CS 33
Data Representation, Part 1
Number Representation

- Hindu-Arabic numerals
  - developed by Hindus starting in 5th century
    » positional notation
    » symbol for 0
  - adopted and modified somewhat later by Arabs
    » known by them as “Rakam Al-Hind” (Hindu numeral system)
  - 1999 rather than MCMXCIX
    » (try doing long division with Roman numerals!)
Which Base?

• 1999
  – base 10
    » 9·10^0+9·10^1+9·10^2+1·10^3
  – base 2
    » 11111001111
      • 1·2^0+1·2^1+1·2^2+1·2^3+0·2^4+0·2^5+1·2^6+1·2^7+1·2^8+1·2^9+1·2^10
  – base 8
    » 3717
      • 7·8^0+1·8^1+7·8^2+3·8^3
        » why are we interested?
  – base 16
    » 7CF
      • 15·16^0+12·16^1+7·16^2
        » why are we interested?
Words ...

12-bit computer word

```
0111111001111
```

3 7 1 7

16-bit computer word

```
000000111111001111
```

0 7 C F
Algorithm ...

```c
void baseX(unsigned int num, unsigned int base) {
    char digits[] = {'0', '1', '2', '3', '4', '5', '6', ... };
    char buf[8*sizeof(unsigned int)+1];
    int i;

    for (i = sizeof(buf) - 2; i >= 0; i--) {
        buf[i] = digits[num%base];
        num /= base;
        if (num == 0)
            break;
    }

    buf[sizeof(buf) - 1] = '\0';
    printf("%s\n", &buf[i]);
}
```
Or …

```
$ bc
obase=16
1999
7CF
$
```
Quiz 1

• What’s the decimal (base 10) equivalent of $23_{16}$?
  a) 19
  b) 33
  c) 35
  d) 37
## 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`
      - `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>
## 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>&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
- \( A | B = 1 \) 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>0</td>
<td>1</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>
General Boolean Algebras

• Operate on bit vectors
  – operations applied bitwise

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

• All of the properties of boolean algebra apply
Example: Representing & Manipulating Sets

• Representation
  – width-w bit vector represents subsets of \{0, ..., w–1\}
  – \( a_j = 1 \) iff \( j \in A \)

  \[
  \begin{array}{ll}
  01101001 & \{0, 3, 5, 6\} \\
  76543210 & \\
  01010101 & \{0, 2, 4, 6\} \\
  76543210 & 
  \end{array}
  \]

• 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 datatype)
  ~0x41 → 0xBE
  ~01000001₂ → 10111110₂
  ~0x00 → 0xFF
  ~00000000₂ → 11111111₂
  0x69 & 0x55 → 0x41
    01101001₂ & 01010101₂ → 01000001₂
  0x69 | 0x55 → 0x7D
    01101001₂ | 01010101₂ → 01111101₂
Contrast: Logic Operations in C

- **Contrast to Logical Operators**
  - `&&`, `||`, `!`
    - view 0 as “false”
    - anything nonzero as “true”
    - always return 0 or 1
    - early termination/short-circuited execution

- **Examples (char datatype)**
  - `!0x41` → 0x00
  - `!0x00` → 0x01
  - `!!0x41` → 0x01
  - `0x69 && 0x55` → 0x01
  - `0x69 || 0x55` → 0x01
  - `p && *p` (avoids null pointer access)
Contrast: Logic Operations in C

• Contrast to Logical Operators
  - &&, ||, !
    » view 0 as “false”
    » anything nonzero as “true”
    » always return 0 or 1
    » early termination/short-circuited execution

• Examples (char datatype)
  !0x41 → 0x00
  !0x00 → 0x01
  !!0x41 → 0x01

0x69 && 0x55 → 0x01
0x69 || 0x55 → 0x01
p && *p (avoids null pointer access)

Watch out for && vs. & (and || vs. |)… One of the more common oopsies in C programming
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 left
- **Undefined Behavior**
  - shift amount < 0 or ≥ word size
Signed Integers

- Sign-magnitude

\[
\begin{array}{cccccc}
  b_{w-1} & b_{w-2} & b_{w-3} & \cdots & b_2 & b_1 & b_0 \\
  \text{sign} & \text{magnitude} \\
\end{array}
\]

\[
\text{value} = (-1)^{b_{w-1}} \cdot \sum_{i=0}^{w-2} b_i \cdot 2^i
\]

- two representations of zero!
  - computer must have two sets of instructions
    - one for signed arithmetic, one for unsigned
Signed Integers

• Ones’ complement
  – negate a number by forming its bitwise complement
    » e.g., (-1)·01101011 = 10010100

\[
\text{value} = -b_{w-1}(2^{w-1} - 1) + \sum_{i=0}^{w-2} b_i \cdot 2^i
\]

\[
= \sum_{i=0}^{w-2} b_i \cdot 2^i \quad \text{if } b_{w-1} = 0
\]

\[
= \sum_{i=0}^{w-2} (b_i - 1) \cdot 2^i \quad \text{if } b_{w-1} = 1
\]

two zeroes!
Signed Integers

- Two’s complement
  \[ b_{w-1} = 0 \implies \text{non-negative number} \]
  \[ \text{value} = \sum_{i=0}^{w-2} b_i \cdot 2^i \]
  \[ b_{w-1} = 1 \implies \text{negative number} \]
  \[ \text{value} = (-1) \cdot 2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i \]
Signed Integers

• Negating two’s complement

\[ \text{value} = -b_{w-1}2^{w-1} + \sum_{i=0}^{w-2} b_i 2^i \]

– how to compute –\text{value}?

\((\neg \text{value})+1\)
Signed Integers

- Negating two’s complement (continued)

\[\text{value} + (\overline{\text{value}} + 1)\]

\[= (\text{value} + \overline{\text{value}}) + 1\]

\[= (2^w - 1) + 1\]

\[= 2^w\]

\[= \begin{array}{cccccc}
1 & 0 & 0 & 0 & \ldots & 0 & 0 & 0
\end{array}\]
Quiz 2

• We have a computer with 4-bit words that uses two’s complement to represent negative numbers. What is the result of subtracting 0010 (2) from 0001 (1)?
  a) 0111
  b) 1001
  c) 1110
  d) 1111