Csci 2021 Class Web Pages

Departmental Class Web Page:
  - Public access without login
  - Course syllabus, schedule, lecture notes, other useful info
    http://www-users.cselabs.umn.edu/classes/Spring-2016/csci2021/
  - You should read course *syllabus* carefully for all course policies, personnel info, and course logistics.

Moodle 2.8 Web page:
  - Use your university x500 account to access
  - Class forums, assignment submission, exam info
    https://ay15.moodle.umn.edu/course/view.php?id=13068
Data Representation:
Bits, Bytes, and Integers

CSci 2021 - Machine Architecture and Organization

Professor Pen-Chung Yew

With sides from Randy Bryant and Dave O’Hallaron
Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, divide with shift
  - Summary
- Representations in memory, pointers, strings
Binary Representations

- Each bit is 0 or 1
- Why bits? Electronic Implementation
  - Easy to store with bistable elements, e.g. switches, transistors,...
  - Reliably transmitted on noisy wires
- By encoding/interpreting sets of bits in various ways
  - Computers determine what to do (instructions),... and represent and manipulate numbers, sets, strings, etc...

![Diagram showing voltage levels for binary representation]

- 0V to 1.1V
- 0.2V to 0.9V
Base-2 number representation

- **Decimal to Binary**
  - Represent $152_{10}$ as $10011000_2$
  - Represent $1.20_{10}$ as $1.0011[0011]..._2$ (binary repetend)
  - Represent $1.51 \times 10^2$ as $1.0010001_2 \times 2^7$

- **Binary to Decimal**
  - Represent $10110_2$ as $22_{10}$
  - Represent $101.101_2$ as $5.625_{10}$
  - Represent $1.1011 \times 2^3$ as $1.35 \times 10^1$
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 FA1D37BE₁₆ in C as
      - 0xFA1D37BE
      - 0xfa1d37be
      - 0xFa1D37bE

<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>
Example Data Representations

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Typical 32-bit</th>
<th>Typical 64-bit</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<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>int</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>4</td>
<td>8</td>
<td>8</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>–</td>
<td>–</td>
<td>10/16</td>
</tr>
<tr>
<td>pointer</td>
<td>4</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>
Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, divide with shift
  - Summary
- Representations in memory, pointers, strings
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 \( B=1 \)

\[
\begin{array}{c|cc}
A & 0 & 1 \\
\hline
0 & 0 & 0 \\
1 & 0 & 1 \\
\end{array}
\]

Not
- \( \sim A = 1 \) when \( A=0 \)

\[
\begin{array}{c|c}
A & \sim \\
\hline
0 & 1 \\
1 & 0 \\
\end{array}
\]

Or
- \( A \mid B = 1 \) when either \( A=1 \) or \( B=1 \)

\[
\begin{array}{c|cc}
A & 0 & 1 \\
\hline
0 & 0 & 1 \\
1 & 1 & 1 \\
\end{array}
\]

Exclusive-Or (Xor)
- \( A^{\wedge}B = 1 \) when either \( A=1 \) or \( B=1 \), but not both

\[
\begin{array}{c|cc}
A & 0 & 1 \\
\hline
0 & 0 & 1 \\
1 & 1 & 0 \\
\end{array}
\]
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

\[ A \& \sim B \lor \sim A \& B = A^{\land}B \]
General Boolean Algebras

- Operate on Bit Vectors
  - Operations applied *bitwise*

```
01101001 & 01010101 = 01000001
01101001 | 01010101 = 01111101
01101001 ^ 01010101 = 00111100
01101001 ~ 01010101 = 10101010
```

- All of the Properties of Boolean Algebra Apply
Representing & Manipulating Sets

- **Representation**
  - Width \( w \) bit vector represents subsets of \( \{0, \ldots, w-1\} \)
  - \( a_j = 1 \) if \( j \in A \)
    - \[01101001\] \( A = \{ 0, 3, 5, 6 \} \)
    - \[76543210\]
    - \[01010101\] \( A = \{ 0, 2, 4, 6 \} \)
    - \[76543210\]

- **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 &,# V, ~, \^ Available in C**
  - Apply to any “integral” data type
    - long, int, short, char, unsigned
  - View arguments as bit vectors
  - Arguments applied bit-wise

- **Examples (char data type) (1 byte)**
  - \( \sim 0x41 = 0xBE \)
    - \( \sim 01000001_2 = 10111110_2 \)
  - \( \sim 0x00 \rightarrow 0xFF \)
    - \( \sim 00000000_2 = 11111111_2 \)
  - \( 0x69 \& 0x55 = 0x41 \)
    - \( 01101001_2 \& 01010101_2 = 01000001_2 \)
  - \( 0x69 \mid 0x55 = 0x7D \)
    - \( 01101001_2 \mid 01010101_2 = 01111101_2 \)
Contrast: Logic Operations in C

