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 $111011011011_2$
- Represent $1.20_{10}$ as $1.0011001100110011…_{2}$
- 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 \(00000000_2\) to \(11111111_2\)
- Decimal: \(0_{10}\) to \(255_{10}\)
- 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>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 \(\approx 1.8 \times 10^{19}\) 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>32-bit Words</th>
<th>64-bit Words</th>
<th>Bytes</th>
<th>Addr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr = 0000</td>
<td>Addr = 0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>Addr = 0000</td>
<td>Addr = 0004</td>
<td>0001</td>
<td>0001</td>
</tr>
<tr>
<td>Addr = 0004</td>
<td>Addr = 0000</td>
<td>0002</td>
<td>0002</td>
</tr>
<tr>
<td>Addr = 0008</td>
<td>Addr = 0006</td>
<td>0003</td>
<td>0003</td>
</tr>
<tr>
<td>Addr = 0004</td>
<td>Addr = 0004</td>
<td>0004</td>
<td>0004</td>
</tr>
<tr>
<td>Addr = 0010</td>
<td>Addr = 0008</td>
<td>0005</td>
<td>0005</td>
</tr>
<tr>
<td>Addr = 0008</td>
<td>Addr = 0006</td>
<td>0006</td>
<td>0006</td>
</tr>
<tr>
<td>Addr = 0012</td>
<td>Addr = 0008</td>
<td>0007</td>
<td>0007</td>
</tr>
<tr>
<td>Addr = 0000</td>
<td>Addr = 0008</td>
<td>0008</td>
<td>0008</td>
</tr>
<tr>
<td>Addr = 0004</td>
<td>Addr = 0008</td>
<td>0009</td>
<td>0009</td>
</tr>
<tr>
<td>Addr = 0010</td>
<td>Addr = 0008</td>
<td>0010</td>
<td>0010</td>
</tr>
<tr>
<td>Addr = 0000</td>
<td>Addr = 0008</td>
<td>0011</td>
<td>0011</td>
</tr>
<tr>
<td>Addr = 0012</td>
<td>Addr = 0008</td>
<td>0012</td>
<td>0012</td>
</tr>
<tr>
<td>Addr = 0004</td>
<td>Addr = 0008</td>
<td>0013</td>
<td>0013</td>
</tr>
<tr>
<td>Addr = 0000</td>
<td>Addr = 0008</td>
<td>0014</td>
<td>0014</td>
</tr>
<tr>
<td>Addr = 0012</td>
<td>Addr = 0008</td>
<td>0015</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>Address</th>
<th>Instruction Code</th>
<th>Assembly Rendition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0x101</td>
<td>0x102 0x103</td>
</tr>
<tr>
<td>0x101</td>
<td>0x102</td>
<td>0x103</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Code</th>
<th>Assembly Rendition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0x101</td>
<td>0x102 0x103</td>
</tr>
<tr>
<td>0x101</td>
<td>0x102</td>
<td>0x103</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 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("%p %02x\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**

```c
int A = 15213;
long int C = 15213;
```

**Decimal:**

15,213

**Binary:**

0011 1011 0110 1101

**Hex:**

3 B 6 D

Linux/Alpha A

Sun A

```
6D
3B
00
6D
```

Sun B

```
93
C4
FF
FF
```

Two's complement representation (Covered next lecture)

**Representing Pointers**

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

**Alpha Address**

```c
Hex: 1 F F F F F C A 0
Binary: 0001 1111 1111 1111 1111 1100 1010 0000
```

**Sun Address**

```c
EF
FF
FB
2C
```

**Linux Address**

```c
D4
F8
FF
BF
```

Different compilers & machines assign different locations to objects

**Representing Floats**

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

**IEEE Single Precision Floating Point Representation**

```c
Hex: 4 6 6 D B 4 0 0
Binary: 0100 0110 0110 1101 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

- 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 “i” has code 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)!

Representing Instructions

```c
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>sum</td>
<td>00</td>
<td>B1</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td>00</td>
<td>C3</td>
<td>89</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>E0</td>
<td>E5</td>
</tr>
<tr>
<td></td>
<td>42</td>
<td>08</td>
<td>8B</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>90</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td>80</td>
<td>02</td>
<td>0C</td>
</tr>
<tr>
<td></td>
<td>FA</td>
<td>00</td>
<td>03</td>
</tr>
<tr>
<td></td>
<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

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!

