In-Class Handout Version  
15-213  
“The Class That Gives CMU Its Zip!”  
Bits and Bytes  
Aug. 30, 2001

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 $1.001100110011011[-0011]…_2$
• 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

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
- Hexadecimal: 0₀₁₆ to FF₁₆
  - Base 16 number representation
  - Use characters '0' to '9' and 'A' to 'F'
  - Write FA1D37B₁₆ 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>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>

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 ≈ 1.8 x 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>32-bit Words</th>
<th>64-bit Words</th>
<th>Bytes</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr =</td>
<td>Addr =</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>Addr =</td>
<td>Addr =</td>
<td>0001</td>
<td></td>
</tr>
<tr>
<td>Addr =</td>
<td>Addr =</td>
<td>0002</td>
<td></td>
</tr>
<tr>
<td>Addr =</td>
<td>Addr =</td>
<td>0003</td>
<td></td>
</tr>
<tr>
<td>Addr =</td>
<td>Addr =</td>
<td>0004</td>
<td></td>
</tr>
<tr>
<td>Addr =</td>
<td>Addr =</td>
<td>0005</td>
<td></td>
</tr>
<tr>
<td>Addr =</td>
<td>Addr =</td>
<td>0006</td>
<td></td>
</tr>
<tr>
<td>Addr =</td>
<td>Addr =</td>
<td>0007</td>
<td></td>
</tr>
<tr>
<td>Addr =</td>
<td>Addr =</td>
<td>0008</td>
<td></td>
</tr>
<tr>
<td>Addr =</td>
<td>Addr =</td>
<td>0009</td>
<td></td>
</tr>
<tr>
<td>Addr =</td>
<td>Addr =</td>
<td>0010</td>
<td></td>
</tr>
<tr>
<td>Addr =</td>
<td>Addr =</td>
<td>0011</td>
<td></td>
</tr>
<tr>
<td>Addr =</td>
<td>Addr =</td>
<td>0012</td>
<td></td>
</tr>
<tr>
<td>Addr =</td>
<td>Addr =</td>
<td>0013</td>
<td></td>
</tr>
<tr>
<td>Addr =</td>
<td>Addr =</td>
<td>0014</td>
<td></td>
</tr>
<tr>
<td>Addr =</td>
<td>Addr =</td>
<td>0015</td>
<td></td>
</tr>
</tbody>
</table>

Data Representations

Sizes of C Objects (in Bytes)

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Compaq Alpha</th>
<th>Typical 32-bit</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 *</td>
<td>8</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

» Or any other pointer
Byte Ordering

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

Conventions
- Alphas, PC's are "Little Endian" machines
- Least significant byte has lowest address
- Sun's, Mac's are "Big Endian" machines
- Least significant byte has highest address

Example
- Variable x has 4-byte representation 0x01234567
  - Address given by &x is 0x100

Big Endian
- 0x100 0x101 0x102 0x103

Little Endian
- 0x100 0x101 0x102 0x103

Exercising Data Representations

Representing Integers
- Casting pointer to unsigned char * creates byte array

typedef unsigned char *pointer;

void show_bytes(pointer start, int len)
{
    int i;
    for (i = 0; i < len; i++)
        printf("0x%02lx
", start+i, start[i]);
    printf("\n");
}

Printf directives:
%p: Print pointer
%x: Print Hexadecimal

show_bytes Execution Example

int a = 15213;
printf("int a = 15213;\n");
show_bytes((pointer) &a, sizeof(int));

Result:

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

Examining Data Representations

Integers
- Casting pointer to unsigned char * creates byte array

typedef unsigned char *pointer;

void show_bytes(pointer start, int len)
{
    int i;
    for (i = 0; i < len; i++)
        printf("0x%02lx
", start+i, start[i]);
    printf("\n");
}

Printf directives:
%p: Print pointer
%x: Print Hexadecimal

show_bytes Execution Example

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

Dec: 15213
Bin: 0011 1011 0110 1101
Hex: 3B 6D

Linux/Alpha A
- 6D 3B 00 00
- 00 3B 00 00
- 00 00 00 00

Sun A
- 6D 3B 00 00
- 00 3B 00 00
- 00 00 00 00

Linux/Alpha B
- 93 FF 00 00
- C4 FF 00 00
- FF C4 FF FF

Sun B
- 93 FF 00 00
- C4 FF 00 00
- FF C4 FF FF

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>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F</td>
<td>0001 1111 1111 1111 1111 1111 1111 1111 1110 1010 0000</td>
</tr>
</tbody>
</table>

Sun Address

<table>
<thead>
<tr>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1F</td>
<td>0001 1111 1111 1111 1111 1111 1111 1111 1010 0000</td>
</tr>
</tbody>
</table>

Representing Floats

```c
Float F = 15213.0;
```

IEEE Single Precision Floating Point Representation

<table>
<thead>
<tr>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>46</td>
<td>0001 1110 1101 1011 0100 0000 0000</td>
</tr>
</tbody>
</table>

Not same as integer representation, but consistent across machines

Representing Strings

Strings in C

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

- 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 0x30
    - Digit “0” has code 0x30+/
- 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!