- Contrast to Logical Operators
  - &&, ||, !
    - View 0 as “False”
    - Anything nonzero as “True”
    - Always return 0 or 1

- Examples (char data type)
  - !0x41 = 0x00 (different from ~ operation)
  - !0x00 = 0x01
  - !!0x41 = 0x01
  - 0x69 && 0x55 = 0x01
  - 0x69 || 0x55 = 0x01
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

Watch out for && vs. & (and || vs. |)… one of the more common oopsies in C programming
### Shift Operations

- **Left Shift:** $x \ll y$
  - Shift bit-vector $x$ left $y$ positions
  - Throw away extra bits on left
    - Fill with 0’s on right
- **Right Shift:** $x \gg y$
  - 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 $\geq$ word size
Programming Assignment #1 *(Data Lab)* will be issued today, due in 2 weeks on *2/8/2016 11:55pm Monday*

- Download *Data Lab* from class Moodle page
- This lab asks you to manipulate bit strings using restricted instruction types and counts
- Each problem in the assignment has different degrees of difficulty, start with easier ones first.
- Feel free to use class *Forum* to ask questions and get information from TA or other students
- TA will discuss Data Lab in Thur recitation sessions.
Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, divide with shift
  - Summary
- Representations in memory, pointers, strings
- Summary
Encoding Unsigned and Signed Integers

**Unsigned (B2U)**

\[
B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i
\]

**Two’s Complement (B2T)**

\[
B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i
\]

- **Sign Bit**
  - For 2’s complement, most significant bit indicates sign
    - 0 for nonnegative
    - 1 for negative

```
short int x = 15213;
short int y = -15213;
```

- **In C, data type short is 2 bytes long**

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>15213</td>
<td>3B 6D</td>
<td>001111011 01101101</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
</tbody>
</table>
Encoding Example (Binary to Unsigned – B2U )

\[ x = 15213: \quad 00111011 \ 01101101 \]
\[ y = -15213: \quad 11000100 \ 10010011 \]

<table>
<thead>
<tr>
<th>Weight</th>
<th>15213</th>
<th>-15213</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>64</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>128</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>256</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>512</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1024</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2048</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4096</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8192</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>16384</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>-32768</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i \]
\[ B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i \]
## Binary Weight Table

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^1$</td>
<td>2</td>
</tr>
<tr>
<td>$2^2$</td>
<td>4</td>
</tr>
<tr>
<td>$2^3$</td>
<td>8</td>
</tr>
<tr>
<td>$2^4$</td>
<td>16</td>
</tr>
<tr>
<td>$2^5$</td>
<td>32</td>
</tr>
<tr>
<td>$2^6$</td>
<td>64</td>
</tr>
<tr>
<td>$2^7$</td>
<td>128</td>
</tr>
<tr>
<td>$2^8$</td>
<td>256</td>
</tr>
<tr>
<td>$2^9$</td>
<td>512</td>
</tr>
<tr>
<td>$2^{10}$</td>
<td>1024</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{11}$</td>
<td>$2^{1+10} = 2^1 \times 1024 = 2K$</td>
</tr>
<tr>
<td>$2^{12}$</td>
<td>$2^{2+10} = 2^2 \times 1024 = 4K$</td>
</tr>
<tr>
<td>$2^{13}$</td>
<td>$2^{3+10} = 2^3 \times 1024 = 8K$</td>
</tr>
<tr>
<td>$2^{14}$</td>
<td>$2^{4+10} = 2^4 \times 1024 = 16K$</td>
</tr>
<tr>
<td>$2^{15}$</td>
<td>$2^{5+10} = 2^5 \times 1024 = 32K$</td>
</tr>
<tr>
<td>$2^{16}$</td>
<td>$2^{6+10} = 2^6 \times 1024 = 64K$</td>
</tr>
<tr>
<td>$2^{17}$</td>
<td>$2^{7+10} = 2^7 \times 1024 = 128K$</td>
</tr>
<tr>
<td>$2^{18}$</td>
<td>$2^{8+10} = 2^8 \times 1024 = 256K$</td>
</tr>
<tr>
<td>$2^{19}$</td>
<td>$2^{9+10} = 2^9 \times 1024 = 512K$</td>
</tr>
<tr>
<td>$2^{20}$</td>
<td>$2^{10+10} = 2^{10} \times 1024 = 1M$</td>
</tr>
</tbody>
</table>
## Converting Unsigned to Binary (U2B)

Dividing the number repeatedly by 2 until the number becomes 0

\[
B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i
\]

Same algorithm to ANY number system

### Example: Converting 49 to Binary

<table>
<thead>
<tr>
<th>Divide by</th>
<th>Number</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 49 ?

