Integers
Sep 4, 2001

Topics
• Numeric Encodings
  – Unsigned & Two's complement
• Programming Implications
  – C promotion rules
• Basic operations
  – Addition, negation, multiplication
• Programming Implications
  – Consequences of overflow
  – Using shifts to perform power-of-2 multiply/divide

C Puzzles
• Taken from Exam #2, CS 347, Spring '97
• Assume machine with 32 bit word size, two’s complement integers
  – Argue that is true for all argument values
  – Give example where not true

Initialization
• \( x < 0 \) \( \Rightarrow \) \((x*2) < 0\)
• \( ux >= 0 \)
• \( x & 7 == 7 \) \( \Rightarrow \) \((x<<30) < 0\)
• \( ux > -1 \)
• \( x > y \) \( \Rightarrow \) \(-x < -y\)
• \( x * x >= 0 \)
• \( x > 0 \ & \ y > 0 \) \( \Rightarrow \) \(x + y > 0\)
• \( x >= 0 \) \( \Rightarrow \) \(-x <= 0\)
• \( x <= 0 \) \( \Rightarrow \) \(-x >= 0\)

\[ \text{int x = foo();} \]
\[ \text{int y = bar();} \]
\[ \text{unsigned ux = x;} \]
\[ \text{unsigned uy = y;} \]

Encoding Integers

Unsigned

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

Two’s Complement

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

• 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>11000100 10010011</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>11000100 10010011</td>
</tr>
</tbody>
</table>

Sign Bit
• For 2’s complement, most significant bit indicates sign
  – 0 for nonnegative
  – 1 for negative
**Numeric Ranges**

**Unsigned Values**
- \( UMin = 0 \)
  
  000...0
- \( UMax = 2^w - 1 \)
  
  111...1

**Two’s Complement Values**
- \( TMin = -2^{w-1} \)
  
  10000000 00000000
- \( TMax = 2^{w-1} - 1 \)
  
  01111111 11111111

**Other Values**
- 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>UMax</th>
<th>UMin</th>
<th>TMax</th>
<th>TMin</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>255</td>
<td>-255</td>
<td>255</td>
<td>-255</td>
</tr>
<tr>
<td>16</td>
<td>65535</td>
<td>-32768</td>
<td>65535</td>
<td>-32768</td>
</tr>
<tr>
<td>32</td>
<td>4294967295</td>
<td>-2147483647</td>
<td>4294967295</td>
<td>-2147483647</td>
</tr>
<tr>
<td>64</td>
<td>18446744073709551615</td>
<td>-9223372036854775807</td>
<td>18446744073709551615</td>
<td>-9223372036854775807</td>
</tr>
</tbody>
</table>

**Observations**
- \(|TMin| = TMax + 1\)
  - Asymmetric range
- \( UMax = 2 \cdot TMax + 1 \)

**C Programming**
- \#include <limits.h>
- K&R App. B11
- Declares constants, e.g.,
  - ULONG_MAX
  - LONG_MAX
  - LONG_MIN
- Values platform-specific

**UnSigned & Signed Numeric Values**

<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>

**Example Values**
- \( W = 4 \)

**Equivalence**
- Same encodings for nonnegative values

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

\( \Rightarrow \) 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

**Casting Signed to Unsigned**

C Allows Conversions from Signed to Unsigned

```c
short int x = 15213;
unsigned short int ux = (unsigned short) x;
short int y = -15213;
unsigned short int uy = (unsigned short) y;
```

**Resulting Value**
- No change in bit representation
- Nonnegative values unchanged
  - \( ux = 15213 \)
- Negative values change into (large) positive values
  - \( uy = 50323 \)
Relation Between 2’s Comp. & Unsigned

Two’s Complement

\[ x \xrightarrow{T2B} \overline{x} \xrightarrow{B2U} u_x \]

Maintain Same Bit Pattern

\[ u_x = \begin{cases} x & x \geq 0 \\ x + 2^w & x < 0 \end{cases} \]

Relation Between Signed & Unsigned

<table>
<thead>
<tr>
<th>Weight</th>
<th>-15213</th>
<th>50323</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>32</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>64</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>128</td>
<td>1</td>
<td>128</td>
</tr>
<tr>
<td>256</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>512</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1024</td>
<td>1</td>
<td>1024</td>
</tr>
<tr>
<td>2048</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4096</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8192</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>16384</td>
<td>1</td>
<td>16384</td>
</tr>
<tr>
<td>32768</td>
<td>1</td>
<td>32768</td>
</tr>
</tbody>
</table>

\[ u_y = y + 2 \cdot 32768 = y + 65536 \]

