tr>
</tbody>
</table>

Linux/Alpha a  Sun a

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

Two’s complement representation (Covered next lecture)

Representing Pointers

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

Alpha Address

<table>
<thead>
<tr>
<th>Hex</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>C</th>
<th>A</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0D</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0D</td>
<td>FF</td>
<td>FF</td>
<td></td>
</tr>
</tbody>
</table>

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

Linux Address

<table>
<thead>
<tr>
<th>Hex</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>B</th>
<th>2</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td></td>
</tr>
</tbody>
</table>

Different compilers & machines assign different locations to objects

Page 4
Representing Floats

Float $F = 1.5213 \times 10^1$:

<table>
<thead>
<tr>
<th>Linux/Alpha</th>
<th>Sun F</th>
</tr>
</thead>
<tbody>
<tr>
<td>08</td>
<td>47</td>
</tr>
<tr>
<td>84</td>
<td>60</td>
</tr>
<tr>
<td>6B</td>
<td>84</td>
</tr>
<tr>
<td>64</td>
<td>02</td>
</tr>
</tbody>
</table>

IEEE Single Precision Floating Point Representation
- Hex: 40 01 01 01 01 00 00 00
- Binary: 0100 0011 0101 1011 0100 0000 0000

15.213: 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
- Each character encoded in ASCII format
- Standard 7-bit encoding of character set
- Character "0" has code 0x30
- Digit / has code 0x31
- 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 = -
  - Mac (\r\n) = 0x0d = -
  - DOS and HTTP (\r\n) = 0x0d0a = -

