Topics

- **Numeric Encodings**
  - Unsigned & Two’s complement

- **Programming Implications**
  - C promotion rules
  - Consequences of overflow

- **Basic operations**
  - Addition, negation, multiplication
Notation

Word Size

- $w$
  - 32 for most machines
  - 64 for Alpha and next generation of high end machines
  - 8 or 16 for many signal processing applications

Integers

- Lower case
- E.g., $x$, $y$, $z$

Bit Vectors

- Upper Case
- E.g., $X$, $Y$, $Z$
- Write individual bits as integers with value 0 or 1
- E.g., $X = x_{w-1}, x_{w-2}, \ldots, x_0$
  - Big-Endian
Encodings

Unsigned

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

Ranges

- \( UMin = 0 \)

000…0

0x00…0

- \( UMax = 2^w - 1 \)

111…1

0xFF…F

Two’s Complement

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

Ranges

- \( TMin = -2^{w-1} \)

100…0

0x80…0

- \( TMax = 2^{w-1} - 1 \)

011…1

0x7F…F

Other Values

- Zero

000…0

- Minus 1

111…1

0xFF…F
Real-Life Values

W = 32

- UMax +4,294,967,295
- Tmax + 2,147,483,647
- Tmin – 2,147,483,648

W = 64

- UMax +18,446,744,073,709,551,615
- Tmax +9,223,372,036,854,775,807
- Tmin –9,223,372,036,854,775,808

C Programming

- #include <limits.h>
- Declares constants:
  - ULONG_MAX
  - LONG_MAX
  - LONG_MIN
Encoding Example

Observe

- Same encodings for nonnegative values
- Assymmetric range
  \( T_{Max} \neq -T_{min} \)

Each Encoding is *Bijection*

- Every bit pattern represents unique integer value
- Each representable integer has unique bit encoding

\[ \implies \text{Can Invert Mappings} \]

\[ U_2B(x) = B2U^{-1}(x) \]
  - Bit pattern for unsigned integer
\[ T_2B(x) = B2T^{-1}(x) \]
  - Bit pattern for two’s comp integer

<table>
<thead>
<tr>
<th>( \chi )</th>
<th>( \text{B2U}(\chi) )</th>
<th>( \text{B2T}(\chi) )</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>
Relation Between Unsigned & Two’s Comp.

Maintain Same Bit Pattern

\[
B2T(X) = \sum_{i=0}^{w-2} x_i \cdot 2^i - x_{w-1} 2^{w-1} \\
= \sum_{i=0}^{w-2} x_i \cdot 2^i + x_{w-1} 2^{w-1} - x_{w-1} 2^w \\
= \sum_{i=0}^{w-1} x_i \cdot 2^i - x_{w-1} 2^w \\
= B2U(X) - x_{w-1} 2^w
\]

\[
x' = \begin{cases} 
  x & x < 2^{w-1} \\
  x - 2^w & x \geq 2^{w-1}
\end{cases}
\]
From Unsigned to Two’s Complement

- \( U2T(x) \)
  
  \[
  U2T(x) = B2T(U2B(x)) = x - x_{w-1} 2^w
  \]

- What you get in C:
  
  ```c
  int u2t(unsigned x)
  {
    return (int) x;
  }
  ```

<table>
<thead>
<tr>
<th>( x )</th>
<th>( B2U(x) )</th>
<th>( B2T(x) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1010</td>
<td>10</td>
<td>-6</td>
</tr>
</tbody>
</table>

\(-16\)
From Two’s Complement to Unsigned

- \( T2U(x) \)
  
  \[ = B2U(T2B(x)) \]
  
  \[ = x + x_{w-1}2^w \]

- What you get in C:

  ```c
  int t2u(int x)
  {
    return (unsigned) x;
  }
  ```

<table>
<thead>
<tr>
<th>( x )</th>
<th>( B2U(x) )</th>
<th>( B2T(x) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1010</td>
<td>10</td>
<td>-6</td>
</tr>
</tbody>
</table>
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
  \[
  \begin{align*}
  \text{int } & \quad \text{tx, ty}; \\
  \text{unsigned } & \quad \text{ux, uy}; \\
  \text{tx} & \quad = \text{(int) ux}; \\
  \text{uy} & \quad = \text{(unsigned) ty};
  \end{align*}
\]
• Implicit casting also occurs via assignments and procedure calls
  \[
  \begin{align*}
  \text{tx} & \quad = \text{ux}; \\
  \text{uy} & \quad = \text{ty};
  \end{align*}
\]
Casting Surprises