\[
1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 49
\]
# Numeric Ranges

## Unssigned Values
- **$UMin$** = 0
  
  000...0

- **$UMax$** = $2^w - 1$
  
  111...1

## Two's Complement Values
- **$TMin$** = $-2^{w-1} = 100...0$

- **$TMax$** = $2^{w-1} - 1 = 011...1$

## Other Values
- Minus 1 = $-2^{w-1} + 1 = 111...1$

---

### Ranges for $w = 16$, i.e. 2 bytes (short)

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>$UMax$</td>
<td>65535</td>
<td>FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>$TMax$</td>
<td>32767</td>
<td>7F</td>
<td>01111111 11111111</td>
</tr>
<tr>
<td>$TMin$</td>
<td>-32768</td>
<td>80</td>
<td>10000000 00000000</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>

\[
B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i
\]

\[
B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i
\]
Values for Different Word Sizes

<table>
<thead>
<tr>
<th></th>
<th>w</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
</tr>
<tr>
<td>UMax</td>
<td>255</td>
</tr>
<tr>
<td>Tmax</td>
<td>127</td>
</tr>
<tr>
<td>Tmin</td>
<td>-128</td>
</tr>
</tbody>
</table>

**Observations**
- \(|TMin| = Tmax + 1\)
- **Asymmetric range**
- \(UMax = 2 \times Tmax + 1\)

**C Programming**
- `#include <limits.h>`
- Declares constants, e.g.,
  - `ULONG_MAX`
  - `LONG_MAX`
  - `LONG_MIN`
- **Values platform specific**
- Need to know -> to avoid **overflow**
**Unsigned & Signed Integer Values**

<table>
<thead>
<tr>
<th>$X$</th>
<th>$B2U(X)$</th>
<th>$B2T(X)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>-8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td>-7</td>
</tr>
<tr>
<td>1010</td>
<td>10</td>
<td>-6</td>
</tr>
<tr>
<td>1011</td>
<td>11</td>
<td>-5</td>
</tr>
<tr>
<td>1100</td>
<td>12</td>
<td>-4</td>
</tr>
<tr>
<td>1101</td>
<td>13</td>
<td>-3</td>
</tr>
<tr>
<td>1110</td>
<td>14</td>
<td>-2</td>
</tr>
<tr>
<td>1111</td>
<td>15</td>
<td>-1</td>
</tr>
</tbody>
</table>

- **Equivalence**
  - Same encodings for nonnegative/ negative integer values

- **Uniqueness**
  - Every bit pattern represents unique integer value
  - Each representable integer has unique bit encoding

- **Can Invert Mappings**
  - $U2B(x) = B2U^{-1}(x)$
    - Bit pattern for unsigned integer
  - $T2B(x) = B2T^{-1}(x)$
    - Bit pattern for two’s comp integer
Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, divide with shift
  - Summary
- Representations in memory, pointers, strings
Mapping Between Signed & Unsigned

Signed (Two’s Complement)  
\[ x \rightarrow \text{T2B} \rightarrow \text{T2U} \rightarrow \text{B2U} \rightarrow \text{Ununsigned} \]

Maintain Same Bit Pattern

Unsigned
\[ ux \rightarrow \text{U2B} \rightarrow \text{U2T} \rightarrow \text{B2T} \rightarrow x \]

Maintain Same Bit Pattern

- Mappings between unsigned and two’s complement numbers: keep bit representations and reinterpret
# Mapping Signed ↔ Unsigned

<table>
<thead>
<tr>
<th>Bits</th>
<th>Signed</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>-8</td>
<td>8</td>
</tr>
<tr>
<td>1001</td>
<td>-7</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>-6</td>
<td>10</td>
</tr>
<tr>
<td>1011</td>
<td>-5</td>
<td>11</td>
</tr>
<tr>
<td>1100</td>
<td>-4</td>
<td>12</td>
</tr>
<tr>
<td>1101</td>
<td>-3</td>
<td>13</td>
</tr>
<tr>
<td>1110</td>
<td>-2</td>
<td>14</td>
</tr>
<tr>
<td>1111</td>
<td>-1</td>
<td>15</td>
</tr>
</tbody>
</table>

- **T2U** (Signed to Unsigned): Converts signed numbers to their unsigned counterparts.
- **U2T** (Unsigned to Signed): Converts unsigned numbers to their signed counterparts.
### Mapping Signed ↔ Unsigned

<table>
<thead>
<tr>
<th>Bits</th>
<th>Signed</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>-8</td>
<td>8</td>
</tr>
<tr>
<td>1001</td>
<td>-7</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>-6</td>
<td>10</td>
</tr>
<tr>
<td>1011</td>
<td>-5</td>
<td>11</td>
</tr>
<tr>
<td>1100</td>
<td>-4</td>
<td>12</td>
</tr>
<tr>
<td>1101</td>
<td>-3</td>
<td>13</td>
</tr>
<tr>
<td>1110</td>
<td>-2</td>
<td>14</td>
</tr>
<tr>
<td>1111</td>
<td>-1</td>
<td>15</td>
</tr>
</tbody>
</table>
Relation between Signed & Unsigned

