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”.
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 \textit{base} representation of \textit{num}. The \texttt{"\%"} operator yields the remainder. E.g., \texttt{"10\%3"} evaluates to 1: the remainder after dividing 10 by 3. (Note that the \texttt{"\ldots"} 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
7GF
$
```
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

Supplied by CMU.
Boolean Algebra

- Developed by George Boole in 19th Century
  - algebraic representation of logic
    » encode “true” as 1 and “false” as 0

And
- \( \text{A} \& \text{B} = 1 \) when both \( A=1 \) and \( B=1 \)

<table>
<thead>
<tr>
<th></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
- \( \text{A} \mid \text{B} = 1 \) 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
- \( \sim \text{A} = 1 \) 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)
- \( \text{A} \land \text{B} = 1 \) 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

  \[
  \begin{array}{cccc}
  01101001 & 01101001 & 01101001 \\
  \& 01010101 & \mid 01010101 & ^{01010101} & \sim 01010101 \\
  01000001 & 01111101 & 00111100 & 10101010
  \end{array}
  \]

- **All of the properties of boolean algebra apply**

Supplied by CMU.
Example: Representing & Manipulating Sets

• Representation
  – width-\(w\) bit vector represents subsets of \(\{0, \ldots, w-1\}\)
  – \(a_j = 1\) 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\} |

Supplied by CMU.
Bit-Level Operations in C

• Operations & \(\land\), | \(\lor\), ~ \(\neg\), ^ \(^{\land}\) 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 → 1011110₂
  ~0x00 → 0xFF
  ~00000000₂ → 1111111₂
  0x69 & 0x55 → 0x41
  01101001₂ & 01010101₂ → 01000001₂
  0x69 | 0x55 → 0x7D
  01101001₂ | 01010101₂ → 01111101₂

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)
  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**
  - `&&`, `||`, `!`
  - `&&` and `||` are bitwise operators, while `&&` and `||` are logical operators.
  - `!` is used for negation in both cases.
  - `&&` and `||` are used in control flow, while `&&` and `||` are used in arithmetic expressions.

- **One of the more common oopsies in C programming**
  - `!0x41` → `0x00`
  - `!0x00` → `0x01`
  - `!!0x41` → `0x01`
  - `0x69 && 0x55` → `0x01`
  - `0x69 || 0x55` → `0x01`
  - `p && *p` (avoids null pointer access)

Watch out for `&&` vs. `&` (and `||` vs. `|`).

Supplied by CMU.
The distinction between logical and arithmetic shifts should be clear by the end of this lecture.

Supplied by CMU.
Signed Integers

• **Sign-magnitude**

<table>
<thead>
<tr>
<th>$b_{w-1}$</th>
<th>$b_{w-2}$</th>
<th>$b_{w-3}$</th>
<th>...</th>
<th>$b_2$</th>
<th>$b_1$</th>
<th>$b_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>sign</td>
<td>magnitude</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

• 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

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

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

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

Note that the most-significant bit serves as the sign bit. Note, as with sign-magnitude, the computer would need two sets of instructions: one for signed arithmetic and one for unsigned.
Note there’s only one zero!

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

- Negating two’s complement

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

- how to compute \(-value\)?
  \((-\text{value})+1\)
Signed Integers

- Negating two’s complement (continued)

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

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

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

\[ = 2^w \]

\[
\begin{array}{cccccccc}
1 & 0 & 0 & 0 & \ldots & 0 & 0 & 0
\end{array}
\]

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