Bits and Bytes

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 $1.1101101101101_2 \times 2^{11}$

Electronic Implementation
- Easy to store with bistable elements
- Reliably transmitted on noisy and inaccurate wires
- Approximate 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\textsubscript{2} to 11111111\textsubscript{2}
- Decimal: 0\textsubscript{10} to 255\textsubscript{10}
- Hexadecimal: 00\textsubscript{16} to FF\textsubscript{16}
- Base 16 number representation
- Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
- Write FA1D37B\textsubscript{16} in C as 0xFA1D37B
  - Or 0xFA1D37B
- Or 0xfa1d37b

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 \(1 \times 10^{19}\) 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>Address of first byte in word</th>
<th>32-bit Words</th>
<th>64-bit Words</th>
<th>Bytes</th>
<th>Addr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td></td>
<td></td>
<td></td>
<td>0000</td>
</tr>
<tr>
<td>0001</td>
<td></td>
<td></td>
<td></td>
<td>0001</td>
</tr>
<tr>
<td>0002</td>
<td></td>
<td></td>
<td></td>
<td>0002</td>
</tr>
<tr>
<td>0003</td>
<td></td>
<td></td>
<td></td>
<td>0003</td>
</tr>
<tr>
<td>0004</td>
<td></td>
<td></td>
<td></td>
<td>0004</td>
</tr>
<tr>
<td>0005</td>
<td></td>
<td></td>
<td></td>
<td>0005</td>
</tr>
<tr>
<td>0006</td>
<td></td>
<td></td>
<td></td>
<td>0006</td>
</tr>
<tr>
<td>0007</td>
<td></td>
<td></td>
<td></td>
<td>0007</td>
</tr>
<tr>
<td>0008</td>
<td></td>
<td></td>
<td></td>
<td>0008</td>
</tr>
<tr>
<td>0009</td>
<td></td>
<td></td>
<td></td>
<td>0009</td>
</tr>
<tr>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td>0010</td>
</tr>
<tr>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td>0011</td>
</tr>
<tr>
<td>0012</td>
<td></td>
<td></td>
<td></td>
<td>0012</td>
</tr>
<tr>
<td>0013</td>
<td></td>
<td></td>
<td></td>
<td>0013</td>
</tr>
<tr>
<td>0014</td>
<td></td>
<td></td>
<td></td>
<td>0014</td>
</tr>
<tr>
<td>0015</td>
<td></td>
<td></td>
<td></td>
<td>0015</td>
</tr>
</tbody>
</table>

Data Representations

Sizes of C Objects (in Bytes)
- C Data Type | Compaq Alpha | Typical 32-bit | Intel IA32 |
- 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
- 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>Little Endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>67 45 23 01</td>
<td>01 23 45 67</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>8048355</td>
<td>5b</td>
<td>pop %ebx</td>
</tr>
<tr>
<td>8048356</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 cmpl $0x0,0x28(%ebx)</td>
<td></td>
</tr>
</tbody>
</table>

Deciphering Numbers
- Value: 0x12ab
- Pad to 4 bytes: 000012ab
- 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	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):

```c
int a = 15213;
0x11ffffcb8 0x6d
0x11ffffcb9 0x3b
0x11ffffcba 0x00
0x11ffffcbb 0x00
```

Representing Integers

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

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

Linux/Alpha A

Sun A

```

Representing Pointers

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

Alpha Address

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

Sun Address

Hex: E F F F F B 2 C
Binary: 1110 1111 1111 1111 1111 1111 1010 1100

Different compilers & machines assign different locations to objects
Representing Floats

Float F = 15213.0;

<table>
<thead>
<tr>
<th>Hex</th>
<th>Linux/Alpha</th>
<th>Sun F</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>4E</td>
<td>6D</td>
</tr>
<tr>
<td>6D</td>
<td>00</td>
<td>46</td>
</tr>
<tr>
<td>46</td>
<td>00</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 1101 1011 0100 0000 0000
15213:

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
- Other encodings exist, but uncommon
- Character "0" has code \text{0x30} \text{i}
  - Digit \text{i} has code \text{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)!

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