Expression Evaluation

- If mix unsigned & signed in single expression, signed values implicitly cast to unsigned
- Including comparison operations <, >, ==, <=, >=

<table>
<thead>
<tr>
<th>Constant\textsubscript{1}</th>
<th>Constant\textsubscript{2}</th>
<th>Relation</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0\text{U}</td>
<td>==</td>
<td>unsigned</td>
</tr>
<tr>
<td>\text{-1}</td>
<td>0</td>
<td>&lt;</td>
<td>signed</td>
</tr>
<tr>
<td>\text{-1}</td>
<td>0\text{U}</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>\text{2147483647}</td>
<td>\text{-2147483648}</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>\text{2147483647}\text{U}</td>
<td>\text{-2147483648}</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>\text{-1}</td>
<td>\text{-2}</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>(unsigned) \text{-1}</td>
<td>\text{-2}</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>\text{2147483647}</td>
<td>\text{2147483648}\text{U}</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>\text{2147483647}</td>
<td>(int) \text{2147483648}\text{U}</td>
<td>?</td>
<td></td>
</tr>
</tbody>
</table>
Adder Circuit

Assume
- Carry Input = 0
- Ignore Carry Output

Observation
- Output at bit position $i$ depends only on inputs at positions 0 to $i$
Unsigned Addition

Standard Addition Function
- Ignores carry output

Implements Modular Arithmetic
\[ s = \text{UAdd}_w(u, v) = u + v \mod 2^w \]

\[
\text{UAdd}_w(u, v) = \begin{cases} 
    u + v & u + v < 2^w \\
    u + v - 2^w & u + v \geq 2^w 
\end{cases}
\]
Visualizing Integer Addition

Integer Addition

- 4-bit integers \( u \) and \( v \)
- Compute sum \( \text{Add}_4(u, v) \)
- Values increase linearly with \( u \) and \( v \)
- Forms planar surface
Visualizing Unsigned Addition

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

True Sum
\[ 2^{w+1} \]
\[ 2^w \]
\[ 0 \]

Modular Sum

UAdd$_4(u, v)$
Mathematical Properties

Modular Addition Forms an Abelian Group

- Closed under addition
  \[ 0 \leq \text{UAdd}_w(u, v) \leq 2^w - 1 \]

- Commutative
  \[ \text{UAdd}_w(u, v) = \text{UAdd}_w(v, u) \]

- Associative
  \[ \text{UAdd}_w(t, \text{UAdd}_w(u, v)) = \text{UAdd}_w(\text{UAdd}_w(t, u), v) \]

- 0 is additive identity
  \[ \text{UAdd}_w(u, 0) = u \]

- Every element has additive inverse
  - Let \[ \text{UComp}_w(u) = 2^w - u \]
  \[ \text{UAdd}_w(u, \text{UComp}_w(u)) = 0 \]
Detecting Unsigned Overflow

Task

- Given \( s = \text{UAdd}_w(u, v) \)
- Determine if \( s = u + v \)

Application

unsigned s, u, v;
\( s = u + v; \)

- Did addition overflow?

Claim

- Overflow iff \( s < u \)
  \( \text{ovf} = (s < u) \)
- Or symmetrically iff \( s < v \)

Proof

- Know that \( 0 \leq v < 2^w \)
- No overflow \( \implies s = u + v \geq u + 0 = u \)
- Overflow \( \implies s = u + v - 2^w < u + 0 = u \)
Two’s Complement Addition

Add Two’s Complement Numbers with Conventional Adder

- E.g., signed addition in C:
  ```c
  int s, u, v;
  s = u + v;
  ```
- Ignores carry output
  \[ s = \text{TAdd}_w(u, v) \]
- What is the behavior of this operation?
Characterizing TAdd

Functionality

- True sum requires $w+1$ bits
- Drop off MSB
- Treat remaining bits as 2's comp. integer

$$TAdd_w(u,v) = \begin{cases} 
  u + v + 2^{w-1} & u + v < TMin_w \\
  u + v & TMin_w \leq u + v \leq TMax_w \\
  u + v - 2^{w-1} & TMax_w < u + v
\end{cases}$$
Visualizing 2’s Comp. Addition

