Bits and Bytes
CS 213
Aug. 27, 1998

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 $1.1101101101101_2 \times 2^{13}$

Electronic Implementation

- Easy to store with bistable elements
- Reliably transmitted on noisy and inaccurate wires
- Straightforward implementation of arithmetic functions
Anatomy of an SRAM Cell

Inverter:
- High input --> Low Output
- Low input --> High Output

(bit line b)

(bit line b')

(word line)
SRAM Cell Principle

Inverter Amplifies
- Negative gain
- Slope < –1 in middle
- Saturates at ends

Inverter Pair Amplifies
- Positive gain
- Slope > 1 in middle
- Saturates at ends

Vin \[\rightarrow\] V1 \[\rightarrow\] Vin

V2 \[\rightarrow\] V1 \[\rightarrow\] V2
Bistable Element

Stability

• Require $V_{in} = V_2$
• Stable at endpoints
  – recover from perturbation
• Metastable in middle
  – Fall out when perturbed

Ball on Ramp Analogy
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 \texttt{0xFA1D37B}
    » Or \texttt{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

• Our Alphas are 64 bits (8 bytes)
  – Potentially address $1.8 \times 10^{19}$ bytes
  – Although current machines cannot do this
    » Limit is 4Terabytes

• 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 (Sun) or 8 (Alpha)
Data Representations

Sizes of C Objects (in Bytes)

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Alpha</th>
<th>Sun, PC, Mac, etc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long int</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>char *</td>
<td>8</td>
<td>4</td>
</tr>
</tbody>
</table>

» Or any other pointer

Byte Ordering

- 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
Examining Data Representations

Code to Print Byte Representation of Data

- Cast 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%x\n", 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;
0x11fffffb8  0x6d
0x11fffffc9  0x3b
0x11fffffcba  0x0
0x11ffffffcbb  0x0
```
Representing Integers

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

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>15213</td>
<td>0011 1011 0110 1101</td>
<td>3 B 6 D</td>
</tr>
</tbody>
</table>

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>1</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>C</th>
<th>A</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>0001 1111 1111 1111 1111 1111 1100 1010 0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Sun Address

<table>
<thead>
<tr>
<th>Hex</th>
<th>E</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>B</th>
<th>2</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>1110 1111 1111 1111 1111 1111 1011 0010 1100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Different compilers & machines assign different locations to objects*
Representing Floats

Float $F = 15213.0$;

IEEE Single Precision Floating Point Representation

Hex: 4 6 6 D B 4 0 0 0
Binary: 0100 0110 0110 1101 1011 0100 0000 0000
15213: 1110 1101 1011 01

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";
```

```
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

```
int sum(int x, int y)
{
    return x+y;
}
```

<table>
<thead>
<tr>
<th>Alpha sum</th>
<th>Sun sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>81</td>
</tr>
<tr>
<td>00</td>
<td>C3</td>
</tr>
<tr>
<td>30</td>
<td>E0</td>
</tr>
<tr>
<td>42</td>
<td>08</td>
</tr>
<tr>
<td>01</td>
<td>90</td>
</tr>
<tr>
<td>80</td>
<td>02</td>
</tr>
<tr>
<td>FA</td>
<td>00</td>
</tr>
<tr>
<td>6B</td>
<td>09</td>
</tr>
</tbody>
</table>

- For this example, both use two 4-byte instructions
  - Use differing numbers of instructions in other cases

*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

<table>
<thead>
<tr>
<th>&amp;</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

<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

<table>
<thead>
<tr>
<th>~</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

## Exclusive-Or (Xor)

- $A\oplus B = 1$ when either $A=1$ or $B=1$, but not both
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
\[ A \& \sim B \mid \sim A \& B \]
\[ = A^{\sim}B \]

Diagram:
```
A

