Bits, Bytes, and Integers – Part 2

15-213: Introduction to Computer Systems
3rd Lecture, Sept. 4, 2018
Assignment Announcements

- **Lab 0 available via course web page and Autolab.**
  - Due Thurs. Sept. 6, 11:59pm
  - No grace days
  - No late submissions
  - Just do it!

- **Lab 1 available via Autolab**
  - Due Thurs, Sept. 13, 11:59pm
  - Read instructions carefully: writeup, bits.c, tests.c
    - Quirky software infrastructure
  - Based on lectures 2, 3, and 4 (CS:APP Chapter 2)
  - After today’s lecture you will know everything for the integer problems
  - Floating point covered Thursday Sept. 6
Summary From Last Lecture

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

**Unsigned**

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

**Two’s Complement**

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

**Two’s Complement Examples (w = 5)**

<table>
<thead>
<tr>
<th>Number</th>
<th>Unsigned Binary</th>
<th>Two’s Complement Binary</th>
<th>Calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>-16</td>
<td>10000</td>
<td>11111</td>
<td>8 + 2 = 10</td>
</tr>
<tr>
<td>10</td>
<td>01010</td>
<td>01110</td>
<td></td>
</tr>
<tr>
<td>-10</td>
<td>10110</td>
<td>11000</td>
<td>-16 + 4 + 2 = -10</td>
</tr>
</tbody>
</table>
## Unsigned & Signed Numeric 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 values

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

**Expression containing signed and unsigned int:**

`int` is cast to `unsigned`
Sign Extension and Truncation

- Sign Extension

- Truncation
Today: Bits, Bytes, and Integers

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

Operands: \( w \) bits

True Sum: \( w+1 \) bits

Discard Carry: \( w \) bits

- **Standard Addition Function**
  - Ignores carry output

- **Implements Modular Arithmetic**
  \[
s = \text{UAdd}_w(u, v) = u + v \mod 2^w
\]

<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:

\[
\begin{array}{c}
\text{unsigned char} \quad 1110 \ 1001 \\
+ \quad 1101 \ 0101 \\
\hline
1 \ 1011 \ 1110
\end{array}
\]

\[
\begin{array}{c}
\text{E9} \quad 223 \\
+ \quad D5 \quad 213 \\
\hline
1BE \quad 446
\end{array}
\]

\[
\begin{array}{c}
\text{BE} \quad 190
\end{array}
\]
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 $\geq 2^w$
  - At most once

True Sum

2\(^{w+1}\) Overflow

2\(^w\) Modular Sum

0

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

Operands: \( w \) bits

\[
\begin{array}{c}
\vspace{3mm}
u \\
\vspace{3mm}
+ \\
\vspace{3mm}v \\
\hline
u + v
\end{array}
\]

True Sum: \( w+1 \) bits

\[
\begin{array}{c}
\vspace{3mm}
u + v
\end{array}
\]

Discard Carry: \( w \) bits

\[
\begin{array}{c}
\vspace{3mm}
\text{TAdd}_w(u, v)
\end{array}
\]

\[\text{TAdd and UAdd have Identical Bit-Level Behavior}\]

- Signed vs. unsigned addition in C:
  \[
  \begin{align*}
  \text{int } s, t, u, v; \\
  s &= \text{(int) } ((\text{unsigned}) u + (\text{unsigned}) v); \\
  t &= u + v
  \end{align*}
  \]

  Will give \( s == t \)

\[
\begin{array}{c}
\vspace{3mm}
1110 1001 & \quad E9 & \quad -23 \\
+ 1101 0101 & \quad \text{D5} & \quad + -43 \\
\hline
1 1011 1110 & \quad 1BE & \quad 446 \\
1011 1110 & \quad \text{BE} & \quad -66
\end{array}
\]

### TAdd Overflow

**Functionality**
- True sum requires $w+1$ bits
- Drop off 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>$2^w - 1$</td>
<td>$011...1$</td>
</tr>
<tr>
<td>$2^w - 1 - 1$</td>
<td>$000...0$</td>
</tr>
<tr>
<td>$0$</td>
<td>$100...0$</td>
</tr>
<tr>
<td>$-2^w$</td>
<td>$100...0$</td>
</tr>
<tr>
<td>$-2^w$</td>
<td>$100...0$</td>
</tr>
</tbody>
</table>

PosOver

NegOver
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
    - At most once
  - If sum $< -2^{w-1}$
    - Becomes positive
    - At most once
Characterizing TAdd

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

\[
\text{TAdd}_w(u, v) = \begin{cases} 
  u + v + 2^w & u + v < T\text{Min}_w \quad \text{(NegOver)} \\
  u + v & T\text{Min}_w \leq u + v \leq T\text{Max}_w \\
  u + v - 2^w & T\text{Max}_w < u + v \quad \text{(PosOver)} 
\end{cases}
\]
Multiplication

