Bits, Bytes and Integers – Part 1

15-213/18-213/14-513/15-513: Introduction to Computer Systems
2nd Lecture, Jan. 17, 2019
Announcements

- Recitations are on Mondays. Recitation #1 is Jan 28.

- Linux Boot Camp Sunday evening 7pm, Rashid Auditorium

- Lab 0 is now available via course web page and Autolab.
  - Due Tue Jan. 22, 11:59pm
  - No grace days!
  - No late submissions!
  - Just do it!

- Problem Set 1 due Jan 20, 11:59pm
  - available via course webpage

- Do not email the course staff for logistics.
  - You will get better help faster if you use Piazza
Logistics

■ Waitlist
  ▪ 15-213: Mary Widom (marwidom@cs.cmu.edu)
  ▪ 18-213: ECE Academic services
    ece-asc@andrew.cmu.edu
  ▪ 15-513: Mary Widom (marwidom@cs.cmu.edu)
  ▪ Please don’t contact the instructors with waitlist questions.

■ Autolab + Canvas Accounts
  ▪ Doing our best to get things sorted for all 400+ of you
  ▪ We will have all accounting problems fixed ASAP
  ▪ There are pinned threads on Piazza for guidance
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
Everything is bits

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

- **Base 2 Number Representation**
  - Represent $15213_{10}$ as $11101101101101_2$
  - Represent $1.20_{10}$ as $1.0011001100110011[0011]..._2$
  - Represent $1.5213 \times 10^4$ as $1.11011011011012 \times 2^{13}$
Encoding Byte Values

- **Byte = 8 bits**
  - Binary: 00000000\(_2\) to 11111111\(_2\)
  - Decimal: 0\(_{10}\) to 255\(_{10}\)
  - Hexadecimal: 00\(_{16}\) to FF\(_{16}\)
    - Base 16 number representation
    - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
    - Write FA1D37B\(_{16}\) in C as
      - 0xFA1D37B
      - 0xfa1d37b

<table>
<thead>
<tr>
<th>Hex</th>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>0111</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>A</td>
<td>10</td>
<td>1010</td>
</tr>
<tr>
<td>B</td>
<td>11</td>
<td>1011</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>1100</td>
</tr>
<tr>
<td>D</td>
<td>13</td>
<td>1101</td>
</tr>
<tr>
<td>E</td>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>F</td>
<td>15</td>
<td>1111</td>
</tr>
</tbody>
</table>

15213: 0011 1011 0110 1101

3 B 6 D
## 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>
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
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 \( B=1 \)

<table>
<thead>
<tr>
<th>&amp;</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
- \( A | B = 1 \) 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
- \( \sim A = 1 \) when \( A=0 \)

<table>
<thead>
<tr>
<th>\sim</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)
- \( A \& B = 1 \) when either \( A=1 \) or \( B=1 \), but not both

<table>
<thead>
<tr>
<th>&amp;</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

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

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

**Representation**

- Width \( w \) bit vector represents subsets of \{0, ..., w−1\}
- \( a_j = 1 \) if \( j \in A \)

- 01101001 \{ 0, 3, 5, 6 \}
- 76543210

- 01010101 \{ 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 & , | , ~ , ^ 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)**
  - ~0x41 →
  - ~0x00 →
  - 0x69 & 0x55 →
  - 0x69 | 0x55 →
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)**
  - ~0x41 → 0xBE
    - ~0100 0001₂ → 1011 1110₂
  - ~0x00 → 0xFF
    - ~0000 0000₂ → 1111 1111₂
  - 0x69 & 0x55 → 0x41
    - 0110 1001₂ & 0101 0101₂ → 0100 0001₂
  - 0x69 | 0x55 → 0x7D
    - 0110 1001₂ | 0101 0101₂ → 0111 1101₂
Contrast: Logic Operations in C

- **Contrast to Bit-Level Operators**
  - 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)

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

- **Left Shift:** $x << y$
  - Shift bit-vector $x$ left $y$ positions
    - Throw away extra bits on left
      - Fill with 0’s on right
- **Right Shift:** $x >> y$
  - Shift bit-vector $x$ right $y$ positions
    - Throw away extra bits on right
  - Logical shift
    - Fill with 0’s on left
  - Arithmetic shift
    - Replicate most significant bit on left

