Bits, Bytes and Integers – Part 1

15-213/18-213/15-513: Introduction to Computer Systems
2\textsuperscript{nd} Lecture, May 23, 2018

Instructors:
Brian Railing
Waitlist questions

- 15-213: Amy Weis alweis@andrew.cmu.edu
- 18-213: Zara Collier (zcollier@andrew.cmu.edu)
- 15-513: Amy Weis alweis@andrew.cmu.edu

Please don’t contact the instructors with waitlist questions.
Bootcamp

- 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, …
First Assignment: Data Lab

- Datalab is out this afternoon
- Due: Thursday, 5/31 at 11:59pm
- Absolute last time to turn in: Saturday, 6/2 at 11:59pm
- Go to GitHub/Autolab soon and read the handout carefully
- Start early
- Don’t be afraid to ask for help
  - Piazza
  - Office hours
- Based on lectures 2, 3 and 4
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 $1.1101101101101_2 \times 2^{13}$
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: \[ \binom{0011}{1011}01101101 \]

- \(3\) \(B\) \(6\) \(D\)
## 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
- $A \mid B = 1$ 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>~</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

### Exclusive-Or (Xor)
- $A \wedge B = 1$ when either $A=1$ or $B=1$, but not both

<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>0</td>
</tr>
</tbody>
</table>
General Boolean Algebras

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

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

- **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 \( 0100001 \) \{ 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 →

### Hex | Decimal | Binary

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></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)**
  - \(~0x41 \rightarrow 0xBE\)
    - \(~0100 0001_2 \rightarrow 1011 1110_2\)
  - \(~0x00 \rightarrow 0xFF\)
    - \(~0000 0000_2 \rightarrow 1111 1111_2\)
  - \(0x69 \& 0x55 \rightarrow 0x41\)
    - \(0110 1001_2 \& 0101 0101_2 \rightarrow 0100 0001_2\)
  - \(0x69 \mid 0x55 \rightarrow 0x7D\)
    - \(0110 1001_2 \mid 0101 0101_2 \rightarrow 0111 1101_2\)
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**
  - !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: $x \ll y$**
  - Shift bit-vector $x$ left $y$ positions
    - Throw away extra bits on left
      - Fill with 0’s on right
  - **Argument $x$**
    | $x$ | 01100010 |
    | $x \ll 3$ | 00010000 |
    | Log. $x \gg 2$ | 00011000 |
    | Arith. $x \gg 2$ | 00011000 |

- **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
  - **Argument $x$**
    | $x$ | 10100010 |
    | $x \ll 3$ | 00010000 |
    | Log. $x \gg 2$ | 00101000 |
    | Arith. $x \gg 2$ | 11101000 |

- **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

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

8+2 = 10

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

-16+4+2 = -10
Two-complement Encoding Example (Cont.)

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

<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>

<table>
<thead>
<tr>
<th>Sum</th>
<th>15213</th>
<th>-15213</th>
</tr>
</thead>
</table>

Numeric Ranges

- **Unsigned Values**
  - $U_{\text{Min}} = 0$
  - $000...0$
  - $U_{\text{Max}} = 2^w - 1$
  - $111...1$

- **Two's Complement Values**
  - $T_{\text{Min}} = -2^{w-1}$
  - $100...0$
  - $T_{\text{Max}} = 2^{w-1} - 1$
  - $011...1$
  - 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></td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>00 00</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>
Values for Different Word Sizes

| W | \(| T_{\text{Min}} | = T_{\text{Max}} + 1 \) | Asymmetric range |
|---|---|---|
| 8 | 16 | 32 | 64 |
| UMax | 255 | 65,535 | 4,294,967,295 | 18,446,744,073,709,551,615 |
| Tmax | 127 | 32,767 | 2,147,483,647 | 9,223,372,036,854,775,807 |
| Tmin | -128 | -32,768 | -2,147,483,648 | -9,223,372,036,854,775,808 |

