Announcements

- All Fish-machine accounts are up (for both regular & waitlisted students)
- All Autolab accounts are up (no more excuse to delay)
- Lab1 deadline is Friday, January 23 @ 11:59pm
  That is 3 days, 14 hours and 58min from now!
- Lab2 is out, due in 2 week
  (Wednesday, Feb. 4 @11:59pm)

- Waitlisted students (58): need to sign attendance sheet
  after each lecture to maintain waitlisted status. We will
  add 30 students and expect about 20+ dropouts based
  on past experience.

C Puzzles

- Taken from old exams
- 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;
```

```
( )< 0
[x*2] < 0
x >= 0
x & 7 == 7
(x<<30) < 0
ux >= 0
ux > -1
ux >> 0
ux > y
ux >> 0
ux << 0
```

Encoding Integers

### Two’s Complement

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

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>15213</td>
<td>00111011 01101101</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
Encoding Example (Cont.)

- 5 - 15-213, S'04

<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>0</td>
</tr>
<tr>
<td>4</td>
<td>0</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>32</td>
</tr>
<tr>
<td>64</td>
<td>1</td>
<td>64</td>
</tr>
<tr>
<td>128</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>256</td>
<td>1</td>
<td>256</td>
</tr>
<tr>
<td>512</td>
<td>1</td>
<td>512</td>
</tr>
<tr>
<td>1024</td>
<td>0</td>
<td>1024</td>
</tr>
<tr>
<td>2048</td>
<td>1</td>
<td>2048</td>
</tr>
<tr>
<td>4096</td>
<td>1</td>
<td>4096</td>
</tr>
<tr>
<td>8192</td>
<td>1</td>
<td>8192</td>
</tr>
<tr>
<td>16384</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>-32768</td>
<td>0</td>
<td>-32768</td>
</tr>
<tr>
<td>Sum</td>
<td>15213</td>
<td>-15213</td>
</tr>
</tbody>
</table>

- 6 - 15-213, S'04

Numeric Ranges

Unsigned Values
- \( U_{\text{Min}} = 0 \)
- \( U_{\text{Max}} = 2^w - 1 \)

Two’s Complement Values
- \( T_{\text{Min}} = -2^{w-1} \)
- \( T_{\text{Max}} = 2^w - 1 \)

Other Values
- Minus 1
- \( 11_{\text{11}} \)

Values for \( W = 16 \)

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>65535</td>
<td>FFFFFFFF</td>
</tr>
<tr>
<td>TMax</td>
<td>32767</td>
<td>7F FF</td>
</tr>
<tr>
<td>TMin</td>
<td>-32768</td>
<td>80 00</td>
</tr>
<tr>
<td>-1</td>
<td>-1 FF</td>
<td>FFFFFFFF</td>
</tr>
<tr>
<td>0</td>
<td>00 00</td>
<td>00000000</td>
</tr>
</tbody>
</table>

- 7 - 15-213, S'04

Values 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
- \( U_{\text{Max}} = 2 \cdot T_{\text{Max}} + 1 \)

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

- 8 - 15-213, S'04

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>

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
Casting Signed to Unsigned

C Allows Conversions from Signed to Unsigned

\[
\begin{align*}
\text{short int } & \quad x = 15213; \\
\text{unsigned short int } & \quad ux = (\text{unsigned short}) x; \\
\text{short int } & \quad y = -15213; \\
\text{unsigned short int } & \quad uy = (\text{unsigned short}) y;
\end{align*}
\]

Resulting Value
- No change in bit representation
- Nonnegative values unchanged
- \(ux = 15213\)
- Negative values change into (large) positive values
- \(uy = 50323\)

Relation between Signed & Unsigned

Two’s Complement

Maintain Same Bit Pattern

\[
\begin{align*}
x & \quad \text{Two’s Complement} \\
\text{T2U} & \quad \text{Unsigned} \\
x & \quad \text{T2B} \\
\text{S2U} & \quad x
\end{align*}
\]

Relation Between Signed & Unsigned

<table>
<thead>
<tr>
<th>Weight</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
<th>8192</th>
<th>16384</th>
<th>32768</th>
<th>65536</th>
</tr>
</thead>
<tbody>
<tr>
<td>-15213</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>50323</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{Sum} & \quad -15213 \\
& \quad 50323
\end{align*}
\]

Signed vs. Unsigned in C

Constants
- By default are considered to be signed integers
- Signed if have “U” as suffix
  \(0U, 4294967259U\)

Casting
- Explicit casting between signed & unsigned same as U2T and T2U
  \[
  \begin{align*}
  \text{int } & \quad tx, ty; \\
  \text{unsigned } & \quad ux, uy; \\
  tx & = \text{ (int) } ux; \\
  uy & = \text{ (unsigned) } ty;
  \end{align*}
  \]
- Implicit casting also occurs via assignments and procedure calls
  \[
  \begin{align*}
  tx & = ux; \\
  uy & = ty;
  \end{align*}
  \]
### Casting Surprises

**Expression Evaluation**
- If mix unsigned and signed in single expression, signed values implicitly cast to unsigned
- Including comparison operations \(<\), \(\le\), \(\ge\)

**Examples for \(W = 32\)**

<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>00</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>00</td>
<td>&gt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>-2147483648</td>
<td>&gt;</td>
<td>signed</td>
</tr>
<tr>
<td>-1</td>
<td>-2</td>
<td>&gt;</td>
<td>unsigned</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>

---

### Explanation of Casting Surprises

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

#### Sign Extension Example

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

---

### 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:
  - \(X' = x_{w-1}, \ldots, x_{w+k-1}, x_{w+k-2}, \ldots, x_0\)

- Converting from smaller to larger integer data type
- C automatically performs sign extension
Justification For Sign Extension

Prove Correctness by Induction on $k$
- Induction Step: extending by single bit maintains value
- Key observation: $-2^{w-1} = -2^w + 2^{w-1}$
- Look at weight of upper bits:
  - $X = -2^{w-1}X_{w-1}$
  - $X' = -2^wX_{w-1} + 2^{w-1}X_{w-1} = -2^{w-1}X_{w-1}$

Why Should I Use Unsigned?

Don’t Use Just Because Number Nonzero
- C compilers on some machines generate less efficient code
  ```c
  int 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