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
  \[ \text{int} \ tx, \ ty; \]
  \[ \text{unsigned} \ ux, \ uy; \]
  \[ tx = (\text{int}) ux; \]
  \[ uy = (\text{unsigned}) ty; \]
- Implicit casting also occurs via assignments and procedure calls
  \[ tx = ux; \]
  \[ uy = ty; \]

Casting Surprises

Expression Evaluation
- If mix unsigned and signed in single expression, signed values implicitly cast to unsigned
- Including comparison operations \(<, >, ==, <=, >=\)
- Examples for \(W = 32\)

Constant_1 | Constant_2 | Relation | Evaluation
---|---|---|---
0 | 0U | 0 | 0
-1 | 0 | -1 | 0U
2147483647 | -2147483648 | -1 | 2147483647U
2147483647U | -2147483648 | (unsigned) -1 | -2
2147483647 | 2147483648U | (int) 2147483648U |
Explanation of Casting Surprises

2’s Comp. → Unsigned

- Ordering Inversion
- Negative → Big Positive

<table>
<thead>
<tr>
<th>TMin</th>
<th>0</th>
<th>-1</th>
<th>-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMin</td>
<td>0</td>
<td>UMax - 1</td>
<td>UMax</td>
</tr>
</tbody>
</table>

TMax + 1

Unsigned Range

UMax

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:
  \[ X' = x_{w-1}, \ldots, x_{w-1}, x_{w-2}, \ldots, x_0 \]

\( k \) copies of MSB

Sign Extension Example

Converting from smaller to larger integer data type
- C automatically performs sign extension

```
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>0011 1011 0110 1101</td>
</tr>
<tr>
<td>ix</td>
<td>15213</td>
<td>00 00 C4 92 0000 0000 0000 0000 0011 1011 0101</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>C4 93</td>
</tr>
<tr>
<td>iy</td>
<td>-15213</td>
<td>FF FF C4 93 1111 1111 1111 1111 1111 1100 1000 1000 1010</td>
</tr>
</tbody>
</table>

Justification For Sign Extension

Prove Correctness by Induction on \( k \)
- Induction Step: extending by single bit maintains value

\( -2^{w-1} = -2^w + 2^{w-1} \)
- Look at weight of upper bits:
  \[ X = -2^{w-1} x_{w-1} \]
  \[ X' = -2^w x_{w-1} + 2^{w-1} x_{w-1} = -2^{w-1} x_{w-1} \]
Why Should I Use Signed?

Don’t Use Just Because Number Nonzero

• C compilers on some machines generate less efficient code
  ```c
  unsigned i;
  for (i = 1; i < cnt ; i++)
    a[i] += a[i-1];
  ```

• Easy to make mistakes
  ```c
  for (i = cnt-2; i >= 0; i-- )
    a[i] += a[i+1];
  ```

Do Use When Performing Modular Arithmetic

• Multiprecision arithmetic
• Other esoteric stuff

Do Use When Need Extra Bit’s Worth of Range

• Working right up to limit of word size

Negating with Complement & Increment

In C

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

Complement

• Observation: \[-x + x == 1111...11_2 == -1\]
  ```
  \[
  x \quad \begin{array}{c}
  1001101 \\
  + \quad \sim x \\
  \hline
  1111111
  \end{array}
  \]

Increment

• \[-x + x + (-x + 1) == \sim x + (-x + 1)\]
• \[-x + 1 == \sim x\]

Warning: Be cautious treating `int`’s as integers

• OK here

Comp. & Incr. Examples

<table>
<thead>
<tr>
<th>x = 15213</th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>15213</td>
<td>3B 6D</td>
<td>00111011 01101101</td>
<td></td>
</tr>
<tr>
<td>-15214</td>
<td>C4 92</td>
<td>11000100 10010010</td>
<td></td>
</tr>
<tr>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
<td></td>
</tr>
<tr>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>-x</th>
<th>15213</th>
<th>3B 6D</th>
<th>00111011 01101101</th>
</tr>
</thead>
<tbody>
<tr>
<td>-x + 1</td>
<td>C4 93</td>
<td>11000100 10010011</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>C4 93</td>
<td>11000100 10010011</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>-1</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>-0 + 1</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Unsigned Addition