- **Undefined Behavior**
  - Shift amount $< 0$ or $\geq$ word size

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

<table>
<thead>
<tr>
<th>Argument $x$</th>
<th>10100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>$&lt;&lt; 3$</td>
<td>00010000</td>
</tr>
<tr>
<td>Log. $&gt;&gt; 2$</td>
<td>00101000</td>
</tr>
<tr>
<td>Arith. $&gt;&gt; 2$</td>
<td>11101000</td>
</tr>
</tbody>
</table>
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

Summary
Encoding Integers

Unsigned

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

Two’s Complement

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

- C does not mandate using two’s complement
  - But, most machines do, and we will assume so

- C short 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>00111011 01101101</td>
</tr>
<tr>
<td>(y)</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
</tbody>
</table>

- Sign Bit
  - For 2’s complement, most significant bit indicates sign
    - 0 for nonnegative
    - 1 for negative
Two-complement: Simple Example

\[
\begin{array}{ccccc}
-16 & 8 & 4 & 2 & 1 \\
10 & = & 0 & 1 & 0 & 1 & 0 \\
\end{array}
\quad 8+2 = 10
\]

\[
\begin{array}{ccccc}
-16 & 8 & 4 & 2 & 1 \\
-10 & = & 1 & 0 & 1 & 1 & 0 \\
\end{array}
\quad -16+4+2 = -10
\]
Two-complement Encoding Example (Cont.)

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

<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>

**Sum**

<table>
<thead>
<tr>
<th></th>
<th>15213</th>
<th>-15213</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sum</td>
<td>15213</td>
<td>-15213</td>
</tr>
</tbody>
</table>
### Numeric Ranges

#### Unsigned 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
- Minus 1
  - 111...1

#### Values for \( W = 16 \)

<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 FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>TMax</td>
<td>32767</td>
<td>7F FF</td>
<td>01111111 11111111</td>
</tr>
<tr>
<td>Tmin</td>
<td>-32768</td>
<td>80 00</td>
<td>10000000 00000000</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>
## Values for Different Word Sizes

<table>
<thead>
<tr>
<th>W</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>255</td>
<td>65,535</td>
<td>4,294,967,295</td>
<td>18,446,744,073,709,551,615</td>
</tr>
<tr>
<td>Tmax</td>
<td>127</td>
<td>32,767</td>
<td>2,147,483,647</td>
<td>9,223,372,036,854,775,807</td>
</tr>
<tr>
<td>Tmin</td>
<td>-128</td>
<td>-32,768</td>
<td>-2,147,483,648</td>
<td>-9,223,372,036,854,775,807</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
### 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

#### 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

<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>
Quiz Time!

Check out:

https://canvas.cmu.edu/courses/8555
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
Mapping Between Signed & Unsigned

Two’s Complement

Unsigned

Mappings between unsigned and two’s complement numbers:
Keep bit representations and reinterpret
Mapping Signed $\leftrightarrow$ 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>
## Mapping Signed ↔ Unsigned

<table>
<thead>
<tr>
<th>Bits</th>
<th>Signed</th>
<th>Signed</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>

The mapping is done by treating the first bit as +/− and the rest as a positive unsigned number.
Relation between Signed & Unsigned

Two’s Complement

\[ x \rightarrow \text{T2B} \rightarrow \text{T2U} \rightarrow \text{B2U} \rightarrow ux \]

Maintain Same Bit Pattern

\[ \begin{array}{c}
| \text{w-1} | 0 \\
\hline
\text{ux} & + & + & + & \ldots & + & + & + \\
\text{x} & - & + & + & \ldots & + & + & + \\
\end{array} \]

Large negative weight

becomes

Large positive weight
Conversion Visualized

- **2’s Comp. → Unsigned**
  - Ordering Inversion
  - Negative → Big Positive
Signed vs. Unsigned in C