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

- **But, exact results can be bigger than \( w \) bits**
  - 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: \( x \times y \geq (-2^{w-1}) \times (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1} \)
  - Two’s complement max (positive): Up to \( 2w \) bits, but only for \((TMin_w)^2\)
    - Result range: \( x \times y \leq (-2^{w-1})^2 = 2^{2w-2} \)

- **So, maintaining exact results...**
  - would need to keep expanding word size with each product computed
  - is done in software, if needed
    - e.g., by “arbitrary precision” arithmetic packages
Unsigned Multiplication in C

Operands: \( w \) bits

\[
\begin{array}{c}
\text{True Product: } 2^w \text{ bits} \\
\hline
u \cdot v \\
\hline
\text{Discard } w \text{ bits: } w \text{ bits}
\end{array}
\]

\[
\text{UMult}_w(u, v) = u \cdot v \mod 2^w
\]

- **Standard Multiplication Function**
  - Ignores high order \( w \) bits

- **Implements Modular Arithmetic**
  \[
  \text{UMult}_w(u, v) = u \cdot v \mod 2^w
  \]

\[
\begin{array}{c}
1110 1001 \\
* \\
1101 0101 \\
\hline
1100 0001 1101 1101
\end{array}
\]

\[
\begin{array}{c}
E9 \\
* \\
D5 \\
* \\
\hline
C1DD \\
DD
\end{array}
\]

\[
\begin{array}{c}
1101 1101 \\
\hline
221
\end{array}
\]

\[
\begin{array}{c}
223 \\
47499
\end{array}
\]
Signed Multiplication in C

Operands: $w$ bits

True Product: $2^w$ 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

\[
\begin{align*}
\text{Operands: } &w \text{ bits} \\
\text{True Product: } &2^w \text{ bits} \\
\text{Discard } &w \text{ bits: } w \text{ bits}
\end{align*}
\]
Power-of-2 Multiply with Shift

Operation
- \( u << k \) gives \( u \times 2^k \)
- Both signed and unsigned

Operands: \( w \) bits

\[
\begin{array}{c}
\text{True Product: } w+k \text{ bits} \\
\text{Discard } k \text{ bits: } w \text{ bits}
\end{array}
\]

Examples
- \( u << 3 \) == \( u \times 8 \)
- \( (u << 5) - (u << 3) == u \times 24 \)
- Most machines shift and add faster than multiply
  - Compiler generates this code automatically
Unsigned Power-of-2 Divide with Shift

- Quotient of Unsigned by Power of 2
  - $u \gg k$ gives $\lfloor u / 2^k \rfloor$
  - Uses logical shift

### Division Example

<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>$x \gg 1$</td>
<td>7606.5</td>
<td>1D B6</td>
<td>00011101 10110110</td>
</tr>
<tr>
<td>$x \gg 4$</td>
<td>950.8125</td>
<td>03 B6</td>
<td>00000011 10110110</td>
</tr>
<tr>
<td>$x \gg 8$</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 $\lfloor x / 2^k \rfloor$
  - Uses arithmetic shift
  - Rounds wrong direction when $u < 0$

### Table

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y$</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>$y \gg 1$</td>
<td>-7606.5</td>
<td>E2 49</td>
<td>11100010 01001001</td>
</tr>
<tr>
<td>$y \gg 4$</td>
<td>-950.8125</td>
<td>FC 49</td>
<td>11111100 01001001</td>
</tr>
<tr>
<td>$y \gg 8$</td>
<td>-59.4257813</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)
- Compute as \( \left\lfloor \frac{x+2^k-1}{2^k} \right\rfloor \)
  - In C: \((x + (1<<k) -1) >> k\)
  - Biases dividend toward 0

Case 1: No rounding

Dividend:

\[
\begin{array}{c}
\text{u} \\
+2^k - 1 \\
\end{array}
\]

\[
\begin{array}{cccccccccc}
\text{k} & 1 & \cdots & 0 & \cdots & 0 & 0 \\
0 & \cdots & 0 & 0 & 1 & \cdots & 1 & 1 \\
\end{array}
\]

\[
\begin{array}{c}
\frac{u}{2^k} \\
\end{array}
\]

\[
\begin{array}{cccccccccc}
\text{k} & 1 & \cdots & 1 & 1 & \cdots & 1 & 1 \\
0 & \cdots & 0 & 1 & 0 & \cdots & 0 & 0 \\
\end{array}
\]

Binary Point

Divisor:

\[
\begin{array}{c}
\text{u} \\
\end{array}
\]

\[
\begin{array}{cccccccccc}
\text{k} & 1 & \cdots & 1 & 1 & \cdots & 1 & 1 \\
0 & \cdots & 0 & 1 & 0 & \cdots & 0 & 0 \\
\end{array}
\]

