Bits, Bytes and Integers – Part 1

15-213/18-213/15-513: Introduction to Computer Systems
2nd Lecture, Jan. 18, 2018

Instructors:
Franz Franchetti
Seth Copen Goldstein
Brian Railing
Announcements

- **Waitlist**
  - Please be patient.
  - Get the work done
    - so you will be ready when you get into the class

- **Linux Bootcamp this Sunday, Rashid, 7pm**

- **First Recitation this coming Monday**

- **AIV Course, please complete by Sunday**

- **Lab 0 Due this Sunday**
  - No grace days
  - No late submissions

- **Redshelf will be on canvas soon**
Waitlist questions

- 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.
Bootcamp

- 7pm Sunday in Rashid
- Linux basics
- Git basics

Things like:

- How to ssh to the shark machines from windows or linux
- How to setup a directory on afs with the right permissions
- How to initialize a directory for git
- The basics of using git as you work on the assignment
- Basic linux tools like: tar, make, gcc, ...
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

![Graph showing voltage levels and states](image-url)
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 $1.1101101101101_2 \times 2^{13}$
Encoding Byte Values

- **Byte = 8 bits**
  - Binary 00000000₂ to 11111111₂
  - Decimal: 0₁₀ to 255₁₀
  - Hexadecimal 00₁₆ to FF₁₆
    - Base 16 number representation
    - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
    - Write FA1D37B₁₆ in C as
      - 0xFA1D37B
      - 0xfa1d37b

<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

<table>
<thead>
<tr>
<th>And</th>
<th>Or</th>
</tr>
</thead>
<tbody>
<tr>
<td>A&amp;B</td>
<td>A</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Not</th>
<th>Exclusive-Or (Xor)</th>
</tr>
</thead>
<tbody>
<tr>
<td>~A</td>
<td>A^B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- **Exclusive-Or (Xor)**
  - A^B = 1 when either A=1 or B=1, but not both
General Boolean Algebras

- **Operate on Bit Vectors**
  - Operations applied bitwise

  \[
  \begin{align*}
  01101001 & \quad 01101001 & \quad 01101001 \\
  & \& 01010101 & | 01010101 & ^ 01010101 & ~ 01010101 \\
  01000001 & \quad 01111101 & \quad 00111100 & \quad 10101010
  \end{align*}
  \]

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

- **Representation**
  - Width w bit vector represents subsets of \( \{0, \ldots, w-1\} \)
  - \( a_j = 1 \) 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)
  - \(~0x41\) → 
  - \(~0x00\) → 
  - \(0x69 \& 0x55\) → 
  - \(0x69 \| 0x55\) → 

<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>
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)
- \( \sim 0x41 \rightarrow 0xBE \)
  - \( \sim 0100 \ 0001_2 \rightarrow 1011 \ 1110_2 \)
- \( \sim 0x00 \rightarrow 0xFF \)
  - \( \sim 0000 \ 0000_2 \rightarrow 1111 \ 1111_2 \)
- \( 0x69 \ & \ 0x55 \rightarrow 0x41 \)
  - \( 0110 \ 1001_2 \ & \ 0101 \ 0101_2 \rightarrow 0100 \ 0001_2 \)
- \( 0x69 \ | \ 0x55 \rightarrow 0x7D \)
  - \( 0110 \ 1001_2 \ | \ 0101 \ 0101_2 \rightarrow 0111 \ 1101_2 \)
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
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 \(\rightarrow\) 0x00
  - !0x00 \(\rightarrow\) 0x01
  - !!0x41 \(\rightarrow\) 0x01
  - 0x69 && 0x55 \(\rightarrow\) 0x01
  - 0x69 || 0x55 \(\rightarrow\) 0x01
  - \(p && *p\) (avoids null pointer access)
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
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 short 2 bytes long**

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>15213</td>
<td>3B 6D 00111011 01101101</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>C4 93 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

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

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

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

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

<table>
<thead>
<tr>
<th>Sum</th>
<th>15213</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sum</th>
<th>-15213</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

\[ -32768 \]
## 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>-9,223,372,036,854,775,808</td>
</tr>
</tbody>
</table>

#### Observations
- \(|T_{Min}| = T_{Max} + 1\)
- Asymmetric range
- \(U_{Max} = 2 \times T_{Max} + 1\)

#### 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>
Break Time!
bumfuzzle: "to confuse or fluster"

Check out:

https://canvas.cmu.edu/courses/3822
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

\[ x \xrightarrow{T2B} X \xrightarrow{B2U} \]

Unsigned

\[ x \xrightarrow{UX} \]

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>

U2T (Unsigned to Signed) and T2U (Signed to Unsigned) mappings are shown for each case.
## 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>

\[=\]

\[+/- 16\]
Relation between Signed & Unsigned

Two’s Complement

\( x \)

T2B

T2U

B2U

Maintain Same Bit Pattern

Unsigned

\( u_x \)

Relation between Signed & Unsigned

Large negative weight

becomes

Large positive weight
Conversion Visualized

- 2’s Comp. → Unsigned
  - Ordering Inversion
  - Negative → Big Positive

2’s Complement Range

Unsigned Range

$T_{Max}$

$U_{Max}$

$U_{Max} - 1$

$T_{Max} + 1$

$T_{Max}$

0

-2

-1

0
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 $W = 32$: $TMIN = -2,147,483,648$, $TMAX = 2,147,483,647$

