Integer Arithmetic and Undefined Behavior in C

Brad Karp
UCL Computer Science

CS 3007
23rd January 2018

(lecture notes derived from material from Eddie Kohler, John Regehr, Phil Gibbons, Randy Bryant, and Dave O’Hallaron)
Outline: Integer Arithmetic and Undefined Behavior in C

- C primitive data types
- C integer arithmetic
  - Unsigned and signed (two’s complement) representations
  - Maximum and minimum values; conversions and casts
  - Perils of C integer arithmetic, unsigned and especially signed
- Undefined behavior (UB) in C
  - As defined in the C99 language standard
  - Consequences for program behavior
  - Consequences for compiler and optimizer behavior
  - Examples
  - Prevalence of UB in real-world Linux application-level code
- Recommendations for defensive programming in C
## 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>

*signed (default) and unsigned variants*
Portables C types with Fixed Sizes

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>all archs</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{u}int8_t</code></td>
<td>1</td>
</tr>
<tr>
<td><code>{u}int16_t</code></td>
<td>2</td>
</tr>
<tr>
<td><code>{u}int32_t</code></td>
<td>4</td>
</tr>
<tr>
<td><code>{u}int64_t</code></td>
<td>8</td>
</tr>
<tr>
<td><code>uintptr_t</code></td>
<td>4 or 8</td>
</tr>
</tbody>
</table>

Type definitions available in `#include <stdint.h>`
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** (on which more shortly...)
  - Shift amount < 0 or ≥ word size

<table>
<thead>
<tr>
<th>Argument ( x )</th>
<th>( 01100010 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \ll 3 )</td>
<td></td>
</tr>
<tr>
<td>( \text{Log.} \gg 2 )</td>
<td>( 00010 )</td>
</tr>
<tr>
<td>( \text{Arith.} \gg 2 )</td>
<td>( 000 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument ( x )</th>
<th>( 10100010 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \ll 3 )</td>
<td></td>
</tr>
<tr>
<td>( \text{Log.} \gg 2 )</td>
<td>( 00010 )</td>
</tr>
<tr>
<td>( \text{Arith.} \gg 2 )</td>
<td>( 00 )</td>
</tr>
</tbody>
</table>
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
  - Undefined Behavior (on which more shortly...)
    - Shift amount $< 0$ or $\geq$ word size

- **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

<table>
<thead>
<tr>
<th>Argument $x$</th>
<th>01100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\ll 3$</td>
<td>00010</td>
</tr>
<tr>
<td>Log. $\gg 2$</td>
<td></td>
</tr>
<tr>
<td>Arith. $\gg 2$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument $x$</th>
<th>10100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\ll 3$</td>
<td></td>
</tr>
<tr>
<td>Log. $\gg 2$</td>
<td></td>
</tr>
<tr>
<td>Arith. $\gg 2$</td>
<td></td>
</tr>
</tbody>
</table>
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** (on which more shortly...)
  - Shift amount < 0 or \( \geq \) word size

---

<table>
<thead>
<tr>
<th>Argument</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></td>
</tr>
<tr>
<td>Arith. ( &gt;&gt; 2 )</td>
<td></td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Argument</th>
<th>10100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>( &lt;&lt; 3 )</td>
<td></td>
</tr>
<tr>
<td>Log. ( &gt;&gt; 2 )</td>
<td></td>
</tr>
<tr>
<td>Arith. ( &gt;&gt; 2 )</td>
<td></td>
</tr>
</tbody>
</table>
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 (on which more shortly...)**
  - Shift amount < 0 or ≥ word size
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 (on which more shortly...)**
  - Shift amount $< 0$ or $\geq$ word size
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 (on which more shortly...)**
  - Shift amount < 0 or ≥ word size
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 (on which more shortly...)**
  - Shift amount < 0 or ≥ word size

