Integer Arithmetic
Native Data Representations in C

- **char, short, int, and long** are “integer” types
  - Signed by default
  - `int x;`
  - Can declare as unsigned
  - `unsigned int x;`
- **float and double** are “real” types
- A pointer is a data type that holds a memory address

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Typical 32-bit</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>pointer</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>
Values for Different Word Sizes

\[
\begin{array}{|c|c|c|c|c|}
\hline
 & 8 & 16 & 32 & 64 \\
\hline
U\text{Max} & 255 & 65,535 & 4,294,967,295 & 18,446,744,073,709,551,615 \\
\hline
T\text{Max} & 127 & 32,767 & 2,147,483,647 & 9,223,720,368,547,758,077 \\
\hline
T\text{Min} & -128 & -32,768 & -2,147,483,648 & -9,223,720,368,547,758,088 \\
\hline
\end{array}
\]

- 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 are platform specific
Mapping Between Signed & Unsigned

• Mappings between unsigned and two’s complement numbers: Keep the same 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>3</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>0111</td>
<td>-7</td>
<td>8</td>
</tr>
<tr>
<td>1000</td>
<td>-8</td>
<td>9</td>
</tr>
<tr>
<td>1001</td>
<td>-7</td>
<td>10</td>
</tr>
<tr>
<td>1010</td>
<td>-6</td>
<td>11</td>
</tr>
<tr>
<td>1011</td>
<td>-5</td>
<td>12</td>
</tr>
<tr>
<td>1100</td>
<td>-4</td>
<td>13</td>
</tr>
<tr>
<td>1101</td>
<td>-3</td>
<td>14</td>
</tr>
<tr>
<td>1110</td>
<td>-2</td>
<td>15</td>
</tr>
<tr>
<td>1111</td>
<td>-1</td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**
- T2U (Signed to Unsigned) and U2T (Unsigned to Signed) mappings are shown.
- The table illustrates how bits are translated between signed and unsigned representations.

**Date:** 1/31/22

**Course:** CMPU 224 -- Computer Organization

**Page:** 5
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

Maintain Same Bit Pattern

Unsigned

T2B

B2U

T2U

x

ux

Large negative weight

becomes

Large positive weight

Relation between Signed & Unsigned

Two’s Complement

Maintain Same Bit Pattern

Unsigned

T2B

B2U

T2U

x

ux

Large negative weight

becomes

Large positive weight
Signed vs. Unsigned in C

• Constants
  • Are by default 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 int ux, uy;
    tx = (int) ux;
    uy = (unsigned) ty;

  • Implicit casting also occurs via assignments and procedure calls
    tx = ux;
    uy = ty;
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:
    • \( X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0 \)
  • Converting from a smaller to larger integer data type
    • C automatically performs sign extension
Expanding and Truncating Rules

• Expanding (e.g., \texttt{short} to \texttt{int})
  • Unsigned: zeros added
  • Signed: sign extension
  • Both yield expected result

• Truncating (e.g., \texttt{int} to \texttt{short})
  • Unsigned/signed: high order bits are truncated (drop)
  • Result reinterpreted
  • For small numbers this yields expected behavior
    \begin{align*}
    1111010 \rightarrow -6 & \quad 8\text{-bit two’s complement} \\
    1010 \rightarrow -6 & \quad 4\text{-bit two’s complement}
    \end{align*}
  • Overflow can result in a sign change
Unsigned Addition

- Standard Addition Function
  - Ignores carry output
- Implements Modular Arithmetic
  \[ s = \text{UAdd}_w(u, v) = u + v \mod 2^w \]
Visualizing Unsigned Addition

- Wraps Around
  - If true sum $\geq 2^w$
  - At most once

![Diagram showing UAdd operation]

Overflow
Two’s Complement Addition

Operands: $w$ bits

True Sum: $w+1$ bits

Discard Carry: $w$ bits

- $\text{TAdd and UAdd have Identical Bit-Level Behavior}$
Visualizing 2’s Complement Addition

• Example Values
  • 4-bit two’s comp
  • Range from -8 to +7

• Wraps Around
  • If sum $\geq 2^{w-1}$: Positive Overflow
    • Adding two positive numbers
      • Answer should be positive
      • Becomes negative
  • If sum $< -2^{w-1}$: Negative Overflow
    • Adding two negative numbers
      • Answer should be negative
      • Becomes positive

\[
TAdd_4(u, v)
\]
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
  • Two’s complement min (negative): Up to $2w-1$ bits
  • Two’s complement max (positive): Up to $2w$ bits, but only for $(TMin_w)^2$

