Bits, Bytes, and Integers – Part 2

15-213: Introduction to Computer Systems
3rd Lecture, Jan. 23, 2018

Instructors:
Franz Franchetti, Seth Copen Goldstein, Brian Railing
First Assignment: Data Lab

- Due: Thursday, Feb 1, 11:59:00 pm
- Last Possible Time to Turn in: Sun, Feb 4, 11:59PM
- Read the instructions carefully
- Please start ASAP
- Seek help (office hours start today!)
- Based on Lecture 2, 3, and 4
- After today’s lecture you know everything for the integer problems, float problems covered on Thursday
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
Bit-Level Operations in C

- **Operations & , | , ~ , ^ 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)**
  - \(~0\times41 \rightarrow 0\timesBE\)
    - \(~0100 0001_2 \rightarrow 1011 1110_2\)
  - \(~0\times00 \rightarrow 0\timesFF\)
    - \(~0000 0000_2 \rightarrow 1111 1111_2\)
  - \(0\times69 \& 0\times55 \rightarrow 0\times41\)
    - \(0110 1001_2 \& 0101 0101_2 \rightarrow 0100 0001_2\)
  - \(0\times69 \mid 0\times55 \rightarrow 0\times7D\)
    - \(0110 1001_2 \mid 0101 0101_2 \rightarrow 0111 1101_2\)
Logic Operations in C

- **Logic Operations: &&, ||, !**
  - 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
  - p && *p (avoids null pointer access)
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)

\[
\begin{align*}
-16 & = 0 1 0 1 0 & 8 + 2 & = 10 \\
10 & = 0 1 0 1 0 & 8 + 2 & = 10 \\
-16 & = 1 0 1 1 0 & -16 + 4 + 2 & = -10 \\
-10 & = 1 0 1 1 0 & &
\end{align*}
\]
Unsigned & Signed Numeric Values

- **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:**
  \[
  \text{int is cast to unsigned!!}
  \]
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

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

True Sum: \( w+1 \) bits

Discard Carry: \( w \) bits

\( UAdd_w(u, v) \)

- **Standard Addition Function**
  - Ignores carry output
- **Implements Modular Arithmetic**
  \[
s = UAdd_w(u, v) = u + v \mod 2^w
\]

Unsigned char

\[
\begin{array}{c}
\begin{array}{c}
1110 \\
+ 1101 \\
\hline
E9
\end{array}
\begin{array}{c}
1001 \\
0101 \\
D5
\hline
223
\end{array}
\begin{array}{c}
+ 213
\hline
223
\end{array}
\end{array}
\]

<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>
Unsigned Addition

Operands: \( w \) bits

True Sum: \( w+1 \) bits

Discard Carry: \( w \) bits

\[ u + v \]

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

- **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>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>A</td>
</tr>
<tr>
<td>1011</td>
<td>B</td>
</tr>
<tr>
<td>1100</td>
<td>C</td>
</tr>
<tr>
<td>1101</td>
<td>D</td>
</tr>
<tr>
<td>1110</td>
<td>E</td>
</tr>
<tr>
<td>1111</td>
<td>F</td>
</tr>
</tbody>
</table>

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

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

\[ \begin{array}{c}
223 \\
+ 213 \\
\hline
446
\end{array} \]

\[ \begin{array}{c}
1011 1110 \\
\hline
190
\end{array} \]
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
$2^{w+1}$
$2^w$
0

Modular Sum

Overflow

UAdd$_4(u, v)$
Two’s Complement Addition

Operands: \( w \) bits

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

True Sum: \( w+1 \) bits

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

Discard Carry: \( w \) bits

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

\textbf{TAdd and UAdd have Identical Bit-Level Behavior}

- Signed vs. unsigned addition in C:

  \[
  \begin{array}{l}
  \text{int } s, t, u, v; \\
  s = (\text{int}) ((\text{unsigned}) u + (\text{unsigned}) v); \\
  t = u + v
  \end{array}
  \]

- Will give \( s == t \)

\[
\begin{array}{cccc}
  1110 & 1001 & \text{E9} & -23 \\
  + & 1101 & 0101 & + \text{D5} \\
  \hline
  1 & 1011 & 1110 & 1\text{BE} \\
  1011 & 1110 & \text{BE} & -66
\end{array}
\]
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>0 111...1 $2^w-1$</td>
<td>011...1</td>
</tr>
<tr>
<td>0 100...0 $2^w-1-1$</td>
<td>000...0</td>
</tr>
<tr>
<td>0 000...0 0</td>
<td></td>
</tr>
<tr>
<td>1 011...1 $-2^w$</td>
<td>100...0</td>
</tr>
<tr>
<td>1 000...0</td>
<td></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

\[
TAdd(u, v) = \begin{cases} 
  u + v + 2^w & u + v < T_m, \quad w \quad \text{(NegOver)} \\
  u + v & T_m \leq u + v \leq T_M, \quad w \\
  u + v - 2^w & T_M < u + v \quad \text{(PosOver)}
\end{cases}
\]
Negation: Complement & Increment