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

### 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 complement 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>
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
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></td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>-8</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>-7</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>-6</td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>-5</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>-4</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>-3</td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>-2</td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Mapping Signed ↔ Unsigned

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

Two’s Complement

\[ x \]

\[ \text{T2B} \rightarrow \text{T2U} \rightarrow \text{B2} \rightarrow u \]

Unsigned

\[ ux \]

Maintain Same Bit Pattern

\[ w-1 \]

\[ u \]

\[ \hat{x} \]

Large negative weight

becomes

Large positive weight
Conversion Visualized

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

2’s Complement Range

TMin

0

-1

-2

TMax

UMax

UMax - 1

TMax + 1

TMax

0

Unsigned Range
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
    - ```
      int tx, ty;
      unsigned ux, uy;
      tx = (int) ux;
      uy = (unsigned) ty;
    ```
  - Implicit casting also occurs via assignments and procedure calls
    - ```
      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$_1$** | **Constant$_2$** | **Relation** | **Evaluation**
  - 0 0U == unsigned
  - -1 0 < signed
  - -1 0U > unsigned
  - 2147483647-2147483647-1 > signed
  - 2147483647U -2147483647U <=1 unsigned
  - -1 -2 > signed
  - (unsigned)-1-2 > unsigned
  - 2147483647 2147483648 unsigned
  - 2147483647 (int) 2147483648U signed
Unsigned vs. Signed: Easy to Make Mistakes

- Can be very subtle

```c
#include <stdio.h>

#define DELTA sizeof(int)

int main()

    unsigned i;
    for (i = cnt-2; i >= 0; i--)
        a[i] += a[i+1];
```
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:
  - \[ X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0 \]

\[ \begin{array}{c}
k \text{ copies of MSB} \\
\end{array}\]

\[ \begin{array}{c}
X \\
\end{array}\]

\[ \begin{array}{c}
X' \\
\end{array}\]

\[ \begin{array}{c}
\vdots \\
\end{array}\]

\[ \begin{array}{c}
w \\
\end{array}\]

\[ \begin{array}{c}
\vdots \\
\end{array}\]

\[ \begin{array}{c}
k \\
\end{array}\]

\[ \begin{array}{c}
w \\
\end{array}\]
# Sign Extension: Simple Example

<table>
<thead>
<tr>
<th>Positive number</th>
<th>Negative number</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 = 10</td>
<td>-10 = -10</td>
</tr>
<tr>
<td>-16 8 4 2 1</td>
<td>-16 8 4 2 1</td>
</tr>
<tr>
<td>0 1 0 1 0</td>
<td>1 1 1 1 0</td>
</tr>
</tbody>
</table>

10 = 10

-32 16 8 4 2 1
0 0 1 0 1 0

-32 16 8 4 2 1
1 1 1 1 0 1 0
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></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>15213</td>
<td>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>ix</td>
<td>15213</td>
<td>00 00 3B 6D</td>
<td>00000000 00000000 00111011 01101101</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>iy</td>
<td>-15213</td>
<td>FF FF C4 93</td>
<td>11111111 11111111 11000100 10010011</td>
</tr>
</tbody>
</table>

- Converting from smaller to larger integer data type
- C automatically performs sign extension
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:
- \( X' = x_{w-1}, x_{w-2}, \ldots, x_0 \)
# Truncation: Simple Example

## No sign change

<table>
<thead>
<tr>
<th>2</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>-8</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>-6</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

2 mod 16 = 2

-6 mod 16 = 26

U mod 16 = 10

U = -6

## Sign change

<table>
<thead>
<tr>
<th>10</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>-10</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

10 mod 16 = 10

U mod 16 = 10

U = -6

-10 mod 16 = 22

U mod 16 = 6

U = 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 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.

```c
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());
```

wtf?
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
”, x);
```
Only care about least significant 12 bits

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

printf("%x\n", x);
Must sign extend

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

There is a better way.
Because you graduated from 213

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

 huh?
Let's 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