Many of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook “Computer Systems: A Programmer’s Perspective.” 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O’Hallaron in Fall 2010. These slides are indicated “Supplied by CMU” in the notes section of the slides.
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!)
Base 2 is known as “binary” notation.
Base 8 is known as “octal” notation.
Base 10 is known as “decimal” notation.
Base 16 is known as “hexadecimal” notation. Note that “hexa” is derived from the Greek language and “decimal” is derived from the Latin language. Many people feel you shouldn’t mix languages when you invent words, but IBM, who coined the term “hexadecimal” in the 1960s, didn’t think their corporate image could withstand “sexadecimal”.

Which Base?

• 1999  
  – base 10  
    » 9·10²+9·10¹+9·10⁰+1·10³  
  – base 2  
    » 11111001111  
      » 1·2⁹+1·2⁸+1·2⁷+1·2⁶+0·2⁵+0·2⁴+0·2³+1·2²+1·2¹+1·2⁰  
  – base 8  
    » 3717  
      » 7·8²+1·8¹+7·8⁰+3·8³  
      » why are we interested?  
  – base 16  
    » 7CF  
      » 15·16²+12·16¹+7·16⁰  
      » why are we interested?
Note that a byte consists of two hexadecimal digits, which are sometimes known as “nibbles”. A 32-bit computer word would then have eight nibbles; a 64-bit computer word would have sixteen nibbles.
This routine prints the base base representation of num. The “%” operator yields the remainder. E.g., “10%3” evaluates to 1: the remainder after dividing 10 by 3. (Note that the “…” is not heretofore unexplained C syntax, but is shorthand for “fill this in to the extent needed.”)
“bc” (it stands for basic calculator, or perhaps better calculator) is a standard Unix command that handles arbitrary-precision arithmetic. Among its features is the ability to specify which base to use for input and output of numbers. The default base for both input and output is ten. Setting `obase` to 16 sets the base for output to 16. Similarly, one can change the base for input numbers by setting `ibase`. 

```bash
$ 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 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
      • 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

<table>
<thead>
<tr>
<th>And</th>
<th>Or</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>1 0 1</td>
</tr>
<tr>
<td>0 0</td>
<td>0 0 1</td>
</tr>
<tr>
<td>1 0 1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Not</th>
<th>Exclusive-Or (Xor)</th>
</tr>
</thead>
<tbody>
<tr>
<td>~A</td>
<td>A^B</td>
</tr>
<tr>
<td>1</td>
<td>0 1</td>
</tr>
<tr>
<td>0</td>
<td>0 0 1</td>
</tr>
<tr>
<td>1 0 1</td>
<td></td>
</tr>
</tbody>
</table>

Supplied by CMU.
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>01010101</td>
<td>^ 01010101</td>
</tr>
<tr>
<td>01000001</td>
<td>01111101</td>
<td>00111100</td>
</tr>
</tbody>
</table>

- 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 \( j \in A \)

  \[
  \begin{align*}
  01101001 & \quad \{0, 3, 5, 6\} \\
  76543210 & \\
  01010101 & \quad \{0, 2, 4, 6\} \\
  76543210 & 
  \end{align*}
  \]

- **Operations**
    - \& intersection
    - | union
    - ^ symmetric difference
    - ~ complement

<table>
<thead>
<tr>
<th></th>
<th>01000001</th>
<th>{0, 6}</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp; \text{intersection}</td>
<td>01111101</td>
<td>{0, 2, 3, 4, 5, 6}</td>
</tr>
<tr>
<td>\text{union}</td>
<td>00111100</td>
<td>{2, 3, 4, 5}</td>
</tr>
<tr>
<td>^ \text{symmetric difference}</td>
<td>10101010</td>
<td>{1, 3, 5, 7}</td>
</tr>
<tr>
<td>~ \text{complement}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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`
  - `~01000001_2 → 10111110_2`
  - `~0x00 → 0xFF`
  - `~00000000_2 → 11111111_2`
  - `0x69 & 0x55 → 0x41`
  - `01101001_2 & 01010101_2 → 01000012`
  - `0x69 | 0x55 → 0x7D`
  - `01101001_2 | 01010110_2 → 01111110_2`

Supplied by CMU.
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)**
  
<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x41</td>
<td>0x00</td>
</tr>
<tr>
<td>0x00</td>
<td>0x01</td>
</tr>
<tr>
<td>0x41</td>
<td>0x01</td>
</tr>
<tr>
<td>0x69 &amp;&amp; 0x55</td>
<td>0x01</td>
</tr>
<tr>
<td>0x69</td>
<td></td>
</tr>
<tr>
<td>p &amp;&amp; *p</td>
<td>(avoids null pointer access)</td>
</tr>
</tbody>
</table>
Contrast: Logic Operations in C

- Contrast to Logical Operators
  - &&, ||, !
  -erview as "false"

Watch out for && vs. & (and || vs. |)...
One of the more common oopsies in C programming

```
0x41 → 0x00
!0x00 → 0x01
!10x41 → 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 left
- **Undefined Behavior**
  - shift amount < 0 or ≥ word size

<table>
<thead>
<tr>
<th>Argument $x$</th>
<th>01100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&lt; 3</td>
<td>00010000</td>
</tr>
<tr>
<td>Log. &gt;&gt; 2</td>
<td>00110000</td>
</tr>
<tr>
<td>Arith. &gt;&gt; 2</td>
<td>00011000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument $x$</th>
<th>10100010</th>
</tr>
</thead>
<tbody>
<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>

Supplied by CMU.

Why we need both logical and arithmetic shifts should be clear by the end of the next lecture.
If a computer word is to be interpreted as an unsigned integer, we can do so as shown in the slide, where $w$ is the number of bits in the word.
Signed Integers

- Sign-magnitude

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

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

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

We might also want to interpret the contents of a computer word as a signed integer. There are a few options for how to do this. One straightforward approach is shown in the slide, where we use the high-order (leftmost) bit as the “sign bit”: 0 means positive and 1 means negative. However, this has the somewhat weird result that there are two representations of zero. This further means that the computer would have to have two implementations of arithmetic instructions: one for signed arithmetic, the other for unsigned arithmetic.
Note that the most-significant bit serves as the sign bit. But, as with sign-magnitude, the computer would need two sets of instructions: one for signed arithmetic and one for unsigned.
Signed Integers

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

There’s only one zero!

Two’s complement is used on pretty much all of today’s computers to represent signed integers.

Note that the high-order (most-significant) bit represents \(-2^{w-1}\). All the other bits represent positive numbers.
To negate a two's-complement number, simply complement each of its bits, then add one to the result. We show why this works in the next slide.
If we add to the two’s complement representation of a w-bit number the result of adding one to its bitwise complement, we get a w+1-bit number whose low-order w bits are zeroes and whose high-order bit is one. However, since we’re constrained to only w bits, the result is a w-bit value of all zeroes, plus an overflow. If we ignore the overflow, the result is zero.
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