\sim A

\sim B

B

```

Connection when
\[ A \& \sim B \mid \sim A \& B \]
\[ = A^{\sim}B \]
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

<table>
<thead>
<tr>
<th>Boolean Algebra</th>
<th>Integer Ring</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Commutativity</strong></td>
<td></td>
</tr>
<tr>
<td>$A</td>
<td>B \quad = \quad B</td>
</tr>
<tr>
<td>$A &amp; B \quad = \quad B &amp; A$</td>
<td>$A * B \quad = \quad B * A$</td>
</tr>
<tr>
<td><strong>Associativity</strong></td>
<td></td>
</tr>
<tr>
<td>$(A</td>
<td>B)</td>
</tr>
<tr>
<td>$(A &amp; B) &amp; C \quad = \quad A &amp; (B &amp; C)$</td>
<td>$(A * B) * C \quad = \quad A * (B * C)$</td>
</tr>
<tr>
<td><strong>Product distributes over sum</strong></td>
<td></td>
</tr>
<tr>
<td>$A &amp; (B</td>
<td>C) \quad = \quad (A &amp; B)</td>
</tr>
<tr>
<td><strong>Sum and product identities</strong></td>
<td></td>
</tr>
<tr>
<td>$A</td>
<td>0 \quad = \quad A$</td>
</tr>
<tr>
<td>$A &amp; 1 \quad = \quad A$</td>
<td>$A * 1 \quad = \quad A$</td>
</tr>
<tr>
<td><strong>Zero is product annihilator</strong></td>
<td></td>
</tr>
<tr>
<td>$A &amp; 0 \quad = \quad 0$</td>
<td>$A * 0 \quad = \quad 0$</td>
</tr>
<tr>
<td><strong>Cancellation of negation</strong></td>
<td></td>
</tr>
<tr>
<td>$\sim (\sim A) \quad = \quad A$</td>
<td>$\sim (\sim A) \quad = \quad A$</td>
</tr>
</tbody>
</table>
## Ring $\neq$ Boolean Algebra

<table>
<thead>
<tr>
<th>Boolean Algebra</th>
<th>Integer Ring</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Boolean:</strong> <em>Sum distributes over product</em></td>
<td></td>
</tr>
<tr>
<td>$A \mid (B &amp; C) = (A \mid B) &amp; (A \mid C)$</td>
<td>$A + (B \times C) \neq (A + B) \times (B + C)$</td>
</tr>
<tr>
<td><strong>Boolean:</strong> <em>Idempotency</em></td>
<td></td>
</tr>
<tr>
<td>$A \mid A = A$</td>
<td>$A + A \neq A$</td>
</tr>
<tr>
<td>“$A$ is true” or “$A$ is true” = “$A$ is true”</td>
<td></td>
</tr>
<tr>
<td>$A &amp; A = A$</td>
<td>$A \times A \neq A$</td>
</tr>
<tr>
<td><strong>Boolean:</strong> <em>Absorption</em></td>
<td></td>
</tr>
<tr>
<td>$A \mid (A &amp; B) = A$</td>
<td>$A + (A \times B) \neq A$</td>
</tr>
<tr>
<td>“$A$ is true” or “$A$ is true and $B$ is true” = “$A$ is true”</td>
<td></td>
</tr>
<tr>
<td>$A &amp; (A \mid B) = A$</td>
<td>$A \times (A + B) \neq A$</td>
</tr>
<tr>
<td><strong>Boolean:</strong> <em>Laws of Complements</em></td>
<td></td>
</tr>
<tr>
<td>$A \mid \sim A = 1$</td>
<td>$A + \sim A \neq 1$</td>
</tr>
<tr>
<td>“$A$ is true” or “$A$ is false”</td>
<td></td>
</tr>
<tr>
<td><strong>Ring:</strong> <em>Every element has additive inverse</em></td>
<td></td>
</tr>
<tr>
<td>$A \mid \sim A \neq 0$</td>
<td>$A + \sim A = 0$</td>
</tr>
</tbody>
</table>
Properties of & and ^

Boolean Ring
- $\langle\{0,1\}, ^, &, I, 0, 1\rangle$
- Identical to integers mod 2
- $I$ is identity operation: $I(A) = A$
  \[-A \land A = 0\]

Property
- 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 $A \land (B \lor C) = (A \land B) \lor (B \land C)$
- 0 is sum identity $A \land 0 = A$
- 1 is prod. identity $A \lor 1 = A$
- 0 is product annihilator $A \land 0 = 0$
- Additive inverse $A \land A = 0$
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 neither A is false nor B is false

Exclusive-Or using Inclusive Or

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

» Exactly one of A and B is true

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

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

Operate on Bit Vectors

- Operations applied bitwise

\[
\begin{align*}
01101001 & \quad 01101001 & \quad 01101001 \\
\& 01010101 & \mid 01010101 & \oplus 01010101 & \sim 01010101 \\
01000001 & \quad 01111101 & \quad 00111100 & \quad 10101010
\end{align*}
\]

Representation of Sets

- Width \( w \) bit vector represents subsets of \( \{0, \ldots, w-1\} \)

- \( a_j = 1 \) if \( j \in A \)

- \( \& \) Intersection

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

- \( \mid \) Union

\[
\begin{align*}
01000001 & \quad \{0, 6\} \\
01111101 & \quad \{0, 2, 3, 4, 5, 6\}
\end{align*}
\]

- \( ^\oplus \) Symmetric difference

\[
\begin{align*}
00111100 & \quad \{2, 3, 4, 5\}
\end{align*}
\]

- \( ^\sim \) Complement

\[
\begin{align*}
10101010 & \quad \{1, 3, 5, 7\}
\end{align*}
\]
Bit-Level Operations in C

Operations &, |, ~, ^ Available in C

- Apply to any “integer” data type
  - long int, int, short, char
- View arguments as bit vectors
- Arguments applied bit-wise

Examples (Char data type)

- \(~0x41 \rightarrow 0xBE\)
  \(~01000001_2 \rightarrow 10111110_2\)

- \(~0x00 \rightarrow 0xFF\)
  \(~00000000_2 \rightarrow 11111111_2\)

- \(0x69 \& 0x55 \rightarrow 0x41\)
  \(01101001_2 \& 01010101_2 \rightarrow 01000001_2\)

- \(0x69 | 0x55 \rightarrow 0x7D\)
  \(01101001_2 | 01010101_2 \rightarrow 01111101_2\)
Contrast: Logic Operations in C

Contrast to Logical Operators

• &&, ||, !
  – View 0 as “False”
  – Anything nonzero as “True”
  – Always return 0 or 1

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></th>
</tr>
</thead>
<tbody>
<tr>
<td>01100010</td>
<td></td>
</tr>
<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></th>
</tr>
</thead>
<tbody>
<tr>
<td>10100010</td>
<td></td>
</tr>
<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>
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>(A(\oplus)B)(\oplus)B = A(\oplus)(B(\oplus)B) = A(\oplus)0 = A</td>
</tr>
<tr>
<td>3</td>
<td>(A(\oplus)B)(\oplus)A = (B(\oplus)A)(\oplus)A = B(\oplus)(A(\oplus)A) = B(\oplus)0 = B</td>
<td>A</td>
</tr>
<tr>
<td>End</td>
<td>B</td>
<td>A</td>
</tr>
</tbody>
</table>