```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
- PC uses 7 instructions with lengths 1, 2, and 3 bytes
  - Same for NT and for Linux
  - NT/Linux not fully binary compatible

<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>81</td>
<td></td>
<td>55</td>
</tr>
<tr>
<td>00</td>
<td>81</td>
<td>03</td>
<td>89</td>
</tr>
<tr>
<td>30</td>
<td>60</td>
<td>85</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>08</td>
<td>82</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>90</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>02</td>
<td>03</td>
<td>0C</td>
<td></td>
</tr>
<tr>
<td>7A</td>
<td>05</td>
<td>01</td>
<td></td>
</tr>
<tr>
<td>09</td>
<td>07</td>
<td>02</td>
<td></td>
</tr>
</tbody>
</table>

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 \( B = 1 \)
- \( \begin{array}{ccc} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{array} \)

**Or**
- \( A \lor B = 1 \) when either \( A = 1 \) or \( B = 1 \)
- \( \begin{array}{ccc} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{array} \)

**Not**
- \( \neg A = 1 \) when \( A = 0 \)
- \( \neg A = 0 \) when \( A = 1 \)
- \( \begin{array}{ccc} 0 & 1 & 1 \\ 1 & 0 & 0 \end{array} \)

**Exclusive-Or (Xor)**
- \( A \oplus B = 1 \) when either \( A = 1 \) or \( B = 1 \), but not both
- \( \begin{array}{ccc} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{array} \)

---

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
- \( A \& \neg B \lor \neg A \& B = A \oplus B \)
Integer Algebra

Integer Arithmetic
- \((\mathbb{Z}, +, *, - , 0, 1)\) 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
- \(\langle\{0,1\}, |, &, ~, 0, 1\rangle\) 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

### Boolean Algebra = Integer Ring

- **Commutativity**
  - \(A | B = B | A\)
  - \(A + B = B + A\)
  - \(A * B = B * A\)
- **Associativity**
  - \((A | B) | C = A | (B | C)\)
  - \((A + B) + C = A + (B + C)\)
  - \((A & B) & C = A & (B & C)\)
  - \((A * B) * C = A * (B * C)\)
- **Product distributes over sum**
  - \(A & (B | C) = (A & B) | (A & C)\)
  - \(A * (B + C) = A * B + B * C\)
- **Sum and product identities**
  - \(A | 0 = A\)
  - \(A + 0 = A\)
  - \(A & 1 = A\)
  - \(A * 1 = A\)
- **Zero is product annihilator**
  - \(A & 0 = 0\)
  - \(A * 0 = 0\)
- **Cancellation of negation**
  - \(\sim (\sim A) = A\)
  - \(\sim (\sim A) = A\)
Boolean Algebra ≠ Integer Ring

Boolean: Sum distributes over product
\[ A \land (B \lor C) = (A \land B) \lor (A \land C) \quad \land \quad (A \lor B) \lor (A \lor C) = (A \lor B) \lor (A \lor C) \]

Boolean: Idempotency
\[ \forall A, A \land A = A \quad \lor \quad A \lor A = A \]

“A is true” or “A is true” = “A is true”

Boolean: Absorption
\[ A \land (A \lor B) = A \quad \lor \quad A \lor (A \land B) = A \]

“A is true” or “A is true and B is true” = “A is true”

Boolean: Laws of Complements
\[ \neg A \land \neg A = 1 \quad \lor \quad A \lor \neg A = 1 \]

“\neg A is true” or “\neg A is false”

Ring: Every element has additive inverse
\[ A \land 0 = 0 \quad \lor \quad A \lor \neg A = 0 \]

Properties of \( \land \) and \( \lor \)

\( (\{0, 1\}, \lor, \land, 0, 1) \)

