15-213
“The course that gives CMU its Zip!”

Integers
Sep 2, 2003

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 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;
```

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

15-213, F’03
Encoding Integers

**Unsigned**

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

**Two’s Complement**

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

- C `short` 2 bytes long

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>15213</td>
<td>3B 6D</td>
<td>001111011 01101101</td>
</tr>
<tr>
<td>( y )</td>
<td>-15213</td>
<td>C4 93</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.)

\[ x = 15213: \quad 00111011 \quad 01101101 \]
\[ y = -15213: \quad 11000100 \quad 10010011 \]

<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>1</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>0</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>0</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>0</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>0</td>
</tr>
<tr>
<td>-32768</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Sum</strong></td>
<td><strong>15213</strong></td>
<td><strong>-15213</strong></td>
</tr>
</tbody>
</table>
Numeric Ranges

Unsigned Values
- UMin = 0
  000...0
- UMax = 2^w – 1
  111...1

Two’s Complement Values
- Tmin = –2^{w-1}
  100...0
- Tmax = 2^{w-1} – 1
  011...1

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></td>
<td>00 00</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>
Values for Different Word Sizes

<table>
<thead>
<tr>
<th></th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
</tr>
<tr>
<td>UMax</td>
<td>255</td>
</tr>
<tr>
<td>Tmax</td>
<td>127</td>
</tr>
<tr>
<td>Tmin</td>
<td>-128</td>
</tr>
</tbody>
</table>

**Observations**

- $|TMin| = Tmax + 1$
  - Asymmetric range
- $UMax = 2 \times Tmax + 1$

**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>

**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

```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 Signed & Unsigned

Two’s Complement → T2U → T2B → B2U → Unsigned

Maintain Same Bit Pattern

\[ +2^{w-1} - (-2^{w-1}) = 2*2^{w-1} = 2^w \]

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

**Sum**

\[ uy = y + 2 \times 32768 = y + 65536 \]
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
  
  int tx, ty;
  unsigned ux, uy;
  tx = (int) ux;
  uy = (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 $W = 32$

<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>-2147483648</td>
<td>&gt;</td>
<td>signed</td>
</tr>
<tr>
<td>2147483647U</td>
<td>-2147483648</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>
Explanation of Casting Surprises

2’s Comp. → Unsigned

- Ordering Inversion
- Negative → Big Positive

2’s Comp. Range

Ordering Inversion: $TMax \rightarrow UMax$

Negative → Big Positive: $TMax + 1 \rightarrow UMax - 1$

Unsigned Range

Ordering Inversion: $TMax \rightarrow UMax$

Negative → Big Positive: $TMax + 1 \rightarrow UMax - 1$
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:
- \( X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0 \)

\( X \)
\[ \begin{array}{c}
\hline
\vdots \\
\hline
x \\
\hline
\vdots \\
\hline
\end{array} \]

\( X' \)
\[ \begin{array}{c}
\hline
\vdots \\
\hline
\text{\( k \) copies of MSB} \quad \text{\( w \) bits} \\
\hline
\vdots \\
\hline
\end{array} \]
Sign Extension Example

```
short int x = 15213;
int ix = (int) x;
short int y = -15213;
int iy = (int) y;
```

<table>
<thead>
<tr>
<th></th>
<th>Decimal</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>ix</td>
<td>15213</td>
<td>00 00 3B 6D</td>
<td>00000000 00000000 00111011 01101101</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>iy</td>
<td>-15213</td>
<td>FF FF C4 93</td>
<td>11111111 11111111 11000100 10010011</td>
</tr>
</tbody>
</table>

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

\[
x \quad -2^{w-1} x_{w-1}
\]
\[
x' \quad -2^w x_{w-1} + 2^{w-1} x_{w-1} = -2^{w-1} x_{w-1}
\]
Why Should I Use Unsigned?

Don’t Use Just Because Number Nonzero

- C compilers on some machines generate less efficient code
  
  ```c
  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

Claim: Following Holds for 2’s Complement

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

Complement

- **Observation:** \( \neg x + x = 1111\ldots11_2 = -1 \)

<table>
<thead>
<tr>
<th>x</th>
<th>10011101</th>
</tr>
</thead>
<tbody>
<tr>
<td>( + \neg x )</td>
<td>01100010</td>
</tr>
<tr>
<td><strong>Result:</strong></td>
<td>11111111</td>
</tr>
</tbody>
</table>

Increment

- \( \neg x + x + (\neg x + 1) = -1 + (\neg x + 1) \)
- \( \neg x + 1 = -x \)

Warning: Be cautious treating int’s as integers

- 18 – **OK here**
Comp. & Incr. Examples

\[ x = 15213 \]

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>15213</td>
<td>3B 6D 00111011 01101101</td>
</tr>
<tr>
<td>( \sim x )</td>
<td>-15214</td>
<td>C4 92 11000100 10010010</td>
</tr>
<tr>
<td>( \sim x + 1 )</td>
<td>-15213</td>
<td>C4 93 11000100 10010011</td>
</tr>
<tr>
<td>( y )</td>
<td>-15213</td>
<td>C4 93 11000100 10010011</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00 00 00000000 00000000</td>
</tr>
<tr>
<td>( \sim 0 )</td>
<td>-1</td>
<td>FF FF 11111111 11111111</td>
</tr>
<tr>
<td>( \sim 0 + 1 )</td>
<td>0</td>
<td>00 00 00000000 00000000</td>
</tr>
</tbody>
</table>
Unsigned Addition

Operands: $w$ bits

True Sum: $w+1$ bits

Discard Carry: $w$ bits

Standard Addition Function

- Ignores carry output

Implements Modular Arithmetic

$$s = UAdd_w(u, v) = u + v \mod 2^w$$

$$UAdd_w(u,v) = \begin{cases} 
    u + v & u + v < 2^w \\
    u + v - 2^w & u + v \geq 2^w 
\end{cases}$$
Integer Addition

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

Add$_4(u, v)$
Visualizing Unsigned Addition

Wraps Around

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

True Sum

$2^{w+1}$

$2^w$

0

Modular Sum

Overflow

UAdd$_4(u, v)$

Overflow
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

\[
\begin{array}{c}
\begin{array}{c}
\text{} \\
\text{u} \\
\text{+} \\
\text{v} \\
\text{u + v}
\end{array}
\end{array}
\]

True Sum: \( w+1 \) bits

Discard Carry: \( w \) bits

TAdd and UAdd have Identical Bit-Level Behavior

- **Signed vs. unsigned addition in C:**