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></th>
<th>And</th>
<th>Or</th>
</tr>
</thead>
<tbody>
<tr>
<td>A&amp;B = 1 when both A=1 and B=1</td>
<td>0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>A</td>
<td>B = 1 when either A=1 or B=1</td>
<td>0 0 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Not</th>
<th>Exclusive-Or (Xor)</th>
</tr>
</thead>
<tbody>
<tr>
<td>~A = 1 when A=0</td>
<td>0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>A^B = 1 when either A=1 or B=1, but not both</td>
<td>0 1 0</td>
<td>0 0 1 1</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 when
  - $A \& \neg B \mid \neg A \& B$
  - $\neg A \& B = A^\wedge B$

---

### 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

---

### Integer Algebra

**Integer Arithmetic**
- $\langle \mathbb{Z}, +, *, \cdot, 0, 1 \rangle$ 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 $\approx$ Integer Ring**

**Commutativity**
- $A \mid B = B \mid A$
- $A \& B = B \& A$
- $A + B = B + A$
- $A * B = B * A$

**Associativity**
- $(A \mid B) \mid C = A \mid (B \mid C)$
- $(A \& B) \& C = A \& (B \& C)$
- $(A + B) + C = A + (B + C)$
- $(A * B) * C = A * (B * C)$

**Product distributes over sum**
- $A \& (B \mid C) = (A \& B) \mid (A \& C)$
- $A * (B + C) = A * B + B * C$

**Sum and product identities**
- $A \mid 0 = A$
- $A \& 1 = A$
- $A + 0 = A$
- $A * 1 = A$

**Zero is product annihilator**
- $A \& 0 = 0$
- $A * 0 = 0$

**Cancellation of negation**
- $\neg (\neg A) = A$
- $\neg (\neg A) = A$
### Boolean Algebra ≠ Integer Ring

**Boolean: Sum distributes over product**

\[ A \land (B \land C) = (A \land B) \land (A \land C) \neq (A \land B) \land (B \land C) \]

**Boolean: Idempotency**

- "A is true" or "A is true" = "A is true"
  \[ A \land A = A \]
  \[ A + A = A \]

**Boolean: Absorption**

- "A is true" or "A is true and B is true" = "A is true"
  \[ A \lor (A \land B) = A \]
  \[ A + (A \cdot B) = A \]

**Boolean: Laws of Complements**

- "A is true" or "A is false"
  \[ A \land \sim A = 1 \]
  \[ A + \sim A = 0 \]

**Ring: Every element has additive inverse**

\[ A \land \sim A \neq 0 \]
\[ A + \sim A \neq 1 \]

---

### Relations Between Operations

**DeMorgan's Laws**

- Express & in terms of |, and vice-versa
  - \[ A \land B = \sim \sim A \lor \sim B \]
    - A and B are true if and only if neither A nor B is false
  - \[ A \lor B = \sim \sim A \land \sim B \]
    - A or B are true if and only if A and B are not both false

**Exclusive-Or using Inclusive Or**

- \[ A \oplus B = (A \land \sim B) \lor (A \land \sim B) \]
  - Exactly one of A and B is true
- \[ A \oplus B = (A \lor B) \land \sim (A \lor B) \]
  - 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>&amp; 01010101</td>
<td>&amp; 01010101</td>
</tr>
<tr>
<td>01111101</td>
<td>00111100</td>
<td>10101010</td>
</tr>
</tbody>
</table>

All of the Properties of Boolean Algebra Apply
Representing & Manipulating Sets

Representation
- Width \( w \) bit vector represents subsets of \( \{0, \ldots, w-1\} \)
- \( a_j = 1 \) if \( j \in A \)

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

Examples
- \( 01101001 \) \( \{0, 3, 5, 6\} \)
- \( 01010101 \) \( \{0, 2, 4, 6\} \)
- \( 01000001 \) \( \{0, 6\} \)
- \( 01111101 \) \( \{0, 2, 3, 4, 5, 6\} \)
- \( 00111100 \) \( \{2, 3, 4, 5\} \)
- \( 10101010 \) \( \{1, 3, 5, 7\} \)

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)
- \( ~0x41 \) --> \( 0xBE \)
- \( ~0x00 \) --> \( 0xFF \)
- \( 0x69 \& 0x55 \) --> \( 0x41 \)
- \( 0x69 \text{ | } 0x55 \) --> \( 0x7D \)
- \( p \&\& \*p \) (avoids null pointer access)

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: \( x << y \)
- 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
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 */
}
```

## 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

<table>
<thead>
<tr>
<th></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^B</td>
<td>B</td>
</tr>
<tr>
<td>2</td>
<td>A^B</td>
<td>(A^B) ^ B = A</td>
</tr>
<tr>
<td>3</td>
<td>(A^B) ^ A = B</td>
<td>A</td>
</tr>
<tr>
<td>End</td>
<td>B</td>
<td>A</td>
</tr>
</tbody>
</table>