Identical to integers mod 2

\( 1 \) is identity operation: \( 1 \land A = A \)

\( \land \lor 0 \)

Properties of \( \land \) and \( \lor \)

- Commutative sum \( A \land B = B \land A \)
- Commutative product \( A \lor B = B \lor A \)
- Associative sum \( (A \land B) \land C = A \land (B \land C) \)
- Associative product \( (A \lor B) \lor C = A \lor (B \lor C) \)
- Prod. over sum \( A \land (B \lor C) = (A \land B) \lor (B \land C) \)
- Prod. over sum \( A \land 0 = 0 \)
- Prod. over sum \( A \land 1 = A \)
- 0 is identity operation \( A \land 0 = 0 \)
- 0 is product annihilator \( A \land 0 = 0 \)
- Additive inverse \( A \land 0 = 0 \)

DeMorgan’s Laws
- Express \& in terms of \lor, and vice-versa
  - \( A \lor B = \neg \neg A \lor \neg B \)
  - A \lor B is true if and only if neither A nor B is false
  - \( A \lor B = \neg \neg A \lor \neg B \)
  - A or B is true if and only if A and B are not both false

Exclusive-Or using Inclusive Or
- \( A \lor B = (A \land \neg B) \lor (B \land \neg A) \)
  - Exactly one of A and B is true
- \( A \lor B = (A \land \neg B) \lor (B \land \neg A) \)
  - Either A is true, or B is true, but not both
General Boolean Algebras

Operate on Bit Vectors

- Operations applied bitwise
  - $01101001 \& 01010101 = 01000001$
  - $01101001 \mid 01010101 = 01111101$
  - $01101001 \^ 01010101 = 00111100$
  - $\sim 01010101 = 10101010$

All of the Properties of Boolean Algebra Apply

Representing & Manipulating Sets

Representation
- Width $w$ bit vector represents subsets of $\{0, \ldots, w-1\}$
- $a = \{1, 2, 3, 5, 6\}$

\[
\begin{align*}
01101001 & \quad \{0, 3, 5, 6\} \\
01010101 & \quad \{0, 2, 4, 6\}
\end{align*}
\]

Operations
- $\&$ Intersection
- $\mid$ Union
- $\^$ Symmetric difference
- $\sim$ Complement

\[
\begin{align*}
\& 01000001 & \quad \{0, 6\} \\
\mid 01111101 & \quad \{0, 2, 3, 4, 5, 6\} \\
\^ 00111100 & \quad \{2, 3, 4, 5\} \\
\sim 10101010 & \quad \{1, 3, 5, 7\}
\end{align*}
\]

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)
- $\sim 0x41 \rightarrow 0xBE$
- $0x69 \& 0x55 \rightarrow 0x41$
- $0x00 \rightarrow 0xFF$
- $0x69 | 0x55 \rightarrow 0x7D$
- $0x00 \rightarrow 0x00$

Spring 2006
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
- \(0x69 \&\& 0x55\) \(\rightarrow\) 0x01
- \(0x69 \|\| 0x55\) \(\rightarrow\) 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
- Arguments: \(10100010\) (before), \(00010\) (after)

Right Shift: \(x >> y\)
- Shift bit-vector \(x\) right \(y\) positions
- Logical shift
- Fill with 0's on left
- Arguments: \(10100010\) (before), \(000010\) (after)

Cool Stuff with Xor

- Bitwise Xor is form of addition
- With extra property that every value is its own additive inverse
- \(A \oplus A = 0\)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></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(\oplus)B) (\oplus)B = A</td>
</tr>
<tr>
<td>3</td>
<td>(A(\oplus)B) (\oplus)A = B</td>
</tr>
<tr>
<td></td>
<td>A</td>
</tr>
</tbody>
</table>

void funny(int *, int *)
{
    x = x \(\oplus\) y;  /* #1 */
    y = x \(\oplus\) y;  /* #2 */
    x = x \(\oplus\) y;  /* #3 */
}
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