Values

- 4-bit two’s comp.
- Range from -8 to +7

Wraps Around

- If sum $\geq 2^{w-1}$
  - At most once
  - Becomes negative
- If sum $< -2^{w-1}$
  - At most once
  - Becomes positive

\[ TAdd_4(u, v) \]
Detecting 2’s Comp. Overflow

Task
• Given $s = TAdd_w(u, v)$
• Determine if $s = u + v$

Application

```c
int s, u, v;
s = u + v;
```

Claim
• Overflow iff either:
  - $-u, v < 0, s \geq 0$ (NegOver)
  - $-u, v \geq 0, s < 0$ (PosOver)

```c
ovf = (u<0 == v<0) && (u<0 != s<0);
```

Proof
• Easy to see that if $u \geq 0$ and $v < 0$, then $TMin_w \leq u + v \leq TMax_w$
• Symmetrically if $u < 0$ and $v \geq 0$
• Other cases from analysis of $TAdd$
Mathematical Properties of TAdd

Isomorphic Algebra to UAdd

- \( \text{TAdd}_w(u, v) = \text{U2T}(\text{UAdd}_w(\text{T2U}(u), \text{T2U}(v))) \)
  - Since both have identical bit patterns

Two’s Complement Under TAdd Forms a Group

- Closed, Commutative, Associative, 0 is additive identity
- Every element has additive inverse
  - Let \( \text{TComp}_w(u) = \text{U2T}(\text{UComp}_w(\text{T2U}(u))) \)
  - \( \text{TAdd}_w(u, \text{TComp}_w(u)) = 0 \)

\[
\text{TComp}_w(u) = \begin{cases} 
-u & \text{if } u \neq \text{TMin}_w \\
?? & \text{if } u = \text{TMin}_w
\end{cases}
\]
Two’s Complement Negation

Mostly like Integer Negation
  - $T\text{Comp}(u) = -u$

$T\text{Min}$ is Special Case
  - $T\text{Comp}(T\text{Min}) = T\text{Min}$

Negation in C is $\text{NOT}$ True Negation
  - $m\text{x} = -x$
    - Really $m\text{x} = T\text{Comp}(x)$
    - But is a complement
      $x + -x == 0$
Negating with Complement & Increment

In C

\[ \sim x + 1 == -x \]

Complement

• Derive using property that for bit value \( x \):

\[ \bar{x} = 1 - x \]

\[
B2T_w (X) = \sum_{i=0}^{w-2} (1 - x_i) \cdot 2^i - (1 - x_{w-1}) \cdot 2^{w-1}
\]

\[
= \left( 2^{w-1} - 1 \right) + 2^{w-1} - B2T_w (X)
\]

\[ = -B2T_w (X) - 1 \]

Increment

\[
TAdd_w (B2T_w (X), 1) = \begin{cases} 
-B2T_w (X) & X \neq 100\ldots0 \\
TMin_w & X = 100\ldots0 \\
TComp_w (B2T_w (X)) & \text{otherwise}
\end{cases}
\]

\[ = TComp_w (B2T_w (X)) \]
## Comp. & Incr. Example

<table>
<thead>
<tr>
<th>$X$</th>
<th>$\text{B2T}(X)$</th>
<th>$\overline{X}$</th>
<th>$\text{B2T}(\overline{X})$</th>
<th>$\text{Add}(X, 1)$</th>
<th>$\text{B2T}(\cdot)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>1111</td>
<td>-1</td>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1110</td>
<td>-2</td>
<td>1111</td>
<td>-1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>1101</td>
<td>-3</td>
<td>1110</td>
<td>-2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>1100</td>
<td>-4</td>
<td>1101</td>
<td>-3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>1011</td>
<td>-5</td>
<td>1100</td>
<td>-4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>1010</td>
<td>-6</td>
<td>1011</td>
<td>-5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>1001</td>
<td>-7</td>
<td>1010</td>
<td>-6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>1000</td>
<td>-8</td>
<td>1001</td>
<td>-7</td>
</tr>
<tr>
<td>1000</td>
<td>-8</td>
<td>0111</td>
<td>7</td>
<td>1000</td>
<td>-8</td>
</tr>
<tr>
<td>1001</td>
<td>-7</td>
<td>0110</td>
<td>6</td>
<td>0111</td>
<td>7</td>
</tr>
<tr>
<td>1010</td>
<td>-6</td>
<td>0101</td>
<td>5</td>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>1011</td>
<td>-5</td>
<td>0100</td>
<td>4</td>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>1100</td>
<td>-4</td>
<td>0011</td>
<td>3</td>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>1101</td>
<td>-3</td>
<td>0010</td>
<td>2</td>
<td>0011</td>
<td>3</td>
</tr>
<tr>
<td>1110</td>
<td>-2</td>
<td>0001</td>
<td>1</td>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>1111</td>
<td>-1</td>
<td>0000</td>
<td>0</td>
<td>0001</td>
<td>1</td>
</tr>
</tbody>
</table>
Complement Upper Bits