Two’s Complement

\[ u_x = \begin{cases} x & x \geq 0 \\ x + 2^w & x < 0 \end{cases} \]

Large negative weight becomes Large positive weight

Maintain Same Bit Pattern
Conversion Visualized

- **Signed → Unsigned**
  - Ordering Inversion
  - Negative → Big Positive
  - Warning: Can cause a lot of confusion and bugs in C !!!

![Diagram showing conversion visualization]

- **2’s Complement Range**
- ** Unsigned Range**
- **Unsigned Range**
- **Signed Range**
Announcement 1/27/2016

- This Thursday’s recitation sessions will cover both Data Lab assignment, and more C programming language features.
Review: Signed vs. Unsigned in C

- Constants
  - By default are considered to be signed integers
  - Unsigned if have “U” as suffix
    
    0U, 4294967259U

- Casting (i.e. conversion)
  - Explicit casting between signed & unsigned same as U2T and T2U
    
    int tx, ty;
    unsigned ux, uy;
    tx = (int) ux;
    uy = (unsigned) ty;

  - Implicit casting also occurs via assignments and procedure calls
    
    tx = ux;
    uy = ty;
Review: Casting Surprises

Expression Evaluation

- If there is a mix of unsigned and signed in single expression, **signed values implicitly cast to unsigned**
- Including comparison operations <, >, ==, <=, >=
- Examples for \( w = 32 \): \( T_{MIN} = -2,147,483,648 \), \( T_{MAX} = 2,147,483,647 \)

<table>
<thead>
<tr>
<th>Constant(_1)</th>
<th>Constant(_2)</th>
<th>Relation</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0U</td>
<td>==</td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>&lt;</td>
<td>signed</td>
</tr>
<tr>
<td>-1</td>
<td>0U</td>
<td>&gt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>-2147483647-1</td>
<td>&gt;</td>
<td>signed</td>
</tr>
<tr>
<td>2147483647U</td>
<td>-2147483647-1</td>
<td>&lt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>-2</td>
<td>&gt;</td>
<td>signed</td>
</tr>
<tr>
<td>(unsigned)-1</td>
<td>-2</td>
<td>&gt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>2147483648U</td>
<td>&lt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>(int) 2147483648U</td>
<td>&gt;</td>
<td>signed</td>
</tr>
</tbody>
</table>
Summary
Casting Signed $\leftrightarrow$ Unsigned: Basic Rules

- Bit pattern is maintained (i.e. kept the same)
- But reinterpreted
- Can have unexpected effects: adding or subtracting $2^w$
- Expression containing unsigned and signed int
  - \textit{int} is cast to \textit{unsigned}!!
Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations

**Integers**
- Representation: unsigned and signed
- Conversion, casting
- Expanding, truncating
- Addition, negation, multiplication, divide with shift
- Summary

- Representations in memory, pointers, strings
Sign Extension

**Task:**
- Given \( w \)-bit signed integer \( x \)
- Convert it to \( w+k \)-bit integer **with same value**

**Rule:**
- Make \( k \) copies of sign bit:
- \( X' = x_{w-1},..., x_{w-1}, x_{w-1}, x_{w-2},..., x_0 \)

---

**Diagram:**
- The diagram illustrates the sign extension process, showing \( X \) being extended to \( X' \) by making \( k \) copies of the most significant bit (MSB) and \( w \) copies of the least significant bit (LSB).
# Sign Extension Example

```c
short int x = 15213;
int ix = (int) x;
short int y = -15213;
int iy = (int) y;
```

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>15213</td>
<td>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>ix</td>
<td>15213</td>
<td>00 00 3B 6D</td>
<td>00000000 00000000 00111011 01101101</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>iy</td>
<td>-15213</td>
<td>FF FF C4 93</td>
<td>11111111 11111111 11000100 10010011</td>
</tr>
</tbody>
</table>

- Converting from smaller to larger integer data type
- C automatically performs sign extension
Summary:
Expanding, Truncating: Basic Rules

- **Expanding** (e.g., short to int)
  - **Unsigned**: zeros added
  - **Signed**: sign extension
  - Both yield expected result

- **Truncating** (e.g., unsigned to unsigned short)
  - Unsigned/signed: high-ordered bits are truncated
  - Result reinterpreted
  - Unsigned: mod operation
  - Signed: similar to mod
  - For small numbers yields expected behavior
Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, divide with shift
- Representations in memory, pointers, strings
- Summary
Negation: Complement & Increment

