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 $152_{10}$ as $11101101101101_2$
- Represent $1.20_{10}$ as $1.0011001100110011[0011]..._2$
- Represent $1.5213 \times 10^4$ as $1.11011011011012 \times 2^{13}$

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
- 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 $\approx 1.8 \times 10^{19}$ 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)
## 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

<table>
<thead>
<tr>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
</table>

Little Endian

<table>
<thead>
<tr>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
</table>
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:

```c
int a = 15213;
0x11fffffcb8 0x6d
0x11fffffcb9 0x3b
0x11fffffcb0 0x00
0x11fffffcb1 0x00
```
Representing Integers

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

Decimal: 15213
Binary: 0011 1011 0110 1101
Hex: 3 B 6 D

Two’s complement representation
(Covered next lecture)
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 1100 1010 0000

Sun Address
Hex:  E  F  F  F  F  B  2  C
Binary: 1110 1111 1111 1111 1111 1011 0010 1100

Linux Address
Hex:  B  F  F  F  F  8  D  4
Binary: 1011 1111 1111 1111 1111 1000 1101 0100

Different compilers & machines assign different locations to objects
Representing Floats

Float \( F = 15213.0; \)

IEEE Single Precision Floating Point Representation

<table>
<thead>
<tr>
<th>Hex:</th>
<th>4</th>
<th>6</th>
<th>6</th>
<th>D</th>
<th>B</th>
<th>4</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15213:</td>
<td>1110</td>
<td>1101</td>
<td>1011</td>
<td>01</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Linux/Alpha \( F \)

Sun \( F \)

00
B4
6D
46

Not same as integer representation, but consistent across machines
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!

char S[6] = "15213";

Linux/Alpha s  Sun s

| 31 | 31 |
| 35 | 35 |
| 32 | 32 |
| 31 | 31 |
| 33 | 33 |
| 00 | 00 |
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 \( B=1 \)
- \( \begin{array}{c|cc}
    & 0 & 1 \\
    \hline
    0 & 0 & 0 \\
    1 & 0 & 1 \\
  \end{array} \)

Or

- \( A \mid B = 1 \) when either \( A=1 \) or \( B=1 \)
- \( \begin{array}{c|cc}
    & 0 & 1 \\
    \hline
    0 & 0 & 1 \\
    1 & 1 & 1 \\
  \end{array} \)

Not

- \( \sim A = 1 \) when \( A=0 \)
- \( \begin{array}{c|c}
    \sim & \\
    \hline
    0 & 1 \\
    1 & 0 \\
  \end{array} \)

Exclusive-Or (Xor)

- \( A \oplus B = 1 \) when either \( A=1 \) or \( B=1 \), but not both
- \( \begin{array}{c|cc}
    & 0 & 1 \\
    \hline
    0 & 0 & 1 \\
    1 & 1 & 0 \\
  \end{array} \)
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:
\[ A \& \neg B \mid \neg A \& B \]

\[ = A^\land B \]
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**  
**Integer Ring**

- **Commutativity**
  - $A | B = B | A$
  - $A & B = B & A$
  - $A + B = B + A$
  - $A * B = B * A$

- **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**
  - $A | 0 = A$
  - $A & 1 = A$
  - $A + 0 = A$
  - $A * 1 = A$

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

- **Cancellation of negation**
  - $\sim (\sim A) = A$
  - $- (\sim A) = A$
Ring ≠ Boolean Algebra

### Boolean Algebra

- **Sum distributes over product**
  \[ A \lor (B \land C) = (A \lor B) \land (A \lor C) \]
- **Idempotency**
  \[ A \lor A = A \]
  \[ A \land A = A \]
  - “A is true” or “A is true” = “A is true”
- **Absorption**
  \[ A \lor (A \land B) = A \]
  \[ A \land (A \lor B) = A \]
- **Laws of Complements**
  \[ A \lor \neg A = 1 \]
  \[ A + -A \neq 1 \]
  - “A is true” or “A is false”