Task

• Given bit vector $X$
• Negate it without using addition

Rule

• Let $k$ be minimum value such that $x_k = 1$
• I.e., $X = x_{w-1}, x_{w-2}, \ldots, x_{k+1}, 1, 0, \ldots, 0$
• Compute $\text{Negate}(X) = x_{w-1}, x_{w-2}, \ldots, x_{k+1}, 1, 0, \ldots, 0$

Justification

• Effect of complement:

$$\bar{X} = \bar{x}_{w-1}, \bar{x}_{w-2}, \ldots, \bar{x}_{k+1}, 0, 1, \ldots, 1$$

• Effect of increment:

$$\text{Add}(\bar{X}, 1) = \bar{x}_{w-1}, \bar{x}_{w-2}, \ldots, \bar{x}_{k+1}, 1, 0, \ldots, 0$$

Observation

• Low order bit of $X$ and $-X$ identical
# Upper Bits Complement Example

<table>
<thead>
<tr>
<th>$X$</th>
<th>B2T($X$)</th>
<th>UC($X$)</th>
<th>B2T($\cdot$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1111</td>
<td>-1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>1110</td>
<td>-2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>1101</td>
<td>-3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>1100</td>
<td>-4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>1011</td>
<td>-5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>1010</td>
<td>-6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>1001</td>
<td>-7</td>
</tr>
<tr>
<td>1000</td>
<td>-8</td>
<td>1000</td>
<td>-8</td>
</tr>
<tr>
<td>1001</td>
<td>-7</td>
<td>0111</td>
<td>7</td>
</tr>
<tr>
<td>1010</td>
<td>-6</td>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>1011</td>
<td>-5</td>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>1100</td>
<td>-4</td>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>1101</td>
<td>-3</td>
<td>0011</td>
<td>3</td>
</tr>
<tr>
<td>1110</td>
<td>-2</td>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>1111</td>
<td>-1</td>
<td>0001</td>
<td>1</td>
</tr>
</tbody>
</table>
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:
• \( X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0 \)

\( k \) copies of MSB

Correctness Argument
• Induct on \( k \), i.e., that extending by single bit maintains value
• Key observation: \( -2^{w-1} = -2^w + 2^{w-1} \)

\[
\begin{align*}
B2T(X') &= -x_{w-1} \cdot 2^w + x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i \\
&= x_{w-1} \cdot (-2^w + 2^{w-1}) + \sum_{i=0}^{w-2} x_i \cdot 2^i \\
&= x_{w-1} \cdot (-2^{w-1}) + \sum_{i=0}^{w-2} x_i \cdot 2^i \\
&= B2T(X)
\end{align*}
\]
Multiplication

Computing Exact Product of $w$-bit numbers $x, y$

- Either signed or unsigned

Ranges

- **Unsigned**: $0 \leq x \cdot y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
  - Up to $2^w$ bits
- **Two’s complement min**: $x \cdot y \geq (-2^{w-1})^2(2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$
  - Up to $2^{w-1}$ bits
- **Two’s complement max**: $x \cdot y \leq (-2^{w-1})^2 = 2^{2w-2}$
  - Up to $2^w$ bits, but only for $TMin_w$

Maintaining Exact Results

- Would need to keep expanding word size with each product computed
- Done in software by “arbitrary precision” arithmetic packages
- Also implemented in Lisp, ML, and other “advanced” languages
Multiplication In C

Operation

```c
int x, y;
int p = x * y;
```

- Compute exact product of two \( w \)-bit numbers \( x, y \)
- Truncate result to \( w \)-bit number \( p = \text{TMult}_w(x, y) \)

Unsigned Counterpart

```c
unsigned ux = (unsigned) x;
unsigned uy = (unsigned) y;
unsigned up = ux * uy
```