Claim: Following Holds for 2’s Complement
- $x + 1 = -x$

Complement
- Observation: $\overline{x} + x = \overline{1111…11} = -1$
- $\overline{x}$
- $\overline{x} + 1$
- Increment
- $\overline{x} + (\overline{x} + 1) = -x + (\overline{x} + 1)$
- $\overline{x} + 1 = -x$

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

Comp. & Incr. Examples

$x = 15213$

<table>
<thead>
<tr>
<th>$x$</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>$\overline{x}$</td>
<td>-15214</td>
<td>C 92</td>
<td>11000100 10010010</td>
</tr>
<tr>
<td>$\overline{x}+1$</td>
<td>-15213</td>
<td>C 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>$y$</td>
<td>-15213</td>
<td>C 93</td>
<td>11000100 10010011</td>
</tr>
</tbody>
</table>

0

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 0 0</td>
<td>00000000 00000000</td>
</tr>
<tr>
<td>-0</td>
<td>-1 FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>-0+1</td>
<td>0 0 0 0</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>
Unsigned Addition

**Standard Addition Function**
- Ignores carry output
- Implements Modular Arithmetic

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

Visualizing Integer Addition

**Integer Addition**
- 4-bit integers \( u, v \)
- Compute true sum \( \text{Add}_d(u, v) \)
- Values increase linearly with \( u \) and \( v \)
- Forms planar surface

Visualizing Unsigned Addition

**Wraps Around**
- If true sum \( \geq 2^w \)
- At most once

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

Mathematical Properties

**Modular Addition Forms an Abelian Group**
- Closed under addition
  \[ 0 \leq \text{UAdd}_d(u, v) \leq 2^w - 1 \]
- Commutative
  \[ \text{UAdd}_d(u, v) = \text{UAdd}_d(v, u) \]
- Associative
  \[ \text{UAdd}_d(t, \text{UAdd}_d(u, v)) = \text{UAdd}_d(\text{UAdd}_d(t, u), v) \]
- 0 is additive identity
  \[ \text{UAdd}_d(u, 0) = u \]
- Every element has additive inverse
  - Let \( \text{UComp}_w(u) = 2^w - u \)
  \[ \text{UAdd}_d(u, \text{UComp}_w(u)) = 0 \]
Two's Complement Addition

Two's Complement Addition

Operands: w bits

\[ u + v \]

True Sum: w+1 bits

\[ u + v + \text{Discard Carry} \]

Discard Carry: w bits

TAdd(u, v)

TAdd and UAdd have Identical Bit-Level Behavior

- Signed vs. unsigned addition in C:
  \[
  \text{int } s, t, u, v; s = (\text{int}) \{(\text{unsigned}) u + (\text{unsigned}) v\};
  \]
  \[ t = u + v \]
- Will give \( s \equiv t \)