### Integer Ring

- **Additive inverse**
  \[ A \lor -A \neq 0 \]
  \[ A + -A = 0 \]
Properties of & and ^

Boolean Ring

• \(\langle \{0,1\}, ^\wedge, &, I, 0, 1 \rangle\)
• Identical to integers mod 2
• \(I\) is identity operation: \(I(A) = A\)
  \[A \wedge A = 0\]

Property

<table>
<thead>
<tr>
<th>Property</th>
<th>Boolean Ring</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commutative sum</td>
<td>(A \wedge B = B \wedge A)</td>
</tr>
<tr>
<td>Commutative product</td>
<td>(A &amp; B = B &amp; A)</td>
</tr>
<tr>
<td>Associative sum</td>
<td>((A \wedge B) \wedge C = A \wedge (B \wedge C))</td>
</tr>
<tr>
<td>Associative product</td>
<td>((A &amp; B) &amp; C = A &amp; (B &amp; C))</td>
</tr>
<tr>
<td>Prod. over sum</td>
<td>(A &amp; (B \wedge C) = (A &amp; B) \wedge (B &amp; C))</td>
</tr>
<tr>
<td>0 is sum identity</td>
<td>(A \wedge 0 = A)</td>
</tr>
<tr>
<td>1 is prod. identity</td>
<td>(A &amp; 1 = A)</td>
</tr>
<tr>
<td>0 is product annihilator</td>
<td>(A &amp; 0 = 0)</td>
</tr>
<tr>
<td>Additive inverse</td>
<td>(A \wedge A = 0)</td>
</tr>
</tbody>
</table>
Relations Between Operations

DeMorgan’s Laws

• Express & in terms of |, and vice-versa

  \[ A \& B = \sim(\sim A \mid \sim B) \]

  » A and B are true if and only if neither A nor B is false

  \[ A \mid B = \sim(\sim A \& \sim B) \]

  » A or B are true if and only if A and B are not both false

Exclusive-Or using Inclusive Or

| A \& B = (\sim A \& B) \mid (A \& \sim B) |

  » Exactly one of A and B is true

| A \& B = (A \mid B) \& \sim(A \& B) |

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

Operate on Bit Vectors
- Operations applied bitwise

\[
\begin{array}{c}
01101001 \\
& 01010101 \\
\hline
01101010
\end{array}
\]

\[
\begin{array}{c}
01101001 \\
| 01010101 \\
\hline
01111011
\end{array}
\]

\[
\begin{array}{c}
01101001 \\
^ 01010101 \\
\hline
00111000
\end{array}
\]

\[
\begin{array}{c}
01101001 \\
\sim 01010101 \\
\hline
10101010
\end{array}
\]

Representation of Sets
- Width \( w \) bit vector represents subsets of \( \{0, \ldots, w-1\} \)
- \( a_j = 1 \) if \( j \in A \)
  - \( -01101001 \)  \{ \}
  - \( -01010101 \)  \{ \}
- & Intersection 01000001
- | Union 01111101
- ^ Symmetric difference 00111100
- ~ Complement 10101010
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)

- \(~0x41\) --> \(\sim01000011_2\) --> \(10111100_2\)
- \(~0x00\) --> \(\sim00000000_2\) --> \(11111111_2\)
- \(0x69 \& 0x55\) --> \(01101001_2 \& 01010101_2\) --> \(01000001_2\)
- \(0x69 \mid 0x55\) --> \(01101001_2 \mid 01010101_2\) --> \(01111101_2\)
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
- !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

<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 \oplus B )</td>
<td>B</td>
</tr>
<tr>
<td>2</td>
<td>( A \oplus B )</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>( A \oplus B )</td>
<td>B</td>
</tr>
<tr>
<td>End</td>
<td>( A \oplus B )</td>
<td>B</td>
</tr>
</tbody>
</table>