- **Claim:** Following Holds for 2’s Complement
  \[ \sim x + 1 = -x \]

- **A Shortcut to Calculate 2’s Complement**
  - **Observation:** \( \sim x + x = 1111...111 = -1 \)

\[
\begin{array}{c}
x \quad \begin{array}{ccccccc}
1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\
\end{array} \\
+ \quad \begin{array}{ccccccc}
0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\
\end{array} \\
\hline \\
\begin{array}{ccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\end{array}
\]

\[
B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i
\]
## Complement & Increment Examples

### $x = 15213$

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>15213</td>
<td>3B</td>
<td>6D 00111011 01101101</td>
</tr>
<tr>
<td>$\sim x$</td>
<td>-15214</td>
<td>C4</td>
<td>92 11000100 10010010</td>
</tr>
<tr>
<td>$\sim x + 1$</td>
<td>-15213</td>
<td>C4</td>
<td>93 11000100 10010011</td>
</tr>
<tr>
<td>$y$</td>
<td>-15213</td>
<td>C4</td>
<td>93 11000100 10010011</td>
</tr>
</tbody>
</table>

### $x = 0$

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>00000000 00000000</td>
</tr>
<tr>
<td>$\sim 0$</td>
<td>-1</td>
<td>FF</td>
<td>FF 11111111 11111111</td>
</tr>
<tr>
<td>$\sim 0 + 1$</td>
<td>0</td>
<td>00</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>

### Ranges for short (2 bytes)

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>65535</td>
<td>FF</td>
<td>FF 11111111 11111111</td>
</tr>
<tr>
<td>Tmax</td>
<td>32767</td>
<td>7F</td>
<td>FF 01111111 11111111</td>
</tr>
<tr>
<td>Tmin</td>
<td>-32768</td>
<td>80</td>
<td>00 10000000 00000000</td>
</tr>
<tr>
<td>$-1$</td>
<td>-1</td>
<td>FF</td>
<td>FF 11111111 11111111</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>

**What about $-T_{min}$?**

$T_{min} = - T_{max} - 1$
Unsigned Addition

Operands: $w$ bits

\[
\begin{array}{c}
\hline
u \\
\hline
+ v \\
\hline
u + v \\
\hline
\end{array}
\]

True Sum: $w+1$ bits

Discard Carry: $w$ bits

\[
\begin{array}{c}
\hline
UAdd_w(u, v) \\
\hline
\end{array}
\]

- **Standard Addition Function**
  - Ignores carry output (overflow is not signaled as errors in C language)

- **Implements Modular Arithmetic**

\[
s = UAdd_w(u, v) = (u + v) \mod 2^w
\]

\[
UAdd_w(u, v) = \begin{cases} 
    u + v & u + v < 2^w \\
    u + v - 2^w & u + v \geq 2^w 
\end{cases}
\]
Visualizing (Mathematical) Integer Addition

- **Integer Addition**
  - 4-bit integers $u, v$
  - Compute true sum $\text{Add}_4(u, v)$
  - Values increase linearly with $u$ and $v$
  - Forms planar surface
Visualizing Unsigned Addition

- Wraps Around
  - If true sum $\geq 2^w$

True Sum

$2^{w+1}$  
$2^w$  
$0$

Modular Sum

Overflow

$UAdd_4(u, v)$
Two’s Complement Addition

Operands: \( w \) bits

\[
\begin{array}{c}
\quad u \\
+ \quad v \\
\hline
u + v
\end{array}
\]

True Sum: \( w+1 \) bits

Discard Carry: \( w \) bits

\[
\text{TAdd}_w(u, v)
\]

- TAdd and UAdd have Identical Bit-Level Behavior
  - Signed vs. unsigned addition in C:
    ```
    int s, t, u, v;
    s = (int) ((unsigned) u + (unsigned) v);
    t = u + v
    ```
  - Will give \( s == t \)
TAdd Overflow

**Functionality**
- True sum requires $w+1$ bits
- Drop off Most Significant Bit (MSB)
- Treat remaining bits as 2’s comp. integer

<table>
<thead>
<tr>
<th>True Sum</th>
<th>TAdd Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0\ 111...1$</td>
<td>$011...1$</td>
</tr>
<tr>
<td>$0\ 100...0$</td>
<td>$000...0$</td>
</tr>
<tr>
<td>$0\ 000...0$</td>
<td>$000...0$</td>
</tr>
<tr>
<td>$1\ 011...1$</td>
<td>$100...0$</td>
</tr>
<tr>
<td>$1\ 000...0$</td>
<td>$100...0$</td>
</tr>
</tbody>
</table>