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

**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></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 (||)**
- \( A \| B = 1 \) 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></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>1</td>
<td>0</td>
</tr>
</tbody>
</table>

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

<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>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

![Connection diagram]

- \( A \& \sim B \mid \sim A \& B = A^B \)

### Properties of & and | Operations

**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**
- \( \langle \{0,1\}, |, \& , \sim, 0, 1 \rangle \) 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
Properties of Rings & Boolean Algebras

**Boolean Algebra**

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

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

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

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

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

- **Cancellation of negation**
  \[ \neg (\neg A) = A \]

**Integer Ring**

- **Commutativity**
  \[ A + B = B + A \]
  \[ A \cdot B = B \cdot A \]

- **Associativity**
  \[ (A + B) + C = A + (B + C) \]
  \[ (A \cdot B) \cdot C = A \cdot (B \cdot C) \]

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

- **Sum and product identities**
  \[ 0 = A + 0 \]
  \[ 1 = A \cdot 1 \]

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

- **Cancellation of negation**
  \[ \neg (\neg A) = A \]

**Ring ≠ Boolean Algebra**

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

- **Boolean: Idempotency**
  \[ A \lor A = A \]
  \[ A \land A = A \]

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

- **Boolean: Laws of Complements**
  \[ A \lor \neg A = 1 \]
  \[ A \land \neg A = 0 \]

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

Properties of & and ^

**Boolean Ring**

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

**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**
  \[ A \lor A = 0 \]

**Relations Between Operations**

**DeMorgan’s Laws**

- Express & in terms of |, and vice-versa
  \[ A \land B = \neg(\neg A \lor \neg B) \]
  \[ A \lor B = \neg(\neg A \land \neg B) \]

- A and B are true if and only if neither A nor B is false
  \[ A \land B = \neg(\neg A \lor \neg B) \]
  \[ \neg(\neg A \lor \neg B) = \neg A \lor \neg B \]

**Exclusive-Or using Inclusive Or**

- \[ A \lor B = \neg(A \land B) \land (A \lor B) \]
  \[ A \lor B = \neg A \land B \lor A \land \neg B \]

- Exactly one of A and B is true
  \[ A \lor B = \neg A \land B \lor A \land \neg B \]

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

Operate on Bit Vectors
- Operations applied bitwise

01101001 01101001 01101001
& 01010101 | 01010101 ^ 01010101 ~ 01010101

Representation of Sets
- Width \( w \) bit vector represents subsets of \{0, ..., w−1\}
- \( a_j = 1 \) if \( j \in A \)
- \( \neg 0101001 \)
- \( \neg 01010101 \)
- \& Union
- ^ Intersection
- ~ Complement

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 -->
- !!0x41 -->
- 0x69 & 0x55 -->
- 0x69 | 0x55 -->
- p & *p (avoids null pointer access)

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 -->
- ~01000001 --> 10111102
- ~0x00 -->
- ~00000002 --> 11111112
- 0x69 & 0x55 -->
- 01101001 \& 01010101 --> 01000001
- 0x69 | 0x55 -->
- 01101001 \| 01010101 --> 01111112

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></td>
</tr>
<tr>
<td>Log. &gt;&gt; 2</td>
<td></td>
</tr>
<tr>
<td>Arith. &gt;&gt; 2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument x</th>
<th>10100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>( &lt;&lt; 3 )</td>
<td></td>
</tr>
<tr>
<td>Log. &gt;&gt; 2</td>
<td></td>
</tr>
<tr>
<td>Arith. &gt;&gt; 2</td>
<td></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
  \( A \oplus A = 0 \)

```c
void funny(int *x, int *y)
{
    *x = *x ^ *y;    /* #1 */
    *y = *x ^ *y;    /* #2 */
    *x = *x ^ *y;    /* #3 */
}
```

<table>
<thead>
<tr>
<th>Step</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(^\lor)B</td>
<td>B</td>
</tr>
<tr>
<td>2</td>
<td>A(^\lor)B</td>
<td>B</td>
</tr>
</tbody>
</table>
| 3    | A \(
| End  |   |   |

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