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

Case 2: Rounding

Dividend:

\[
x = 1.\underbrace{\ldots}_{k} \underbrace{0}_{+2^k - 1}
\]

Divisor:

\[
\frac{x}{2^k} = 1.\underbrace{\ldots}_{k} \underbrace{1}_{\text{Incremented by 1}} \underbrace{1}_{\text{Incremented by 1}} \underbrace{\ldots}_{\text{Incremented by 1}}
\]

Binary Point

\[
+2^k - 1 = 0.\underbrace{\ldots}_{k} \underbrace{001}_{\text{Incremented by 1}} \underbrace{\ldots}_{11}
\]

Biasing adds 1 to final result
Negation: Complement & Increment

- Negate through complement and increase
  \[ \neg x + 1 = -x \]

- Example
  - Observation: \( \neg x + x = 111\ldots111 = -1 \)

\[
\begin{align*}
x &= 111101101 \\
+ \quad \neg x &= 011000100 \\
\hline
-1 &= 111111111
\end{align*}
\]

\[ x = 15213 \]
### Complement & Increment Examples

#### x = 0

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00 00 0000 00000000</td>
</tr>
<tr>
<td>~0</td>
<td>-1</td>
<td>FF FF 11111111 11111111</td>
</tr>
<tr>
<td>~0+1</td>
<td>0</td>
<td>00 00 0000 00000000</td>
</tr>
</tbody>
</table>

#### x = TMin

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>-32768</td>
<td>80 00 10000000 00000000</td>
</tr>
<tr>
<td>~x</td>
<td>32767</td>
<td>7F FF 01111111 11111111</td>
</tr>
<tr>
<td>~x+1</td>
<td>-32768</td>
<td>80 00 10000000 00000000</td>
</tr>
</tbody>
</table>

**Canonical counter example**
Today: Bits, Bytes, and Integers

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

- **Addition:**
  - Unsigned/signed: 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:**
  - Unsigned/signed: Normal multiplication followed by truncate, same operation on bit level
  - Unsigned: multiplication mod $2^w$
  - Signed: modified multiplication mod $2^w$ (result in proper range)
Why Should I Use Unsigned?

- Don’t use without understanding implications
  - Easy to make mistakes
    ```c
    unsigned i;
    for (i = cnt-2; i >= 0; i--)
        a[i] += a[i+1];
    ```
  - Can be very subtle
    ```c
    #define DELTA sizeof(int)
    int i;
    for (i = CNT; i-DELTA >= 0; i-= DELTA)
        ...
    ```
Counting Down with Unsigned

- **Proper way to use unsigned as loop index**
  
  ```c
  unsigned i;
  for (i = cnt-2; i < cnt; i--)
    a[i] += a[i+1];
  ```

- **See Robert Seacord, Secure Coding in C and C++**
  - C Standard guarantees that unsigned addition will behave like modular arithmetic
    - $0 - 1 \rightarrow UMax$

- **Even better**
  
  ```c
  size_t i;
  for (i = cnt-2; i < cnt; i--)
    a[i] += a[i+1];
  ```
  - Data type `size_t` defined as unsigned value with length = word size
  - Code will work even if `cnt = UMax`
  - What if `cnt` is signed and < 0?
Why Should I Use Unsigned? (cont.)

- **Do Use When Performing Modular Arithmetic**
  - Multiprecision arithmetic

- **Do Use When Using Bits to Represent Sets**
  - Logical right shift, no sign extension

- **Do Use In System Programming**
  - Bit masks, device commands,...
Quiz Time!

Check out:

https://canvas.cmu.edu/courses/5835
Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
- Representations in memory, pointers, strings
Byte-Oriented Memory Organization

- **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

- **Note: system provides private address spaces to each “process”**
  - Think of a process as a program being executed
  - So, a program can clobber its own data, but not that of others
Machine Words

- Any given computer has a “Word Size”
  - Nominal size of integer-valued data
    - and of addresses
  - Until recently, most machines used 32 bits (4 bytes) as word size
    - Limits addresses to 4GB (2^{32} bytes)
  - Increasingly, machines have 64-bit word size
    - Potentially, could have 18 EB (exabytes) of addressable memory
    - That’s 18.4 X 10^{18}
  - Machines still 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 (32-bit) or 8 (64-bit)

<table>
<thead>
<tr>
<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>Addr = 0000</td>
<td></td>
<td>0000</td>
</tr>
<tr>
<td>Addr = 0004</td>
<td></td>
<td></td>
<td>0001</td>
</tr>
<tr>
<td>Addr = 0008</td>
<td></td>
<td></td>
<td>0002</td>
</tr>
<tr>
<td>Addr = 0012</td>
<td></td>
<td></td>
<td>0003</td>
</tr>
<tr>
<td></td>
<td>Addr = 0000</td>
<td></td>
<td>0004</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0005</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0006</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0007</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0008</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0009</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0011</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0012</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0013</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0014</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0015</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>pointer</td>
<td>4</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>
Byte Ordering

- So, how are the bytes within a multi-byte word ordered in memory?