Characterizing TAdd

Functionality

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

True Sum

\[
\begin{array}{ccccccc}
0 & 111 & \ldots & 1 & 0 & 100 & \ldots & 0 \\
\hline
\text{NegOver} & \text{TAdd Result} & \text{PosOver}
\end{array}
\]

TAdd(u, v) = \[
\begin{align*}
&u + v + 2^{w-1} & & u + v < TMin_u, \text{(NegOver)} \\
&u + v & & TMin_u \leq u + v \leq TMax_u \\
&u + v - 2^{w-1} & & TMax_u < u + v \text{(PosOver)}
\end{align*}
\]

Detecting 2's Comp. Overflow

Task

- Given \( s = \text{TAdd}_w(u, v) \)
- Determine if \( s = \text{Add}_w(u, v) \)
- Example
  \[
  \text{int } s, u, v; s = u + v;
  \]

Claim

- Overflow iff either:
  \[
  u, v < 0, s \geq 0 \text{ (NegOver)} \\
  u, v \geq 0, s < 0 \text{ (PosOver)}
  \]

Claim:

\[
\begin{align*}
\text{ovf} &= (u < 0 \equiv v < 0) \&\& (u < 0 \equiv s < 0) \\
\end{align*}
\]

Visualizing 2's Comp. Addition

Values

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

Wraps Around

- If sum \( \geq 2^{w-1} \)
  - Becomes negative
  - At most once
- If sum \( \leq -2^{w-1} \)
  - Becomes positive
  - At most once
Mathematical Properties of TAdd

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

\[
TComp_w(u) = \begin{cases} 
-u & u \neq TMin_w \\
TMin_w & u = TMin_w 
\end{cases}
\]

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 \((TMin_w)^2\)

Maintaining Exact Results
- Would need to keep expanding word size with each product computed
- Done in software by “arbitrary precision” arithmetic packages

Unsigned Multiplication in C

Operands: \( w \) bits

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c}
\hline
& u & & & & & & & & & & \hline
\hline
* v & & & & & & & & & & \hline
\hline
\end{array}
\]

True Product: \( 2^w \) bits

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c}
\hline
& u & & & & & & & & & & \hline
\hline
\cdot v & & & & & & & & & & \hline
\hline
\end{array}
\]

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

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c}
\hline
UMult_w(u, v) & & & & & & & & & & \hline
\hline
\end{array}
\]

Standard Multiplication Function
- Ignores high order \( w \) bits

Implements Modular Arithmetic
- \( UMult_u(u, v) = u \cdot v \mod 2^w \)

Unsigned vs. Signed Multiplication

Unsigned Multiplication
- \( \text{unsigned } ux = (\text{unsigned}) x; \)
- \( \text{unsigned } uy = (\text{unsigned}) y; \)
- \( \text{unsigned } up = ux \cdot uy \)

- Truncates product to \( w \)-bit number \( up = UMult_u(ux, uy) \)
- Modular arithmetic: \( up = ux \cdot uy \mod 2^w \)

Two’s Complement Multiplication
- \( \text{int } x, y; \)
- \( \text{int } p = x \cdot y; \)
- Compute exact product of two \( w \)-bit numbers \( x, y \)
- Truncate result to \( w \)-bit number \( p = TMult_u(x, y) \)
Unsigned vs. Signed Multiplication

Unsigned Multiplication
- unsigned ux = (unsigned) x;
- unsigned uy = (unsigned) y;
- unsigned up = ux * uy

Two's Complement Multiplication
- int x, y;
- int p = x * y;

Relation
- Signed multiplication gives same bit-level result as unsigned
- $up == (\text{unsigned}) p$

Power-of-2 Multiply with Shift

Operation
- $u << k$ gives $u \cdot 2^k$
- Both signed and unsigned

Operands: w bits
- $u$
- $2^k$

True Product: w-k bits
- $u \cdot 2^k$

Discard k bits: w bits
- UMult$_w(u, 2^k)$
- TMult$_w(u, 2^k)$

Examples
- $u << 3 == u \cdot 8$
- $u << 5 - u << 3 == u \cdot 24$
- Most machines shift and add much 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 shift

Operands:
- $u$
- $2^k$

Division:
- $u / 2^k$

Result:
- $\lfloor u / 2^k \rfloor$

<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 0110101</td>
</tr>
<tr>
<td>x &gt;&gt; 1</td>
<td>7608,5</td>
<td>1D B6</td>
<td>00011101 10110110</td>
</tr>
<tr>
<td>x &gt;&gt; 4</td>
<td>950,6125</td>
<td>03 B6</td>
<td>00000001 10110110</td>
</tr>
<tr>
<td>x &gt;&gt; 8</td>
<td>59,4257813</td>
<td>3B</td>
<td>00000000 00111011</td>
</tr>
</tbody>
</table>