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

- **Casting**
  - Explicit casting between signed & unsigned same as U2T and T2U
    ```c
    int tx, ty;
    unsigned ux, uy;
    tx = (int) ux;
    uy = (unsigned) ty;
    ```
  - Implicit casting also occurs via assignments and procedure calls
    ```c
    tx = ux;  
    int fun(unsigned u);  
    uy = ty;  
    uy = fun(tx);  
    ```
## 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 $W = 32$: $\text{TMIN} = -2,147,483,648$, $\text{TMAX} = 2,147,483,647$

<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 ↔ Unsigned: Basic Rules

- Bit pattern is maintained
- But reinterpreted
- Can have unexpected effects: adding or subtracting $2^w$

Expression containing signed and unsigned int
  - int is cast to 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, shifting
- Summary

Representations in memory, pointers, strings
Sign Extension

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

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

\[ k \text{ copies of MSB} \]
Sign Extension: Simple Example

Positive number

\[
\begin{array}{cccccc}
-16 & 8 & 4 & 2 & 1 \\
0 & 1 & 0 & 1 & 0 \\
\end{array}
\]

Negative number

\[
\begin{array}{cccccc}
-16 & 8 & 4 & 2 & 1 \\
1 & 0 & 1 & 1 & 0 \\
\end{array}
\]

\[
\begin{array}{cccccc}
-32 & 16 & 8 & 4 & 2 & 1 \\
0 & 0 & 1 & 0 & 1 & 0 \\
\end{array}
\]

\[
\begin{array}{cccccc}
-32 & 16 & 8 & 4 & 2 & 1 \\
1 & 1 & 0 & 1 & 1 & 0 \\
\end{array}
\]

Positive number

\[
\begin{array}{cccccc}
-16 & 8 & 4 & 2 & 1 \\
0 & 1 & 0 & 1 & 0 \\
\end{array}
\]

Negative number

\[
\begin{array}{cccccc}
-16 & 8 & 4 & 2 & 1 \\
1 & 0 & 1 & 1 & 0 \\
\end{array}
\]

\[
\begin{array}{cccccc}
-32 & 16 & 8 & 4 & 2 & 1 \\
0 & 0 & 1 & 0 & 1 & 0 \\
\end{array}
\]

\[
\begin{array}{cccccc}
-32 & 16 & 8 & 4 & 2 & 1 \\
1 & 1 & 0 & 1 & 1 & 0 \\
\end{array}
\]
Larger Sign Extension Example

```
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</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</td>
<td>11111111 11111111 11000100 10010011</td>
</tr>
</tbody>
</table>

- Converting from smaller to larger integer data type
- C automatically performs sign extension
Truncation

■ Task:
  - Given $k+w$-bit signed or unsigned integer $X$
  - Convert it to $w$-bit integer $X'$
    - (with same value for “small enough” $X$)

■ Rule:
  - Drop top $k$ bits:
  - $X' = x_{w-1}, x_{w-2}, ..., x_0$
Truncation: Simple Example

**No sign change**

\[
\begin{array}{c|ccccc}
 2 &=& -16 & 8 & 4 & 2 & 1 \\
 2 &=& 0 & 0 & 0 & 1 & 0 \\
-6 &=& -16 & 8 & 4 & 2 & 1 \\
-6 &=& 1 & 1 & 0 & 1 & 0 \\
\end{array}
\]

\[2 \mod 16 = 2\]

**Sign change**

\[
\begin{array}{c|ccccc}
 10 &=& -16 & 8 & 4 & 2 & 1 \\
 10 &=& 0 & 1 & 0 & 1 & 0 \\
-6 &=& -16 & 8 & 4 & 2 & 1 \\
-10 &=& 1 & 0 & 1 & 1 & 0 \\
\end{array}
\]

\[10 \mod 16 = 10U \mod 16 = 10U = -6\]

\[
\begin{array}{c|ccccc}
-6 &=& -16 & 8 & 4 & 2 & 1 \\
-6 &=& 1 & 0 & 1 & 0 \\
\end{array}
\]

\[-6 \mod 16 = 26U \mod 16 = 10U = -6\]
Summary: Expanding, Truncating: Basic Rules

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

- **Truncating (e.g., unsigned to unsigned short)**
  - Unsigned/signed: bits are truncated
  - Result reinterpreted
  - Unsigned: mod operation
  - Signed: similar to mod
  - For small (in magnitude) numbers yields expected behavior
Summary of 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