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

<table>
<thead>
<tr>
<th>Argument $x$</th>
<th>10100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x &lt;&lt; 3$</td>
<td></td>
</tr>
<tr>
<td>Log. $x &gt;&gt; 2$</td>
<td>00011000</td>
</tr>
<tr>
<td>Arith. $x &gt;&gt; 2$</td>
<td>00011000</td>
</tr>
</tbody>
</table>
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 (on which more shortly...)**
  - Shift amount < 0 or ≥ word size
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 (on which more shortly...)**
  - Shift amount < 0 or ≥ word size
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** (on which more shortly...)
  - Shift amount $< 0$ or $\geq$ word size

### Examples

<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><strong>Log. $&gt;&gt; 2$</strong></td>
<td>$00011000$</td>
</tr>
<tr>
<td><strong>Arith. $&gt;&gt; 2$</strong></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><strong>Log. $&gt;&gt; 2$</strong></td>
<td>$101000$</td>
</tr>
<tr>
<td><strong>Arith. $&gt;&gt; 2$</strong></td>
<td>$101000$</td>
</tr>
</tbody>
</table>
# 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 (on which more shortly...)**
  - Shift amount < 0 or ≥ 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>00101000</td>
</tr>
</tbody>
</table>
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 (on which more shortly...)**
  - Shift amount < 0 or \( \geq \) word size

<table>
<thead>
<tr>
<th>Argument ( x )</th>
<th>( 01100010 )</th>
<th>( 10100010 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( &lt;&lt; 3 )</td>
<td>( 00010000 )</td>
<td>( 00010000 )</td>
</tr>
<tr>
<td>Log. ( &gt;&gt; 2 )</td>
<td>( 00011000 )</td>
<td>( 00101000 )</td>
</tr>
<tr>
<td>Arith. ( &gt;&gt; 2 )</td>
<td>( 00011000 )</td>
<td>( 101000 )</td>
</tr>
</tbody>
</table>
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 (on which more shortly...)**
- Shift amount < 0 or \( \geq \) word size
Integer Numeric Ranges
Integer Numeric Ranges

- Unsigned Values
  - $UMin = 0$
    - 000...0
  - $UMax = 2^w - 1$
    - 111...1
Integer 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
  - negative 1
    - 111...1
## Integer 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
- negative 1
  111...1

### Values for word size $W = 16$

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>$U_{\text{Max}}$</td>
<td>65535</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>$T_{\text{Max}}$</td>
<td>32767</td>
<td>7F FF</td>
<td>01111111 11111111</td>
</tr>
<tr>
<td>$T_{\text{Min}}$</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>
Integer Ranges 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**
  - \(|T_{\text{Min}}| = T_{\text{Max}} + 1\)
    - 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`, etc.
  - Values platform specific
  - Also, in `<stdint.h>`
    - `INT\{8,16,32,64\}_{MIN,MAX}`
    - `UINT\{8,16,32,64\}_{MAX}`
Unsigned & Signed Integer 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>
### Mapping Signed $\leftrightarrow$ Unsigned (W=4)

<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 (W=4)

<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 (W=4)

<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>
Conversion Visualized

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

2’s Complement Range

TMax

0

-1

-2

TMin

UMax

UMax – 1

TMax

TMax + 1

Unsigned Range
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 just **takes bits as-is and reinterprets their value using the other representation**
    ```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></td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>0U</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td>-2147483647-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2147483647U</td>
<td>-2147483647-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(unsigned)-1</td>
<td>-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td>2147483648U</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td>(int) 2147483648U</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 \(<\), \(\geq\), \(==\), \(<=\), \(>=\)
  - Examples for \(W = 32\):
    - \(TMIN = -2,147,483,648\), \(TMAX = 2,147,483,647\)

<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>
Unsigned vs. Signed: Easy to Make Mistakes

```c
unsigned i;
for (i = cnt-2; i >= 0; i--)
    a[i] += a[i+1];
```
Unsigned vs. Signed: 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)
      ...
  ```
Safely Using an unsigned Loop Index

Broken:

```c
unsigned i;
for (i = cnt-2; i >= 0; i--)
    a[i] += a[i+1];
```
Safely Using an unsigned Loop Index

- Broken:

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

- Safe:

  unsigned i;
  for (i = cnt-2; i < cnt; i--)
      a[i] += a[i+1];
Safely Using an unsigned Loop Index

■ Broken:

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

■ Safe:

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

■ Why is the latter safe?
  - because 0U - 1 == UINT_MAX in unsigned arithmetic!
Summary: Casting Signed ↔ Unsigned

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

- Beware expressions mixing `signed` and `unsigned int`
  - `int` implicitly cast to `unsigned`!!
Stepping Back: Undefined Behavior in C

- Many operations in C offer **defined behavior**, where (e.g.,) the C99 standard specifies exactly what result the C abstract machine will produce.
- Some (alas, fairly many) operations in C are explicitly noted by the C99 standard as **undefined behavior**.
- In the words of the C99 standard (ISO be praised):
Stepping Back: Undefined Behavior in C

- Many operations in C offer defined behavior, where (e.g.,) the C99 standard specifies exactly what result the C abstract machine will produce.
- Some (alas, fairly many) operations in C are explicitly noted by the C99 standard as undefined behavior.
- In the words of the C99 standard (ISO be praised):
  - **undefined behavior**: behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements.

  **NOTE** Possible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message).

  **EXAMPLE** An example of undefined behavior is the behavior on integer overflow.
Stepping Back: Undefined Behavior in C

- Many operations in C offer defined behavior, where (e.g.,) the C99 standard specifies exactly what result the C abstract machine will produce.

- Some (alas, fairly many) operations in C are explicitly noted by the C99 standard as undefined behavior.

- In the words of the C99 standard (ISO be praised):

  **undefined behavior**: behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements.

  NOTE: Possible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message).

  EXAMPLE: An example of undefined behavior is the behavior on integer overflow.

- **199 undefined behaviors** in C99!
Broad Categories of UB in C (most common, but not exhaustive!)

- Programmer error involving pointers and memory allocation/deallocation
  - walking past the end of an array; use-after-free(); double free(); memory leaks; &c.

- Integer overflow
  - in unsigned arithmetic, exceeding UINT_MAX is defined; wrap to zero
  - similarly for subtracting 0U – 1U; wrap to UINT_MAX
  - in signed arithmetic, though, overflow is undefined behavior
  - Why?
    - Historically, different CPU architectures represented signed integers differently, and yielded different results upon overflow in signed integer arithmetic.
  - So why is unsigned overflow defined?
    - Because unsigned representations and overflow results didn’t/don’t vary across CPU architectures!
Wait. Why *deliberately* design a language to include UB???

- **Performance**
  - possible for compiler to generate instructions that explicitly detect some UB cases and generate run-time errors, and render such behavior defined
  - e.g., check shift less than word length; check for overflow on every signed arithmetic operation; bounds checks on array accesses; &c.
  - cost:
    - “30-50% performance reduction” on tight loops [Regehr, others]
    - optimized Rust 15-20% slower than optimized C for loops that do signed integer arithmetic

- **Simplifies compiler design and implementation**
  - Compiler needn’t reason about results generated by complex “corner cases”; enjoin programmer from writing such code
And what are the drawbacks of a language with 199 UBs?

- More difficult to write correct programs
  - Programmers often unaware of UBs, unintentionally write code that exercises them
  - Can think of such code as “buggy”
  - ...but the compiler can behave in surprising ways when it encounters code identifiable as exercising UB
  - ...because the C99 spec says that anything can happen once a program exercises UB (silent incorrect results; expected results; termination; etc.), C compilers assume programmers don’t write code that exercises UB

- Trend in recent years: compilers may discard code when they detect UB
  - Again, C99 says that incorrect results are fine once UB invoked...so discarding code may be totally consistent with C99
  - Less code means faster programs; perverse incentive
Integer Undefined Behavior in C

- Unsigned arithmetic generally defined; one exception is $x / 0$ (also UB for signed)
- Signed arithmetic UB:
  - Overflow for addition, subtraction, negation
    - Compiler assumes machine result is valid (it may not be)
  - Overflow for multiplication
  - Overflow for integer division and remainder (mod)
    - Consider INT_MIN / -1
    - Remember, one more negative value in two’s complement than there are positive values!
- Remember: the compiler may assume that no program ever produces any of the above results!
Example: The Case of the Dropped Assertion