$2^{(w-1)}$
Visualizing 2’s Complement Addition

- **Values**
  - 4-bit two’s comp.
  - Range from -8 to +7

- **Wraps Around**
  - If sum $\geq 2^{w-1}$
    - Becomes negative
  - If sum $< -2^{w-1}$
    - Becomes positive
Characterizing TAdd

- **Functionality**
  - True sum requires $w+1$ bits
  - Drop off MSB
  - Treat remaining bits as 2’s comp. integer

$$TAdd_w(u, v) = \begin{cases} 
  u + v + 2^{w-1} & u + v < TMin_w \quad \text{(NegOver)} \\
  u + v & TMin_w \leq u + v \leq TMax_w \\
  u + v - 2^{w-1} & TMax_w < u + v \quad \text{(PosOver)}
\end{cases}$$
Unsigned Multiplication in C

Operands: $w$ bits

True Product: $2^w$ bits

Discard $w$ bits: $w$ bits

- **Standard Multiplication Function**
  - Ignores high order $w$ bits

- **Implements Modular Arithmetic**
  
  $UMult_w(u, v) = u \cdot v \mod 2^w$
Signed Multiplication in C

Operands: $w$ bits

True Product: $2w$ bits

Discard $w$ bits: $w$ bits

- **Standard Multiplication Function**
  - Ignores high order $w$ bits
  - Some of which are different for signed vs. unsigned multiplication
  - Lower bits are the same as unsigned multiplication
Multiplication

- **Computing Exact Product of** \( w \)-**bit numbers** \( x, y \)
  - Either *signed* or *unsigned*

- **Ranges**
  - **Unsigned**: Up to \( 2w \) bits
    - Result range: \( 0 \leq x \times y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1 \)
  - **Two’s complement min (negative)**: Up to \( 2w - 1 \) bits
    - Result range: \( x \times y \geq (-2^{w-1}) \times (2^w - 1) = -2^{2w-2} + 2^{w-1} \)
  - **Two’s complement max (positive)**: Up to \( 2w \) bits, but only for \( (TMin_w)^2 \)
    - Result range: \( x \times y \leq (-2^{w-1})^2 = 2^{2w-2} \)

- **Maintaining Exact Results**
  - Would need to keep expanding word size with each product computed
  - Done in software by “arbitrary precision” arithmetic packages
Power-of-2 Multiply with Shift

**Operation**
- $u << k$ gives $u \times 2^k$
- Both signed and unsigned

Operands: $w$ bits

True Product: $w+k$ bits

Discard $k$ bits: $w$ bits

**Examples**
- $u << 3 = u \times 8$
- $u << 5 - u << 3 = (u \times 32) - (u \times 8) = u \times 24$

Most machines shift and add faster than multiply
- Compiler generates this code automatically
Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations

**Integers**
- Representation: unsigned and signed
- Conversion, casting
- Expanding, truncating
- Addition, negation, multiplication, divide with shift
- Summary

- Representations in memory, pointers, strings
**Unsigned Power-of-2 Divide with Shift**

- **Quotient of Unsigned by Power of 2**
  - $u >> k$ gives $\lfloor u / 2^k \rfloor$ (“floor” operations)
  - Uses logical right shift

**Operands:**

<table>
<thead>
<tr>
<th>$u$</th>
<th>$l$</th>
<th>$2^k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\cdots$</td>
<td>$0$</td>
<td>$\cdots$</td>
</tr>
</tbody>
</table>

**Division:**

<table>
<thead>
<tr>
<th>$u / 2^k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lfloor u / 2^k \rfloor$</td>
</tr>
</tbody>
</table>

**Result:**

<table>
<thead>
<tr>
<th>$\lfloor u / 2^k \rfloor$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0$</td>
</tr>
</tbody>
</table>

**Table:**

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>15213</td>
<td>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>x &gt;&gt; 1</td>
<td>7606.5</td>
<td>1D B6</td>
<td>00011101 10110110</td>
</tr>
<tr>
<td>x &gt;&gt; 4</td>
<td>950.8125</td>
<td>03 B6</td>
<td>00000011 10110110</td>
</tr>
<tr>
<td>x &gt;&gt; 8</td>
<td>59.4257813</td>
<td>00 3B</td>
<td>00000000 00111011</td>
</tr>
</tbody>
</table>
Signed Power-of-2 Divide with Shift

- **Quotient of Signed by Power of 2**
  - $x \gg k$ gives $\lfloor x \div 2^k \rfloor$
  - Uses *arithmetic* right shift
  - Ok when $u \geq 0$, but rounds wrong direction when $u < 0$ *(should round toward 0)*

![Diagram showing quotient of signed by power of 2]