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;
}
```

<table>
<thead>
<tr>
<th></th>
<th>Alpha sum</th>
<th>Sun sum</th>
<th>PC sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>02</td>
<td>02</td>
<td>02</td>
<td>02</td>
</tr>
<tr>
<td>03</td>
<td>03</td>
<td>03</td>
<td>03</td>
</tr>
<tr>
<td>04</td>
<td>04</td>
<td>04</td>
<td>04</td>
</tr>
<tr>
<td>05</td>
<td>05</td>
<td>05</td>
<td>05</td>
</tr>
<tr>
<td>06</td>
<td>06</td>
<td>06</td>
<td>06</td>
</tr>
<tr>
<td>07</td>
<td>07</td>
<td>07</td>
<td>07</td>
</tr>
<tr>
<td>08</td>
<td>08</td>
<td>08</td>
<td>08</td>
</tr>
<tr>
<td>09</td>
<td>09</td>
<td>09</td>
<td>09</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</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**
- $A \& B = 1$ when both $A = 1$ and $B = 1$
- $A \& B = 0$ otherwise

**Or**
- $A \lor B = 1$ when either $A = 1$ or $B = 1$
- $A \lor B = 0$ otherwise

**Not**
- $\sim A = 1$ when $A = 0$
- $\sim A = 0$ when $A = 1$

**Exclusive-Or (Xor)**
- $A \oplus B = 1$ when either $A = 1$ or $B = 1$, but not both

### 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**
- $A \& \neg B \lor \neg A \& B$

### Integer Algebra

**Integer Arithmetic**
- $\langle \mathbb{Z}, +, \cdot, -, 0, 1 \rangle$ forms a “ring”
- Addition is “sum” operation
- Multiplication is “product” operation
- $-a$ is additive inverse
- $0$ is identity for sum
- $1$ is identity for product

### Boolean Algebra

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

- Commutativity
  \( A \land B = B \land A \)
  \( A + B = B + A \)

- Associativity
  \( (A \land B) \land C = A \land (B \land C) \)
  \( (A + B) + C = A + (B + C) \)

- Product distributes over sum
  \( A \land (B + C) = (A \land B) + (A \land C) \)
  \( A + (B \land C) = (A + B) \land (A + C) \)

- Sum and product identities
  \( A \land 0 = 0 \)
  \( A + 0 = A \)

- Zero is product annihilator
  \( A \land 0 = 0 \)

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

Boolean Algebra ≠ Integer Ring

- Boolean: Sum distributes over product
  \( A \land (B + C) = (A \land B) + (A \land C) \)
  \( A + (B \land C) = (A + B) \land (A + C) \)

- Boolean: Idempotency
  \( A \land A = A \)
  \( A + A = A \)

- Boolean: Absorption
  \( A \land (A + B) = A \)
  \( A + (A \land B) = A \)

- Boolean: Laws of Complements
  \( \neg \neg A = A \)
  \( \neg A = \neg \neg \neg A \)

- Boolean: Every element has additive inverse
  \( A \land A = 0 \)
  \( A + A = 0 \)

Properties of & and ^

- \((0,1), ^, \land, \lor, \land, \lor, 1\)
- Identical to integers mod 2
- \(I\) is identity operation: \(I(A) = A\)
  \(A \land 0 = 0\)

Property
- 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
  \( A \land (B \lor C) = (A \land B) \lor (A \land C) \)

- 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
  \( A \land A = 0 \)

Relations Between Operations

DeMorgan’s Laws
- \( A \lor \neg B = \neg (A \land \neg B) \)
- \( A \land \neg B = \neg (A \lor \neg B) \)
- \( A \lor B \land A \land B \land 0 \land 1 \leq 1 \)
- \( A \land B \lor \neg A \land \neg B \land 0 \land 1 \leq 1 \)
- \( A \lor B = \neg (A \land \neg B) \)
- \( A \land B = \neg (A \lor \neg B) \)

Exclusive-Or using Inclusive Or
- \( A \lor B = (A \land B) \lor (A \land \neg B) \land (A \land \neg B) \)
- \( A \land B = (A \lor B) \land (A \land B) \)

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

Operate on Bit Vectors

- Operations applied bitwise
  - 01010101
  - 01010001
  - 01011001
  - 01011010
  - 01011100
  - 01011110
  - 01011111

All of the Properties of Boolean Algebra Apply

Representing & Manipulating Sets

Representation

- Width w bit vector represents subsets of \([0, \ldots, w-1]\)
- \(2^n \text{ if } A \)
- \(01010001 \quad (0, 2, 4, 6)\)
- \(01111101 \quad (0, 2, 3, 4, 5, 6)\)
- \(00111100 \quad (2, 3, 4, 5)\)
- \(10101001 \quad (1, 3, 5, 7)\)

Operations

- \& Intersection
- \| Union
- ^ Symmetric difference
- ~ Complement

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 \(\rightarrow\) 0x6E
- 0x00 \(\rightarrow\) 0xFF
- 0x69 \& 0x55 \(\rightarrow\) 0x41
- 0x10101001 \& 0x1010101 \(\rightarrow\) 0x00000001
- 0x69 | 0x55 \(\rightarrow\) 0x7D
- 0x1010101 | 0x10101001 \(\rightarrow\) 0x11110111

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 \(\rightarrow\) 0x00
- !0x00 \(\rightarrow\) 0x01
- !0x41 \(\rightarrow\) 0x01
- 0x69 && 0x55 \(\rightarrow\) 0x61
- 0x69 || 0x55 \(\rightarrow\) 0x61
- 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</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&lt; 3</td>
<td>00000000</td>
</tr>
<tr>
<td>Log. &gt;&gt; 2</td>
<td>00010000</td>
</tr>
<tr>
<td>Arith. &gt;&gt; 2</td>
<td>00110000</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>*x</th>
<th>*y</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</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>
</tbody>
</table>

### More Fun with Bitvectors

Bitboard representation of chess positions:
```
unsigned long long blk_king, wht_king, wht_rook_mw2, ...
```

```c
wht_king = 0x0000000000001000b1;
blk_king = 0x0000000000000100b1;
wht_rook_mw2 = 0x10ef10f10101010b1;
```

/* Is black king under attack from white rook? */
if (blk_king & wht_rook_mw2)
    printf("Tsk\n");
```

### More Bitvector Magic

Count the number of 1's in a word
```
int bitcount(unsigned int n)
{
    unsigned int tmp;
    tmp = n - ((n >> 1) & 033333333333);
    return ((n >> 2) & 011111111111) + (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^{18} + x^{14} + x^{12} + 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