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

- Example
  - Observation: \[ \neg x + x = 1111\ldots111 = -1 \]

\[
\begin{array}{c}
x \quad 10011101 \\
+ \quad \neg x \quad 01100010 \\
\hline \\
-1 \quad 11111111
\end{array}
\]

\[ x = 15213 \]

<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 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>( \neg x )</td>
<td>-15214</td>
<td>C4 92</td>
<td>11000100 10010010</td>
</tr>
<tr>
<td>( \neg x + 1 )</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>( y )</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
</tbody>
</table>
## Complement & Increment Examples

### $x = 0$

<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>~0</td>
<td>-1</td>
<td>FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>~0+1</td>
<td>0</td>
<td>00</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>

### $x = T_{\text{Min}}$

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

**Canonical counter example**

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

True Product: \( 2w \) bits

Discard \( w \) bits: \( w \) bits

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

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

\[
\begin{array}{c}
1110 1001 \\
\times \ \\
1101 0101 \\
\hline
1100 0001 1101 0010 \\
\hline
\text{E9} \quad \text{223}
\end{array}
\]

\[
\begin{array}{c}
1101 0101 \\
\times \ \\
1101 0101 \\
\hline
1101 1101 1101 1101 \\
\hline
\text{C1DD} \quad \text{47499}
\end{array}
\]
Signed Multiplication in C

Operands: \( w \) bits

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

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

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

\[
\begin{array}{c}
1100 0001 \\
1101 0010
\end{array}
\]

\[
\begin{array}{c}
E9 \\
* \\
D5 \\
* \\
-23 \\
-43
\end{array}
\]

\[
\begin{array}{c}
C1DD \\
16896
\end{array}
\]

\[
\begin{array}{c}
1101 1101 \\
DD \\
-35
\end{array}
\]
Power-of-2 Multiply with Shift

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

- **Examples**
  - \( u \ll 3 \) == \( u \times 8 \)
  - \((u \ll 5) - (u \ll 3)\) == \( u \times 24 \)
  - Most machines shift and add faster than multiply
    - Compiler generates this code automatically
Break Time!

taradiddle: "pretentious nonsense"

Check out: quiz: day 3: operators

https://canvas.cmu.edu/courses/3822/quizzes/9020
Unsigned Power-of-2 Divide with Shift

- **Quotient of Unsigned by Power of 2**
  - \( u >> k \) gives \( \lfloor u / 2^k \rfloor \)
  - Uses logical shift

### Division

<table>
<thead>
<tr>
<th>Value</th>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>15213</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>7606</td>
<td>1D B6</td>
<td>00011101 10110110</td>
</tr>
<tr>
<td>( x &gt;&gt; 4 )</td>
<td>950.8125</td>
<td>950</td>
<td>03 B6</td>
<td>00000011 10110110</td>
</tr>
<tr>
<td>( x &gt;&gt; 8 )</td>
<td>59.4257813</td>
<td>59</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 $u < 0$

**Operands:**

$$
\begin{array}{c|c}
\div \phantom{2^k} & \phantom{x} 2^k \\
\hline
x & \cdots \cdots \cdots \\
\hline
l & 0 \cdots 0 1 0 \cdots 0 0 \\
\hline
x / 2^k & \cdots \cdots \cdots \\
\end{array}
$$

**Division:**

$$
x / 2^k
$$

**Result:**

RoundDown($x / 2^k$)

<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 $\lceil x / 2^k \rceil$ (Round Toward 0)
- Compute as $\lfloor (x+2^k-1)/2^k \rfloor$
  - In C: $(x + (1<<k)-1) >> k$
  - Biases dividend toward 0

Case 1: No rounding

Dividend:

\[
\begin{array}{cccccc}
\text{u} & \cdots & \text{0} & \cdots & \text{0} & \text{0} \\
+2^k-1 & 0 & \cdots & 0 & 1 & \cdots & 1 & 1
\end{array}
\]

Divisor:

\[
\begin{array}{cccccc}
\text{u} & \cdots & \text{1} & \cdots & \text{1} & \text{1} \\
2^k & 0 & \cdots & 0 & 1 & 0 & \cdots & 0 & 0
\end{array}
\]

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

Case 2: Rounding

Dividend: \[ x + 2^k - 1 \]