- Truncates product to \( w \)-bit number \( up = \text{UMult}_w(ux, uy) \)
- Simply modular arithmetic
  \[ up = ux \cdot uy \mod 2^w \]

Relation

- Claim that \( up == (unsigned) p \)
Multiplication Analysis

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

\[ ax \quad sx \]

\[ x = ax - 2^{w-1} \cdot sx \]

\[ ux = ax + 2^{w-1} \cdot sx \]

\[ x \cdot y = ax \cdot ay - 2^{w-1} (sx \cdot ay + sy \cdot ax) + 2^{2w-2} sx \cdot sy \]

\[ ux \cdot uy = ax \cdot ay + 2^{w-1} (sx \cdot ay + sy \cdot ax) + 2^{2w-2} sx \cdot sy \]

\[ \alpha \text{ (2w-2 bits)} \quad \beta \text{ (w bits)} \quad \gamma \text{ (1 bit)} \]
Multiplication Analysis (cont.)

Signed ($w = 5$)

\[
\begin{align*}
\alpha & \quad \text{[bit patterns]} \\
-\beta & \quad \text{[bit patterns]} \\
+\gamma & \quad \text{[bit patterns]} \\
\hline \\
\quad & \quad \text{[bit patterns]} \\
\quad & \quad \text{[bit patterns]} \\
\quad & \quad \text{[bit patterns]} \\
\quad & \quad \text{[bit patterns]} \\
\quad & \quad \text{[bit patterns]} \\
\quad & \quad \text{[bit patterns]} \\
\end{align*}
\]

\[
\begin{align*}
p & \quad \text{[bit patterns]} \\
\end{align*}
\]

Unsigned

\[
\begin{align*}
\alpha & \quad \text{[bit patterns]} \\
+\beta & \quad \text{[bit patterns]} \\
+\gamma & \quad \text{[bit patterns]} \\
\end{align*}
\]

\[
\begin{align*}
\quad & \quad \text{[bit patterns]} \\
\quad & \quad \text{[bit patterns]} \\
\quad & \quad \text{[bit patterns]} \\
\quad & \quad \text{[bit patterns]} \\
\quad & \quad \text{[bit patterns]} \\
\quad & \quad \text{[bit patterns]} \\
\quad & \quad \text{[bit patterns]} \\
\quad & \quad \text{[bit patterns]} \\
\end{align*}
\]

Observe

- Only LSB of $\beta$ affects values of $p$ and $up$
- LSB of $\beta$ and $-\beta$ are the same
- $p$ and $up$ have matching bit patterns!
Algebraic Properties of Unsigned

Unsigned Multiplication with Addition Forms
Commutative Ring

- Addition is commutative group
- Closed under multiplication
  \[ 0 \leq \text{UMult}_w(u, v) \leq 2^w - 1 \]
- Multiplication Commutative
  \[ \text{UMult}_w(u, v) = \text{UMult}_w(v, u) \]
- Multiplication is Associative
  \[ \text{UMult}_w(t, \text{UMult}_w(u, v)) = \text{UMult}_w(\text{UMult}_w(t, u), v) \]
- 1 is multiplicative identity
  \[ \text{UMult}_w(u, 1) = u \]
- Multiplication distributes over addition
  \[ \text{UMult}_w(t, \text{UAdd}_w(u, v)) = \text{UAdd}_w(\text{UMult}_w(t, u), \text{UMult}_w(t, v)) \]
Algebraic Properties of Two’s Comp.

Isomorphic Algebras

- Unsigned multiplication and addition
  - Truncating to $w$ bits

- Two’s complement multiplication and addition
  - Truncating to $w$ bits

Both Form Rings

- Isomorphic to ring of integers mod $2^w$
- NOT isomorphic to ring of integers
C Puzzles

• Taken from Exam #2, CS 347, Spring ‘97
• Assume machine with 32 bit word size, two’s complement integers
• For each of the following C expressions, either:
  – Argue that is true for all argument values
  – Give example where not true

Initialization

```c
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```

- \( x < 0 \implies (x \times 2 < 0) \)
- \( ux \geq 0 \)
- \( x \& 7 == 7 \implies (x \ll 30) < 0 \)
- \( ux > -1 \)
- \( x > y \implies -x < -y \)
- \( x \times x \geq 0 \)
- \( x > 0 && y > 0 \implies x + y > 0 \)
- \( x \geq 0 \implies -x \leq 0 \)