**Operands:**
- $x$
- $2^k$

**Division:**
- $x / 2^k$

**Result:** $\text{RoundDown}(x / 2^k)$

<table>
<thead>
<tr>
<th>$y$</th>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y$</td>
<td>-15213</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>$y \gg 1$</td>
<td>-7606.5</td>
<td>-7607</td>
<td>E2 49</td>
<td>11100010 01001001</td>
</tr>
<tr>
<td>$y \gg 4$</td>
<td>-950.8125</td>
<td>-951</td>
<td>FC 49</td>
<td>11111100 01001001</td>
</tr>
<tr>
<td>$y \gg 8$</td>
<td>-59.4257813</td>
<td>-60</td>
<td>FF C4</td>
<td>11111111 11000100</td>
</tr>
</tbody>
</table>
Correct Power-of-2 Divide

- **Quotient of Negative Number by Power of 2**
  - Want $\left\lfloor \frac{x}{2^k} \right\rfloor$ (Round toward 0, i.e. use “ceiling” operation)
  - Compute as $\left\lfloor \frac{x+2^k-1}{2^k} \right\rfloor$
    - In C: $(x + (1<<k)-1) \gg k$
  - Biase dividend toward 0

**Case 1: No rounding**

<table>
<thead>
<tr>
<th>Dividend:</th>
<th>$\left\lfloor \frac{u}{2^k} \right\rfloor$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u$</td>
<td>$1 \ldots 0 \ldots 0$</td>
</tr>
<tr>
<td>$+2^k - 1$</td>
<td>$0 \ldots 01 \ldots 11$</td>
</tr>
<tr>
<td>$+2^k - 1$</td>
<td>$1 \ldots 1 \ldots 11$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Divisor:</th>
<th>$\left\lfloor \frac{u}{2^k} \right\rfloor$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\frac{l}{2^k}$</td>
<td>$0 \ldots 010 \ldots 00$</td>
</tr>
<tr>
<td>$\left\lfloor \frac{u}{2^k} \right\rfloor$</td>
<td>$1 \ldots 111 \ldots 11$</td>
</tr>
</tbody>
</table>

**Biasing has no effect**
Correct Power-of-2 Divide (Cont.)

Case 2: Rounding

Dividend:

\[ \frac{x}{2^k - 1} \]

Divisor:

\[ \left\lfloor \frac{x}{2^k} \right\rfloor \]

Biasing adds 1 to final result
Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, divide with shift
  - Summary
- Representations in memory, pointers, strings
Arithmetic: Basic Rules

- **Addition:**
  - Normal addition followed by *truncate*, same operation on bit level
    - **Unsigned**: addition mod $2^w$
      - Mathematical addition & Possible subtraction of $2^w$
    - **Signed**: modified addition mod $2^w$ (result in proper range)
      - Mathematical addition & Possible addition or subtraction of $2^w$

- **Multiplication**: (product has $2w$ bits)
  - Normal multiplication followed by *truncate*, same operation on bit level
    - **Unsigned**: multiplication mod $2^w$ (result in proper range)
    - **Signed**: modified multiplication mod $2^w$ (result in proper range)
Arithmetic: Basic Rules

- **Left shift (Multiplication)**
  - Unsigned/signed: multiplication by $2^k$
  - Always **logical left** shift

- **Right shift (Division)**
  - Unsigned: **logical right** shift, div (division + round toward zero) by $2^k$
  - Signed: **arithmetic right** shift
    - Positive numbers: div (division + round toward zero) by $2^k$
    - Negative numbers: div (division + round away from zero) by $2^k$
      Use biasing to fix
Integer C Puzzle

- **Tmin** and **Tmax** make a good counter example in many cases.

1. \( x < 0 \) \( \Rightarrow \) \( ((x*2) < 0) \)  
   False: \( Tmin \)

2. \( ux >= 0 \)

3. \( x & 7 == 7 \) \( \Rightarrow \) \( (x<<30) < 0 \)
   True: \( x_1 = 1 \)

4. \( ux > -1 \)

5. \( x > y \) \( \Rightarrow \) \( -x < -y \)
   False: \( -1, \ Tmin \)

6. \( x * x >= 0 \)

7. \( x > 0 && y > 0 \) \( \Rightarrow \) \( x + y > 0 \)
   False: \( Tmax, \ Tmax \)

8. \( x >= 0 \) \( \Rightarrow \) \( -x <= 0 \)
   True: \( -TMax < 0 \)

9. \( x <= 0 \) \( \Rightarrow \) \( -x >= 0 \)
   False: \( Tmin \)