### Constant

<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><code>&lt;</code></td>
<td>signed</td>
</tr>
<tr>
<td>-1</td>
<td>0U</td>
<td><code>&gt;</code></td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>-2147483647-1</td>
<td><code>&gt;</code></td>
<td>signed</td>
</tr>
<tr>
<td>2147483647U</td>
<td>-2147483647-1</td>
<td><code>&lt;</code></td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>-2</td>
<td><code>&gt;</code></td>
<td>signed</td>
</tr>
<tr>
<td>(unsigned)-1</td>
<td>-2</td>
<td><code>&gt;</code></td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>2147483648U</td>
<td><code>&lt;</code></td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>(int) 2147483648U</td>
<td><code>&gt;</code></td>
<td>signed</td>
</tr>
</tbody>
</table>
Unsigned vs. Signed: Easy to Make Mistakes

unsigned i;
for (i = cnt-2; i >= 0; i--)
    a[i] += a[i+1];

- Can be very subtle

#define DELTA sizeof(int)
int i;
for (i = CNT; i-DELTA >= 0; i-= DELTA)
    ...

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 $w$-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 showing sign extension](image)
Sign Extension: Simple Example

Positive number

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

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

Negative number

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

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

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

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

<table>
<thead>
<tr>
<th>No sign change</th>
<th>Sign change</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>2 =</strong></td>
<td></td>
</tr>
<tr>
<td>[-16 8 4 2 1]</td>
<td></td>
</tr>
<tr>
<td>[0 0 0 1 0]</td>
<td></td>
</tr>
<tr>
<td>[-8 4 2 1]</td>
<td></td>
</tr>
<tr>
<td>[0 0 1 0]</td>
<td></td>
</tr>
<tr>
<td>2 mod 16 = 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>2 =</strong></td>
<td></td>
</tr>
<tr>
<td>[-16 8 4 2 1]</td>
<td></td>
</tr>
<tr>
<td>[0 0 0 1 0]</td>
<td></td>
</tr>
<tr>
<td>[-8 4 2 1]</td>
<td></td>
</tr>
<tr>
<td>[0 0 1 0]</td>
<td></td>
</tr>
<tr>
<td>-6 mod 16 = 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>-6 =</strong></td>
<td></td>
</tr>
<tr>
<td>[-16 8 4 2 1]</td>
<td></td>
</tr>
<tr>
<td>[1 1 0 1 0]</td>
<td></td>
</tr>
<tr>
<td>[-8 4 2 1]</td>
<td></td>
</tr>
<tr>
<td>[1 0 1 0]</td>
<td></td>
</tr>
<tr>
<td>-6 mod 16 = 2</td>
<td></td>
</tr>
<tr>
<td>-6 mod 16 = 26U mod 16 = 10U = -6</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>-10 =</strong></td>
<td></td>
</tr>
<tr>
<td>[-16 8 4 2 1]</td>
<td></td>
</tr>
<tr>
<td>[1 0 1 1 0]</td>
<td></td>
</tr>
<tr>
<td>[-8 4 2 1]</td>
<td></td>
</tr>
<tr>
<td>[1 0 1 0]</td>
<td></td>
</tr>
<tr>
<td>-10 mod 16 = 2</td>
<td></td>
</tr>
<tr>
<td>-10 mod 16 = 22U mod 16 = 10U = -6</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>2 mod 16 = 2</strong></td>
<td></td>
</tr>
<tr>
<td><strong>-6 mod 16 = 2</strong></td>
<td></td>
</tr>
<tr>
<td><strong>-10 mod 16 = 2</strong></td>
<td></td>
</tr>
<tr>
<td><strong>6 mod 16 = 10U mod 16 = 6U = 6</strong></td>
<td></td>
</tr>
</tbody>
</table>
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 numbers yields expected behavior
Fake real world example

- Acme, Inc. has developed a state of the art voltmeter they are connecting to a pc. It is precise to the millivolt and does not drain the unit under test.
- Your job is to develop the driver software.

printf("%d\n", getValue());
Fake real world example

- Acme, Inc. has developed a state of the art voltmeter they are connecting to a pc. It is precise to the millivolt and does not drain the unit under test.
- Your job is to develop the driver software.

```c
printf("%d\n", getValue());
```
Lets run some tests

```c
printf("%d\n", getValue());
```

- 50652
- 1500
- 9692
- 26076
- 17884
- 42460
- 34268
- 50652
Lets run some tests

```c
int x=getValue(); printf("%d %08x\n", x, x);
```

- 50652 0000c5dc
- 1500 000005dc
- 9692 000025dc
- 26076 000065dc
- 17884 000045dc
- 42460 0000a5dc
- 34268 000085dc
- 50652 0000c5dc

Those darn engineers!
Only care about least significant 12 bits

```c
int x=getValue();
x=(x & 0x0fff);
printf("%d\n",x);
```
Only care about least significant 12 bits

```c
int x = getValue();
x = x(&0x0fff);
printf("%d\n", x);

printf("%x\n", x);
```

hmm?
Must sign extend

```c
int x = getValue();
x = (x & 0x007ff) | (x & 0x0800 ? 0xfffff000 : 0);
printf("%d\n", x);
```

There is a better way.
Because you graduated from 213

```c
int x=getValue();
x=(x&0x007ff)|(x&0x0800?0xffffffff:0);
printf("%d\n",x);
```

huh?
Lets be really thorough

```c
int x=getValue();
x=(x&0x00fff)|(x&0x0800?0xfffff000:0);
printf("%d\n",x);
```
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