Divisor: \[ \frac{x}{2^k} \]

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, shifting
- **Summary**

- Representations in memory, pointers, strings
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, ...
Integer Arithmetic Example

**unsigned char**

<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>

| 1111 0011 | F3 | 243 |
| + 0101 0010 | + 52 | + 82 |
| 1 0100 0101 | 145 | 325 |
| 0101 0101 | 45 | 69 |

| 0001 1001 | 19 | 25 |
| * 0000 0010 | * 02 | * 2 |
| 0 0011 0010 | 032 | 50 |
| 0011 0010 | 32 | 50 |

**unsigned char**
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 $18.4 \times 10^{18}$
  - 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>0000</td>
<td>0000</td>
</tr>
<tr>
<td>Addr = 0004</td>
<td>Addr = 0004</td>
<td>0001</td>
<td>0001</td>
</tr>
<tr>
<td>Addr = 0008</td>
<td>Addr = 0008</td>
<td>0002</td>
<td>0002</td>
</tr>
<tr>
<td>Addr = 0012</td>
<td>Addr = 0012</td>
<td>0003</td>
<td>0003</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0004</td>
<td>0004</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0005</td>
<td>0005</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0006</td>
<td>0006</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0007</td>
<td>0007</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0008</td>
<td>0008</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0009</td>
<td>0009</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0010</td>
<td>0010</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0011</td>
<td>0011</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0012</td>
<td>0012</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0013</td>
<td>0013</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0014</td>
<td>0014</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0015</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, PPC Mac, Internet
    - Least significant byte has highest address
  - Little Endian: x86, ARM processors running Android, iOS, and Windows
    - Least significant byte has lowest address
Byte Ordering Example

Example

- Variable x has 4-byte value of 0x01234567
- Address given by &x is 0x100

Big Endian

<table>
<thead>
<tr>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>23</td>
<td>45</td>
<td>67</td>
</tr>
</tbody>
</table>

Little Endian

<table>
<thead>
<tr>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td>67</td>
<td>45</td>
<td>23</td>
<td>01</td>
</tr>
</tbody>
</table>
Representing Integers

int A = 15213;

IA32, x86-64          Sun

IA32, x86-64          Sun

int B = -15213;

long int C = 15213;

IA32          x86-64          Sun

IA32          x86-64          Sun

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):**

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

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

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

```c
char S[6] = "18213";
```
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</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

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

1. \( x < 0 \)  \( \Rightarrow \)  \( ((x*2) < 0) \)
2. \( ux >= 0 \)
3. \( x & 7 == 7 \)  \( \Rightarrow \)  \( (x<<30) < 0 \)
4. \( ux > -1 \)
5. \( x > y \)  \( \Rightarrow \)  \( -x < -y \)
6. \( x * x >= 0 \)
7. \( x > 0 && y > 0 \)  \( \Rightarrow \)  \( x + y > 0 \)
8. \( x >= 0 \)  \( \Rightarrow \)  \( -x <= 0 \)
9. \( x <= 0 \)  \( \Rightarrow \)  \( -x >= 0 \)
10. \( (x|-x)>>31 == -1 \)
11. \( ux >> 3 == ux/8 \)
12. \( x >> 3 == x/8 \)
13. \( x & (x-1) != 0 \)
Integer C Puzzles

\[
\begin{align*}
\text{x < 0} & \quad \Rightarrow \quad ((x\times2) < 0) \quad \times \\
\text{ux >= 0} & \quad \Rightarrow \quad (x\ll30) < 0 \quad \checkmark \\
\text{x & 7 == 7} & \quad \Rightarrow \quad (x\ll30) < 0 \quad \checkmark \\
\text{ux > -1} & \quad \Rightarrow \quad -x < -y \quad \times \\
\text{x > y} & \quad \Rightarrow \quad -x < -y \quad \times \\
\text{x * x >= 0} & \quad \Rightarrow \quad x + y > 0 \quad \times \\
\text{x > 0 && y > 0} & \quad \Rightarrow \quad x + y > 0 \quad \times \\
\text{x >= 0} & \quad \Rightarrow \quad -x <= 0 \quad \checkmark \\
\text{x <= 0} & \quad \Rightarrow \quad -x >= 0 \quad \times \\
\text{(x|-x)>>31 == -1} & \quad \checkmark \\
\text{ux >> 3 == ux/8} & \quad \times \\
\text{x >> 3 == x/8} & \quad \times \\
\text{x & (x-1) != 0} & \quad \times \\
\text{Initialization} & \\
\text{int x = foo();} & \\
\text{int y = bar();} & \\
\text{unsigned ux = x;} & \\
\text{unsigned uy = y;} & \\
\end{align*}
\]
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