Initialization

```c
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```
Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, divide with shift
  - Summary
- Data representation in memory, pointers, character strings
Byte-Oriented Memory Organization

- System provides a *private address space* to each “process”
  - Think of a process as a program being executed
  - A program can clobber its own data, but not that of others

- Programs refer to data by address
  - Conceptually, envision it as a very large array of bytes
    - In reality, it’s not, but can think of it that way
  - An address is like an index into that array
    - and, a *pointer* variable stores an address
Machine Words

- Any given machine has a “Word Size”
  - Nominal size of integer-valued data
    - Including addresses
  - Many current machines use 32 bits (4 bytes) words
    - Limits addresses to $2^{32} = 2^2 \times 2^{30} = 4 \times 10^9 = 4$ GB
    - Becoming too small for memory-intensive applications
  - Most high-end systems use 64 bits (8 bytes) words
    - Potential address space $\approx 18 \times 10^{18}$ bytes (exabytes)
  - Machines support multiple data formats (data types)
    - Fractions or multiples of word size
    - Always integral number of bytes

$10^6 \approx 2^{20} \approx 1$ mega- 
$10^9 \approx 2^{30} \approx 1$ giga- 
$10^{12} \approx 2^{40} \approx 1$ tera- 
$10^{15} \approx 2^{50} \approx 1$ peta- 
$10^{18} \approx 2^{60} \approx 1$ exa- 
$10^{21} \approx 2^{70} \approx 1$ zetta- 
$10^{24} \approx 2^{80} \approx 1$ yotta-
Word-Oriented Memory Organization

- Addresses specify byte locations
  - Address of first byte in a word
  - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)
- Need to distinguish the notion of address vs. data stored in that address

<table>
<thead>
<tr>
<th></th>
<th>32-bit Words</th>
<th>64-bit Words</th>
<th>Bytes</th>
<th>Addr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr = 0000</td>
<td></td>
<td></td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>Addr = 0004</td>
<td></td>
<td></td>
<td>0004</td>
<td></td>
</tr>
<tr>
<td>Addr = 0008</td>
<td></td>
<td></td>
<td>0008</td>
<td></td>
</tr>
<tr>
<td>Addr = 0012</td>
<td></td>
<td></td>
<td>0012</td>
<td></td>
</tr>
</tbody>
</table>

- Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)
Byte Ordering

- How should bytes within a multi-byte word be ordered in memory?

- Conventions
  - **Big Endian**: Sun, PPC Mac, Internet
    - Most significant byte first
    - Least significant byte has **highest** address, i.e.
  - **Little Endian**: x86, ARM processors running Android, iOS and Windows
    - Least significant byte first
    - Least significant byte has **lowest** address, i.e.
Announcement 2/1/2016

- Homework Assignment #1 has been issued today
  - Download from Moodle class web page
  - Due date before class Wednesday 2/10/2016 (check class schedule)
- Data Lab due next Monday 2/8/2016
  - Should start asap, if not yet
  - Check Data Lab Forum for common Q&As, or post your problem there.
Review: Byte Ordering Example

- **Big Endian**
  - Most significant byte first
    - i.e. Least significant byte has highest address

- **Little Endian**
  - Least significant byte first
    - i.e. Least significant byte has lowest address

- **Example**
  - Variable x has 4-byte representation 0x76543210
  - Address given by &x is 0x100

<table>
<thead>
<tr>
<th>Big Endian</th>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>76</td>
<td>54</td>
<td>32</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Little Endian</th>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td>32</td>
<td>54</td>
<td>76</td>
</tr>
</tbody>
</table>
Representing Integers

int A = 15213;

long int C = 15213;

int B = -15213;
Examining Data Representations

- **Code to Print Byte Representation of Data**
  - Casting pointer to “unsigned char *” allows treatment as a byte array

```c
typedef unsigned char *pointer;

void show_bytes(pointer start, size_t len) {
    size_t i;
    for (i = 0; i < len; i++)
        printf("%p\t0x%.2x\n", start+i, start[i]);
    printf("\n");
}
```

**Printf directives:**
- `%p`: print pointer (i.e. address)
- `%x`: print Hexadecimal
- `\t`: tab
- `\n`: new line
show_bytes Execution Example

```c
int a = 15213;
printf("int a = 15213;\n");
show_bytes((pointer) &a, sizeof(int));
```

Result (Linux x86-64):

```c
int a = 15213;
0x7fffb7f71dbc 6d
0x7fffb7f71dbd 3b
0x7fffb7f71dbe 00
0x7fffb7f71dbf 00
```
Representing Pointers

```cpp
int B = -15213;
int *P = &B;
```

- Different compilers & machines assign different locations to objects
- Even get different results each time run program
Representing Strings

- **Strings in C**
  - Represented by array of characters
  - Each character encoded in ASCII format
    - Standard 7-bit encoding of character set
    - Character “0” has code 0x30
      - Digit i has code 0x30+i
  - String should be **null-terminated**
    - Final character = 0 # invisible

- **Compatibility**
  - Byte ordering not an issue

- **ASCII**: American Standard Code for Information Interchange
- **Unicode Consortium**
Move on to Floating Point Numbers