- Conventions
  - Big Endian: Sun (Oracle SPARC), PPC Mac, *Internet*
    - Least significant byte has highest address
  - Little Endian: *x86*, ARM processors running Android, iOS, and Linux
    - Least significant byte has lowest address
### Byte Ordering Example

**Example**

- Variable x has 4-byte value of 0x01234567
- 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>01</td>
<td>23</td>
<td>45</td>
<td>67</td>
</tr>
</tbody>
</table>

| Little Endian | 0x100 | 0x101 | 0x102 | 0x103 |
|              | 67    | 45    | 23    | 01    |
Representing Integers

int A = 15213;

IA32, x86-64  Sun

6D
3B
00
00

long int C = 15213;

IA32  x86-64  Sun

6D
3B
00
00

6D
3B
00
00

int B = -15213;

IA32, x86-64  Sun

93
C4
FF
FF

Two’s complement representation

Decimal: 15213
Binary: 0011 1011 0110 1101
Hex: 3 B 6 D
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
- %x: Print Hexadecimal
##### 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;
0x7ffffb7f71dbc  6d
0x7ffffb7f71dbd  3b
0x7ffffb7f71dbe  00
0x7ffffb7f71dbf  00
```
## Representing Pointers

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

<table>
<thead>
<tr>
<th>Sun</th>
<th>IA32</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>EF</td>
<td>AC</td>
<td>3C</td>
</tr>
<tr>
<td>FF</td>
<td>28</td>
<td>1B</td>
</tr>
<tr>
<td>FB</td>
<td>F5</td>
<td>FE</td>
</tr>
<tr>
<td>2C</td>
<td>FF</td>
<td>82</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FD</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7F</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00</td>
</tr>
</tbody>
</table>

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

- **Compatibility**
  - Byte ordering not an issue
Reading Byte-Reversed Listings

- **Disassembly**
  - Text representation of binary machine code
  - Generated by program that reads the machine code

- **Example Fragment**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Code</th>
<th>Assembly Rendition</th>
</tr>
</thead>
<tbody>
<tr>
<td>8048365:</td>
<td>5b</td>
<td>pop %ebx</td>
</tr>
<tr>
<td>8048366:</td>
<td>81 c3 ab 12 00 00</td>
<td>add $0x12ab,%ebx</td>
</tr>
<tr>
<td>804836c:</td>
<td>83 bb 28 00 00 00 00</td>
<td>cmpl $0x0,0x28(%ebx)</td>
</tr>
</tbody>
</table>

- **Deciphering Numbers**
  - Value: 0x12ab
  - Pad to 32 bits: 0x000012ab
  - Split into bytes: 00 00 12 ab
  - Reverse: ab 12 00 00
Integer C Puzzles

Initialization

```
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```

<table>
<thead>
<tr>
<th>Puzzle</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x &lt; 0</code></td>
<td><code>x*2 &lt; 0</code></td>
</tr>
<tr>
<td><code>ux &gt;= 0</code></td>
<td>✓</td>
</tr>
<tr>
<td><code>x &amp; 7 == 7</code></td>
<td><code>(x&lt;&lt;30) &lt; 0</code></td>
</tr>
<tr>
<td><code>ux &gt; -1</code></td>
<td>✗</td>
</tr>
<tr>
<td><code>x &gt; y</code></td>
<td><code>-x &lt; -y</code></td>
</tr>
<tr>
<td><code>x * x &gt;= 0</code></td>
<td>✗</td>
</tr>
<tr>
<td><code>x &gt; 0 &amp;&amp; y &gt; 0</code></td>
<td><code>x + y &gt; 0</code></td>
</tr>
<tr>
<td><code>x &gt;= 0</code></td>
<td><code>-x &lt;= 0</code></td>
</tr>
<tr>
<td><code>x &lt;= 0</code></td>
<td><code>-x &gt;= 0</code></td>
</tr>
<tr>
<td>`(x</td>
<td>(~x)) &gt;&gt; 31 == -1`</td>
</tr>
<tr>
<td><code>ux &gt;&gt; 3 == ux/8</code></td>
<td>✓</td>
</tr>
<tr>
<td><code>x &gt;&gt; 3 == x/8</code></td>
<td>✗</td>
</tr>
<tr>
<td><code>x &amp; (x-1) != 0</code></td>
<td>✗</td>
</tr>
</tbody>
</table>
Summary

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