```c
int check_signed_increment(int x) {
    assert(x + 1 > x);
    return x + 1;
}

int main(int argc, char** argv) {
    int x = strtol(argv[1], NULL, 0);
    int x_incr = check_signed_increment(x);
    printf("%d + 1 = %d\n", x, x_incr);
}
```

- Compile without optimization, invoke with 0x7fffffffff:
  - result is crash (assertion failure), as overflow from 0x7fffffffff to 0x80000000 (INT_MIN) when computing `x + 1`
- Compile with optimization, invoke with 0x7fffffffff:
  - Result is output "2147483647 + 1 = -2147483648"!
  - Assertion code dropped by compiler! Why: `x + 1 > x` always true, if signed overflow cannot be executed by programmer!
Example: Programmer Errs; Compiler Takes Creative License

```c
struct tun_struct *tun = ...;
struct sock *sk = tun->sk;
if (!tun)
    return POLLERR;
/* write to address based on tun */
```

- Real code from Linux kernel (CVE-2009-1897)
- Programmer dereferences tun before null pointer check
- Compiler notes dereference in second line, concludes “tun != NULL; if it were, programmer would have implemented undefined behavior”
- Compiler does not emit any code for if statement!
- Result: exploit that allows attacker to elevate privilege in Linux (because of later use of `NULL`)
Example: UB Throwdown, Postgres Devs vs. MIT PhD Student

```c
int64_t arg1 = ...;         // user input
int64_t arg2 = ...;         // user input
if (arg2 == 0)              // prevent division by zero
    ereport(ERROR, ...);
int64_t result = arg1 / arg2; // division of signed 64-bit ints
if (arg2 == -1 && arg1 < 0 && result <= 0)  // UB!
    ereport(ERROR, ...);
```

- **Context:** check for signed division overflow in POSTgres SQL database; **division precedes check**

- **Programmer error:** incorrect expectation that integer division overflow of \(-2^{63} / -1\) in C “wraps” from \(2^{63}\) (unrepresentable in signed 64-bit integer) to \(-2^{63}\)
  - check \(\text{result} <= 0\) intended to catch this
  - Java behaves that way: defines signed integer division overflow to “wrap”
  - but in C, all signed overflow is UB
Example: UB Throwdown, Postgres Devs vs. MIT PhD Student

```c
int64_t arg1 = ...; // user input
int64_t arg2 = ...; // user input
if (arg2 == 0)        // prevent division by zero
    ereport(ERROR, ...);
int64_t result = arg1 / arg2; // division of signed 64-bit ints
if (arg2 == -1 && arg1 < 0 && result <= 0)    // ALWAYS FALSE (!)
    ereport(ERROR, ...);
```

- Compiler notes computation of `arg1 / arg2`, both signed quantities...
- ...and concludes expression in final `if` must always be false: when `arg2 == -1` and `arg1 < 0`, `result <= 0` implies the prior division was UB, which “cannot be”
- x86-64 `idivq` instruction traps upon overflow
- Consequence: user can issue SQL query to database that provokes integer divide overflow, crashing database
Example: UB Throwdown, Postgres Devs vs. MIT PhD Student

