Bits and Bytes
Why don’t computers use Base 10?

- **Base 10 number representation**
  - “Digit” in many languages also refers to fingers (and toes)
    - Decimal (from latin decimus) means tenth
  - A position numeral system (unlike, say Roman numerals)
  - Natural representation for financial transactions (problems?)
  - Even carries through in scientific notation

- **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
  - Harder 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
  - Straightforward implementation of arithmetic functions
Byte-oriented memory organization

- Programs refer to virtual addresses
  - Conceptually very large array of bytes (byte = 8 bits)
  - 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 manipulate 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
How do we represent the address space?

- **Hexadecimal notation**
  - Base 16 number representation
  - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
  - E.g., \( \text{FA1D37B}_{16} \)
    - In C, 0xFA1D37B or 0xfa1d37b
  - Each digit unpacks directly to binary
    - \( \text{A9} \) unpacks to 1010 1001

- **Byte = 8 bits**
  - Binary: 00000000\(_2\) to 11111111\(_2\)
  - Decimal: 0\(_{10}\) to 255\(_{10}\)
  - Hexadecimal: 00\(_{16}\) to FF\(_{16}\)
Checkpoint
Checkpoint
What about Octal?

- Octal notation:
  - Digits 0 through 7, e.g., 7120
    - In C, C++, Java, Javascript…, signaled with leading 0, e.g., 077
    - Source of surprise in things like `new Date(09/11/2011)`
  - Encodes 3 bits at a time
  - Like hex, unpacks directly to binary
  - Unlike hex, no extra digit characters needed

- Used to be a serious competitor to hex
  - Unix `od` command stands for "octal dump"
  - Older architectures had word sizes divisible by 3, e.g., 24, 36, 60

- Octal needed to understand this riddle:
  - Why do programmers confuse Halloween and Christmas?
    - Because 31 OCT = 25 DEC
Machine words

- Machine has “word size”
  - Nominal size of integer-valued data
    - Including addresses
    - A virtual address is encoded by such a word
  - 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)

<table>
<thead>
<tr>
<th>Addr.</th>
<th>Bytes</th>
<th>32-bit Words</th>
<th>64-bit Words</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td></td>
<td>Addr = 0000</td>
<td>Addr = 0000</td>
</tr>
<tr>
<td>0001</td>
<td></td>
<td>Addr = 0004</td>
<td></td>
</tr>
<tr>
<td>0002</td>
<td></td>
<td>Addr = 0008</td>
<td></td>
</tr>
<tr>
<td>0003</td>
<td></td>
<td>Addr = 0012</td>
<td></td>
</tr>
<tr>
<td>0004</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0005</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0006</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0007</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0008</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0009</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0012</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0013</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0014</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0015</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Wednesday, September 28, 2011
Data representations

- Sizes of C Objects (in Bytes)

<table>
<thead>
<tr>
<th>C Data type</th>
<th>Compaq Alpha</th>
<th>Typical 32b</th>
<th>Intel IA32</th>
</tr>
</thead>
<tbody>
<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</td>
<td>8</td>
<td>10/12</td>
</tr>
<tr>
<td>Char * (any pointer)</td>
<td>8</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

- Portability:
  - Many programmers assume that object declared as int can be used to store a pointer
    - OK for a typical 32-bit machine
    - Not for Alpha
Byte ordering

- How to order bytes within multi-byte word in memory
- Conventions
  - Sun’s, Mac’s are “Big Endian” machines
    - Least significant byte has highest address (comes last)
  - Alphas, PC’s are “Little Endian” machines
    - Least significant byte has lowest address (comes first)
- Example
  - Variable $x$ has 4-byte representation $0x01234567$
  - Address given by $\&x$ is $0x100$

**Big Endian**

```
0x100 0x101 0x102 0x103

01 23 45 67
```

**Little Endian**

```
0x100 0x101 0x102 0x103

67 45 23 01
```
Reading byte-reversed Listings

- For most programmers, these issues are invisible
- Except with networking or 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>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
Representing strings in C

- A null-terminated array of characters
  - Final character = 0
- Each character encoded in 7-bit ASCII format
  - Other encodings exist, but uncommon
  - “0” has code 0x30
  - Digit i has code 0x30+i
- Compatibility
  - Byte ordering not an issue
    - Data are single byte quantities
  - Text files generally platform independent
    - Except for different line termination character(s)!

```c
char S[6] = "15213";
```

<table>
<thead>
<tr>
<th>Linux/Alpha</th>
<th>Sun</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>33</td>
<td>33</td>
</tr>
<tr>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>35</td>
<td>35</td>
</tr>
</tbody>
</table>
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

- A fundamental concept:
  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 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