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

Operands: $w$ bits

True Product: $2 \times w$ bits

Discard $w$ bits: $w$ bits

• Standard Multiplication Function
  • Ignores high order $w$ bits

• Implements Modular Arithmetic

\[
UMult_w(u, v) = u \cdot v \mod 2^w
\]
Signed Multiplication in C

• **Standard Multiplication Function**
  • Ignores high order \( w \) bits
  • Some of which are different for signed vs. unsigned multiplication
  • Lower bits are the same

Operands: \( w \) bits

True Product: \( 2^w \) bits

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

\[
\begin{array}{c}
\text{Operands: } w \text{ bits} \\
\text{True Product: } 2^w \text{ bits} \\
\text{Discard } w \text{ bits: } w \text{ bits}
\end{array}
\]

\[
\begin{array}{c}
\text{TMult}_w(u, v)
\end{array}
\]
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 \( \geq \) data size

<table>
<thead>
<tr>
<th>Argument ( x )</th>
<th>( 01100010 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \ll \ 3 )</td>
<td>( 00010000 )</td>
</tr>
<tr>
<td>Log. ( \gg ) 2</td>
<td>( 00011000 )</td>
</tr>
<tr>
<td>Arith. ( \gg ) 2</td>
<td>( 00011000 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument ( x )</th>
<th>( 10100010 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \ll \ 3 )</td>
<td>( 00010000 )</td>
</tr>
<tr>
<td>Log. ( \gg ) 2</td>
<td>( 00101000 )</td>
</tr>
<tr>
<td>Arith. ( \gg ) 2</td>
<td>( 11101000 )</td>
</tr>
</tbody>
</table>
Power-of-2 Multiply with Shift

• Operation
  • $u \ll k$ gives $u \times 2^k$
  • Both signed and unsigned

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

• Quotient of Unsigned by Power of 2
  • \( u \gg k \) gives \( \lfloor u / 2^k \rfloor \)
  • Uses logical right shift
  • Rounds towards zero (truncates decimal part)

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>15213</td>
<td>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>x &gt;&gt; 1</td>
<td>7606.5</td>
<td>1D B6</td>
<td>00011101 10110110</td>
</tr>
<tr>
<td>x &gt;&gt; 4</td>
<td>950.8125</td>
<td>03 B6</td>
<td>00000011 10110110</td>
</tr>
<tr>
<td>x &gt;&gt; 8</td>
<td>59.4257813</td>
<td>00 3B</td>
<td>00000000 00111011</td>
</tr>
</tbody>
</table>
Signed Power-of-2 Divide with Shift

- Quotient of signed by Power of 2
  - \( u \gg k \) gives \( \lfloor u / 2^k \rfloor \) (greatest integer less than)
  - Uses arithmetic right shift
  - Rounds down

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>-12,340.0</td>
<td>11001111 11001100</td>
</tr>
<tr>
<td>( x \gg 1 )</td>
<td>-6,170.0</td>
<td>11100111 11100110</td>
</tr>
<tr>
<td>( x \gg 4 )</td>
<td>-771.25</td>
<td>11111100 11111100</td>
</tr>
<tr>
<td>( x \gg 8 )</td>
<td>-48.203125</td>
<td>11111111 11001111</td>
</tr>
<tr>
<td>( X \gg 14 )</td>
<td>-0.75317382</td>
<td>11111111 11111111</td>
</tr>
</tbody>
</table>
Arithmetic: Basic Rules

• Addition:
  • Unsigned/signed: Normal addition followed by truncate
    • same operation on bit level
  • Unsigned: addition mod $2^w$
    • Mathematical addition + possible subtraction of $2^w$
  • Signed: modified addition mod $2^w$ (result in proper range)
    • Mathematical addition + possible addition or subtraction of $2^w$

• Multiplication:
  • Unsigned/signed: Normal multiplication followed by truncate, same operation on bit level
  • Unsigned: multiplication mod $2^w$
  • Signed: modified multiplication mod $2^w$ (result in proper range)

• Shifting:
  • Multiplying/Dividing by powers of 2
  • Logical right shift: shift in 0
  • Arithmetic right shift: shift in the sign bit