Operands: w bits

\[ u \]

\[ v \]

True Sum: w+1 bits

\[ u + v \]

Discard Carry: w bits

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

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 true 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

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 \( \text{UComp}_w(u) = 2^w - u \)
  \( \text{UAdd}_w(u, \text{UComp}_w(u)) = 0 \)

Two’s Complement Addition

Operands: \( w \) bits

\[
\begin{array}{cccccc}
& & & & & \\
\text{UAdd}_w(u, v) & & & & & \\
\end{array}
\]

True Sum: \( w+1 \) bits

\[
\begin{array}{cccccc}
& & & & & \\
\text{UAdd}_w(u, v) & & & & & \\
\end{array}
\]

Discard Carry: \( w \) bits

\[
\begin{array}{cccccc}
& & & & & \\
\text{TAdd}_w(u, v) & & & & & \\
\end{array}
\]

TAdd and UAdd have Identical Bit-Level Behavior
- Signed vs. unsigned addition in C:
  \[
  \begin{array}{l}
  \text{int } s, t, u, v; \\
  s = (\text{int}) ((\text{unsigned}) u + (\text{unsigned}) v); \\
  t = u + v \\
  \end{array}
  \]
- Will give \( s == t \)
Characterizing TAdd

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

\[
TAdd(u, v) = \begin{cases} 
  u + v & \text{if } u + v \leq TMax_w \\
  (u + v - 2^w) + TMax_w & \text{if } u + v > TMax_w 
\end{cases}
\]

\[
TAdd(u, v) = \begin{cases} 
  u + v & \text{if } u + v \leq TMax_w \\
  u + v - 2^w & \text{if } u + v > TMax_w 
\end{cases}
\]

Visualizing 2’s Comp. Addition

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

Wraps Around
- If sum \( \geq 2^w \)
  - Becomes negative
  - At most once
- If sum \( < -2^w \)
  - Becomes positive
  - At most once

Detecting 2’s Comp. Overflow

Task
- Given \( s = TAdd(u, v) \)
- Determine if \( s = Add_w(u, v) \)
- Example
  ```c
  int s, u, v;
  s = u + v;
  
  Claim
  - Overflow if either:
    \( u, v < 0, s \geq 0 \) (NegOver)
    \( u, v \geq 0, s < 0 \) (PosOver)
  
  ovf = (u<0 == v<0) && (u<0 || s<0);
  ```

Mathematical Properties of TAdd

Isomorphic Algebra to UAdd
- \( TAdd_w(u, v) = U2T(Add_w(T2U(u), T2U(v))) \)
  - 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 \( TComp_w(u) = U2T(UComp_w(T2U(u))) \)
  \( TAdd_w(u, TComp_w(u)) = 0 \)

\[
TComp_w(u) = \begin{cases} 
  -u & u \neq TMin_w \\
  u = TMin_w & \text{else}
\end{cases}
\]
Negating with Complement & Increment

In C

$\neg x + 1 \equiv \neg x$

Complement

- Observation: $\neg x + x \equiv \underbrace{111\ldots1}_{2w} \equiv -1$

Increment

- $\neg x + x + (-x + 1) \equiv \neg x + (-x + y)$
- $\neg x + 1 \equiv \neg x$

Warning: Be cautious treating int's as integers
- OK here: We are using group properties of TAdd and TComp

Multiplication

Computing Exact Product of $w$-bit numbers $x, y$
- Either signed or unsigned

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

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, and other “advanced” languages

Unsigned Multiplication in C

Standard Multiplication Function
- Ignores high order $w$ bits

Implements Modular Arithmetic

$UMult_w(u, v) \equiv u \cdot v \mod 2^w$

Unsigned vs. Signed Multiplication

Unsigned Multiplication

unsigned ux = (unsigned) x;
unsigned uy = (unsigned) y;
unsigned up = ux * uy
- Truncates product to $w$-bit number $up = UMult_w(ux, uy)$
- Simply modular arithmetic
  $up = ux \cdot uy \mod 2^w$

Two’s Complement Multiplication

int x, y;
int p = x * y;
- Compute exact product of two $w$-bit numbers $x, y$
- Truncate result to $w$-bit number $p = TMult_w(x, y)$

Relation
- Signed multiplication gives same bit-level result as unsigned
- $up \equiv (unsigned) p$
Power-of-2 Multiply with Shift