<table>
<thead>
<tr>
<th>~A</th>
<th>A &amp; B</th>
<th>A</th>
<th>A ^ B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 0</td>
<td>1</td>
<td>0 1 0</td>
</tr>
<tr>
<td>1</td>
<td>1 0 1</td>
<td>1</td>
<td>1 1 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

\[ A \& \sim B \quad \sim A \& B = A \wedge B \]

Connection when

\[ A \& \sim B \mid \sim A \& B \]

\[ = A \wedge B \]
Integer  Boolean algebra

- **Integer Arithmetic**
  \[ \langle \mathbb{Z}, +, *, -, 0, 1 \rangle \] forms a mathematical structure called “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 mathematical structure called “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

EECS 213 Introduction to Computer Systems
## Boolean Algebra ≈ Integer Ring

<table>
<thead>
<tr>
<th>Property</th>
<th>Algebraic Exponentiation</th>
</tr>
</thead>
</table>
| **Commutativity**               | $A | B = B | A$  
|                                 | $A & B = B & A$                                                                        |
| **Associativity**               | $(A | B) | C = A | (B | C)$  
|                                 | $(A & B) & C = A & (B & C)$                                                            |
| **Product distributes over sum**| $A & (B | C) = (A & B) | (A & C)$  
| **Sum and product identities**  | $A | 0 = A$  
|                                 | $A & 1 = A$                                                                             |
| **Zero is product annihilator** | $A & 0 = 0$                                                                             |
|                                 | $A * 0 = 0$                                                                             |
| **Cancellation of negation**    | $\sim (\sim A) = A$                                                                     |
|                                 | $\neg (\neg A) = A$                                                                     |
## Boolean Algebra ≠ Integer Ring

| Boolean, not Ring: Sum distributes over product | $A | (B \& C) = (A | B) \& (A | C)$ | $A + (B * C) \neq (A + B) * (B + C)$ |
|------------------------------------------------|-----------------------------------|-----------------------------------|
| Boolean, not Ring: Idempotency                  | $A | A = A$                          | $A + A \neq A$                    |
|                                                | $A \& A = A$                      | $A \neq A$                        |
| Boolean, not Ring: Absorption                  | $A | (A \& B) = A$                   | $A + (A * B) \neq A$              |
|                                                | $A \& (A | B) = A$                 | $A * (A + B) \neq A$              |
| Boolean, not Ring: Laws of Complements          | $A | \sim A = 1$                    | $A + \sim A \neq 1$               |
| Ring, not Boolean: Every element has additive inverse | $A | \sim A \neq 0$                  | $A + \sim A = 0$                  |

EECS 213 Introduction to Computer Systems
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 \( A ^ A = 0 \)
Checkpoint
Relations between operations

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

- **Exclusive-Or using Inclusive Or**
  - \(A \wedge B = (~A \& B) \| (A \& ~B)\)
    - Exactly one of \(A\) and \(B\) is true
  - \(A \wedge B = (A \| B) \sim (A \& B)\)
    - Either \(A\) is true, or \(B\) is true, but not both
General Boolean algebras

- We can extend the four Boolean operations to also operate on bit vectors
  - Operations applied bitwise
    
    \[
    \begin{array}{ccc}
    01101001 & 01101001 & 01101001 \\
    \& 01010101 & \mid 01010101 & \wedge 01010101 & \sim 01010101 \\
    \hline
    01000001 & 01111101 & 00111100 & 10101010
    \end{array}
    \]

- All of the Properties of Boolean Algebra Apply

- Resulting algebras:
  - Boolean algebra: $\langle \{0,1\}(w), |, \&, \sim, 0(w), 1(w) \rangle$
  - Ring: $\langle \{0,1\}(w), \wedge, \&, I, 0(w), 1(w) \rangle$
Representing manipulating sets

- Useful application of bit vectors – represent finite sets

- Representation
  - Width w bit vector represents subsets of \{0, \ldots, w–1\}
  - \(a_j = 1\) if \(j \in A\)
    - 01101001 represents \{ 0, 3, 5, 6 \}
    - 01010101 represents \{ 0, 2, 4, 6 \}

- 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
  - View arguments as bit vectors
  - Arguments applied bit-wise

- Examples (Char data type)
  - ~0x41 --> 0xBE
    - ~01000001<sub>2</sub> --> 10111110<sub>2</sub>
  - ~0x00 --> 0xFF
    - ~00000000<sub>2</sub> --> 11111111<sub>2</sub>
  - 0x69 & 0x55 --> 0x41
    - 01101001<sub>2</sub> 01010101<sub>2</sub> --> 01000001<sub>2</sub>
  - 0x69 | 0x55 --> 0x7D
    - 01101001<sub>2</sub> | 01010101<sub>2</sub> --> 01111101<sub>2</sub>
Logic operations in C – not quite the same

• Contrast to logical operators
  – &&, ||, !
    • View 0 as “False”
    • Anything nonzero as “True”
    • Always return 0 or 1
    • Early termination (if you can answer looking at first argument, you are done)

• 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

<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>
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