```
int64_t arg1 = ...;    // user input
int64_t arg2 = ...;    // user input
if (arg2 == 0)          // prevent division by zero
  ereport(ERROR, ...);
int64_t result = arg1 / arg2; // division of signed 64-bit ints
if (arg2 == -1 && arg1 < 0 && result <= 0)  // ALWAYS FALSE (!)
  ereport(ERROR, ...);

MIT PhD student working on tools to detect UB in C finds above bug, submits patch to POSTgres team:

```
int64_t arg1 = ...;    // user input
int64_t arg2 = ...;    // user input
if (arg2 == 0)          // prevent division by zero
  ereport(ERROR, ...);
int64_t result;
if (arg1 == INT64_MIN && arg2 == -1) // check for overflow first
  ereport(ERROR, ...);  // report error on UB
else
  result = arg1 / arg2; // only divide if safe
Example: UB Throwdown, Postgres Devs vs. MIT PhD Student

Postgres team thanks reporter for report, but rejects proposed fix in favor of its own design:

```c
int64_t arg1 = ...; // user input
int64_t arg2 = ...; // user input
int64_t result;
if (arg1 != 0 && (-arg1 < 0) == (arg1 < 0)) // @!$%!!
    ereport(ERROR, ...); // report error on UB
else
    result = arg1 / arg2; // only divide if safe
```

Alas, Postgres team’s fix still invokes UB!

- \((-\text{arg1} < 0)\) attempts to detect whether \text{arg1} is \(-2^{63}\), but does so by (as in the original broken code) assuming that overflow when negating this value “wraps” to a negative value...
- ...when computing \(2^{63}\) in signed arithmetic is UB, so compiler concludes that \((-\text{arg1} < 0)\) is always false
UB That Causes “Unstable” Code: It’s Out There

<table>
<thead>
<tr>
<th>UB condition</th>
<th># reports</th>
<th># packages</th>
</tr>
</thead>
<tbody>
<tr>
<td>null pointer dereference</td>
<td>59,230</td>
<td>2,800</td>
</tr>
<tr>
<td>buffer overflow</td>
<td>5,795</td>
<td>1,064</td>
</tr>
<tr>
<td>signed integer overflow</td>
<td>4,364</td>
<td>780</td>
</tr>
<tr>
<td>pointer overflow</td>
<td>3,680</td>
<td>614</td>
</tr>
<tr>
<td>oversized shift</td>
<td>594</td>
<td>193</td>
</tr>
<tr>
<td>aliasing</td>
<td>330</td>
<td>70</td>
</tr>
<tr>
<td>overlapping memory copy</td>
<td>227</td>
<td>47</td>
</tr>
<tr>
<td>division by zero</td>
<td>226</td>
<td>95</td>
</tr>
<tr>
<td>use after free</td>
<td>156</td>
<td>79</td>
</tr>
<tr>
<td>other libc (cttz, ctlz)</td>
<td>132</td>
<td>7</td>
</tr>
<tr>
<td>absolute value overflow</td>
<td>86</td>
<td>23</td>
</tr>
<tr>
<td>use after realloc</td>
<td>22</td>
<td>10</td>
</tr>
</tbody>
</table>

- Universe of Debian wheezy packages
- [Wang, Zeldovich et al., SOSP 2013]
More on Undefined Behavior in C

- John Regehr’s (Utah) humbling “Quiz About Integers in C”:

- Regehr’s 3-part series of blog posts on UB from 2010:
  - https://blog.regehr.org/archives/213
  - https://blog.regehr.org/archives/226
  - https://blog.regehr.org/archives/232

- Regehr et al. on the incidence of integer overflow in real, widely used C and C++ code:

- Regehr and Cuoq’s blog post on the state of UB in 2017, including great sanitizer tools to detect UB in your code:
  - https://blog.regehr.org/archives/1520

- Wang, Zeldovich et al. (MIT) on overzealous C optimizers and UB, and a tool to detect silently dropped code (2013):