**Operation**
- \( u << k \) gives \( u \times 2^k \)
- Both signed and unsigned
- Operands: \( w \) bits
- True Product: \( w+k \) bits
- Discard \( k \) bits: \( w \) bits

**Examples**
- \( u << 3 \) gives \( u \times 8 \)
- \( u << 5 - u << 3 \) gives \( u \times 24 \)
- Most machines shift and add much faster than multiply
  - Compiler will generate this code automatically

---

Unsigned Power-of-2 Divide with Shift

**Quotient of Unsigned by Power of 2**
- \( u >> k \) gives \( \lfloor u / 2^k \rfloor \)
- Uses logical shift
- Operands: \( u / 2^k \)
- Division: \( u / 2^k \)
- Quotient: \( \lfloor u / 2^k \rfloor \)

**2’s Comp Power-of-2 Divide with Shift**

**Quotient of Signed by Power of 2**
- \( u >> k \) gives \( \lfloor u / 2^k \rfloor \)
- Uses arithmetic shift
- Rounds wrong direction when \( u < 0 \)
- Operands: \( u \)
- Division: \( u / 2^k \)
- Result: \( \lfloor u / 2^k \rfloor \)

**Correct Power-of-2 Divide**

**Quotient of Negative Number by Power of 2**
- Want \( \lceil u / 2^k \rceil \) (Round Toward 0)
- Compute as \( \lceil (u+2^k-1) / 2^k \rceil \)
- In C: \( u + (1<<k) - 1 \) >> \( k \)
- Biases dividend toward 0

**Case 1: No rounding**
- Dividend: \( u \)
- Divisor: \( 2^k \)
- Result: \( \lceil u / 2^k \rceil \)

---

Biasing has no effect
Correct Power-of-2 Divide (Cont.)

Case 2: Rounding

<table>
<thead>
<tr>
<th>Dividend:</th>
<th>( u )</th>
<th>( 2^k )</th>
<th>( +2^k - 1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Incremented by 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary Point</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Divisor:</th>
<th>( / 2^k )</th>
<th>( \lceil u / 2^k \rceil )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Incremented by 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Biasing adds 1 to final result

Properties of Unsigned Arithmetic

Unsigned Multiplication with Addition Forms

Commutative Ring

- Addition is commutative group
- Closed under multiplication
  \( 0 \leq \text{Mult}_u(u, v) \leq 2^w - 1 \)
- Multiplication Commutative
  \( \text{Mult}_u(u, v) = \text{Mult}_u(v, u) \)
- Multiplication is Associative
  \( \text{Mult}_u(t, \text{Mult}_u(u, v)) = \text{Mult}_u(\text{Mult}_u(t, u), v) \)
- 1 is multiplicative identity
  \( \text{Mult}_u(u, 1) = u \)
- Multiplication distributes over addition
  \( \text{Mult}_u(t, \text{Add}_u(u, v)) = \text{Add}_u(\text{Mult}_u(t, u), \text{Mult}_u(t, v)) \)

Properties of Two’s Comp. Arithmetic

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 \( 2^w \)

Comparison to Integer Arithmetic

- Both are rings
- Integers obey ordering properties, e.g.,
  \( u > 0 \) \( \Rightarrow \) \( u + v > v \)
  \( u > 0, v > 0 \) \( \Rightarrow \) \( u \cdot v > 0 \)
- These properties are not obeyed by two’s complement arithmetic

\( T_{\text{Max}} + 1 = T_{\text{Min}} \)
\( 15213 \times 30426 = -10030 \) (16-bit words)

C Puzzle Answers

- Assume machine with 32 bit word size, two’s complement integers
- \( T_{\text{Min}} \) makes a good counterexample in many cases

- \( x < 0 \) \( \Rightarrow \) \( (x*2) < 0 \)
- \( u < 0 \) \( \Rightarrow \) \( (u+u) < 0 \)
- \( x \& 7 == 7 \) \( \Rightarrow \) \( (x\ll30) < 0 \)
- \( u < -1 \)
- \( x > y \) \( \Rightarrow \) \( -x < -y \)
- \( x + x >= 0 \)
- \( x > 0 \&\& y > 0 \) \( \Rightarrow \) \( x + y > 0 \)
- \( x >= 0 \) \( \Rightarrow \) \( -x <= 0 \)
- \( x <= 0 \) \( \Rightarrow \) \( -x >= 0 \)