Signed Power-of-2 Divide with Shift

Quotient of Signed by Power of 2
- $x \gg k$ gives $\lfloor x / 2^k \rfloor$
- Uses arithmetic shift
- Rounds wrong direction when $u < 0$

Operands:
- $x$
- $2^k$

Division:
- $x / 2^k$

Result:
- RoundDown($x / 2^k$)

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>-15213</td>
<td>C6 93</td>
<td>11000100 10010101</td>
</tr>
<tr>
<td>y &gt;&gt; 1</td>
<td>-7608,5</td>
<td>7607</td>
<td>11100010 01001001</td>
</tr>
<tr>
<td>y &gt;&gt; 4</td>
<td>-950,6125</td>
<td>69 93</td>
<td>11111110 00100101</td>
</tr>
<tr>
<td>y &gt;&gt; 8</td>
<td>-59,4257813</td>
<td>-60</td>
<td>11111111 11000100</td>
</tr>
</tbody>
</table>

Division | Computed | Hex | Binary         |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>-15213</td>
<td>C6 93</td>
<td>11000100 10010101</td>
</tr>
<tr>
<td>y &gt;&gt; 1</td>
<td>-7608,5</td>
<td>7607</td>
<td>11100010 01001001</td>
</tr>
<tr>
<td>y &gt;&gt; 4</td>
<td>-950,6125</td>
<td>69 93</td>
<td>11111110 00100101</td>
</tr>
<tr>
<td>y &gt;&gt; 8</td>
<td>-59,4257813</td>
<td>-60</td>
<td>11111111 11000100</td>
</tr>
</tbody>
</table>
**Correct Power-of-2 Divide**

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

**Case 1: No rounding**
- \[ \text{Dividend: } \]
- \[ \text{Divisor: } \]
  - \[ \text{Biasing has no effect} \]

**Case 2: Rounding**
- \[ \text{Dividend: } \]
- \[ \text{Divisor: } \]
  - \[ \text{Biasing adds } 1 \text{ to final result} \]

---

**Properties of Unsigned Arithmetic**

**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
  - $\text{UMult}_w(t, \text{UAdd}_w(u, v)) = \text{UAdd}_w(\text{UMult}_w(t, u), \text{UMult}_w(t, v))$

---

**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.,
    - $u > 0 \quad \Rightarrow \quad u + v > v$
    - $u > 0, v > 0 \quad \Rightarrow \quad u \cdot v > 0$
- These properties are not obeyed by two’s comp. arithmetic
  - $\text{TMax} + 1 \neq \text{TMin}$

---

Page 10
### C Puzzle Answers

- Assume machine with 32 bit word size, two’s comp. integers
- TMin makes a good counterexample in many cases

<table>
<thead>
<tr>
<th>Condition</th>
<th>Implication</th>
<th>True/False</th>
<th>Counterexample</th>
</tr>
</thead>
<tbody>
<tr>
<td>x &lt; 0</td>
<td>(x*2) &lt; 0</td>
<td>False: TMin</td>
<td></td>
</tr>
<tr>
<td>ux &gt;= 0</td>
<td></td>
<td>True: 0 = UMin</td>
<td></td>
</tr>
<tr>
<td>x &amp; 7 == 7</td>
<td>(x&lt;&lt;30) &lt; 0</td>
<td>True: x = 1</td>
<td></td>
</tr>
<tr>
<td>ux &gt; -1</td>
<td></td>
<td>False: 0</td>
<td></td>
</tr>
<tr>
<td>x &gt; y</td>
<td>-x &lt; -y</td>
<td>False: -1, TMin</td>
<td></td>
</tr>
<tr>
<td>x * x &gt;= 0</td>
<td></td>
<td>False: 30426</td>
<td></td>
</tr>
<tr>
<td>x &gt; 0 &amp;&amp; y &gt; 0</td>
<td>x + y &gt; 0</td>
<td>False: TMax, TMax</td>
<td></td>
</tr>
<tr>
<td>x &gt;= 0</td>
<td>-x &lt;= 0</td>
<td>True: -TMax &lt; 0</td>
<td></td>
</tr>
<tr>
<td>x &lt;= 0</td>
<td>-x &gt;= 0</td>
<td>False: TMin</td>
<td></td>
</tr>
</tbody>
</table>