  ```c
  int s, t, u, v;
  s = (int) ((unsigned) u + (unsigned) v);
  t = u + v
  ```

- **Will give**  \( s == t \)
Characterizing TAdd

Functionality

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

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

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

Wraps Around
- If sum $\geq 2^{w-1}$
  - Becomes negative
  - At most once
- If sum $< -2^{w-1}$
  - Becomes positive
  - At most once
Detecting 2’s Comp. Overflow

**Task**
- **Given** \( s = T\text{Add}_w(u, v) \)
- **Determine if** \( s = \text{Add}_w(u, v) \)
- **Example**
  ```
  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)

  \( \text{ovf} = (u<0 == v<0) && (u<0 != s<0); \)
Mathematical Properties of TAdd

Isomorphic Algebra to UAdd

- TAdd\(_w(u, v) = U2T(UAdd\(_w(T2U(u), T2U(v)))\)
  - 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 \\
TMin\(_w & u = TMin\(_w 
\end{cases}
\]
Multiplication

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

- Either signed or unsigned

Ranges

- **Unsigned:** $0 \leq x \times y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
  - Up to $2w$ bits
- **Two’s complement min:** $x \times y \geq (-2^{w-1}) \times (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$
  - Up to $2w-1$ bits
- **Two’s complement max:** $x \times y \leq (-2^{w-1})^2 = 2^{2w-2}$
  - 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
Unsigned Multiplication in C

Operands: \( w \) bits

\[ u \cdot v \]

True Product: \( 2^w \) bits

\[ u \cdot v \]

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

UMult\(_w\)(\( u \), \( v \))

Standard Multiplication Function

- Ignores high order \( w \) bits

Implements Modular Arithmetic

\[ \text{UMult}_w(u, v) = u \cdot v \mod 2^w \]
Unsigned vs. Signed Multiplication

Unsigned Multiplication

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

- Truncates product to $w$-bit number $up = \text{UMult}_w(ux, uy)$
- 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 $p = \text{TMult}_w(x, y)$
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 == (unsigned) p
Power-of-2 Multiply with Shift

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

![Diagram]

**Examples**
- $u << 3 = = u \times 8$
- $u << 5 - u << 3 = = u \times 24$
- 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

- \( u >> k \) gives \( \lfloor u / 2^k \rfloor \)
- Uses logical shift

\[
\begin{array}{c}
\text{Operands:} \\
\frac{u}{2^k} \\
\text{Division:} \\
\frac{u}{2^k} \\
\text{Result:} \end{array}
\]

<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

- $x >> k$ gives $\lfloor x / 2^k \rfloor$
- Uses arithmetic shift
- Rounds wrong direction when $u < 0$

Operands:

$\begin{array}{cccccc}
x & \cdots & \cdots & \cdots & \cdots & \cdots \\
\end{array}$

Division:

$x / 2^k$

Result: $\text{RoundDown}(x / 2^k)$

<table>
<thead>
<tr>
<th></th>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y$</td>
<td>-15213</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>$y &gt;&gt; 1$</td>
<td>-7606.5</td>
<td>-7607</td>
<td>E2 49</td>
<td>11100010 01001001</td>
</tr>
<tr>
<td>$y &gt;&gt; 4$</td>
<td>-950.8125</td>
<td>-951</td>
<td>FC 49</td>
<td>11111100 01001001</td>
</tr>
<tr>
<td>$y &gt;&gt; 8$</td>
<td>-59.4257813</td>
<td>-60</td>
<td>FF C4</td>
<td>11111111 11000100</td>
</tr>
</tbody>
</table>
Correct Power-of-2 Divide

Quotient of Negative Number by Power of 2

- Want \( \left\lfloor \frac{x}{2^k} \right\rfloor \) (Round Toward 0)
- Compute as \( \left\lfloor \frac{x+2^k-1}{2^k} \right\rfloor \)
  - In C: \((x + (1<<k) - 1) >> k\)
  - Biases dividend toward 0

Case 1: No rounding

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c}
\text{Dividend:} & u & \cdot & \cdot & \cdot & 0 & \cdot & \cdot & 0 & 0 \\
+2^k +1 & 0 & \cdot & \cdot & 0 & 0 & 1 & \cdot & \cdot & 1 & 1 \\
\hline
\text{Divisor:} & / & 2^k & 0 & \cdot & \cdot & 0 & 1 & 0 & \cdot & \cdot & 0 & 0 \\
\hline
\left\lfloor u / 2^k \right\rfloor & 1 & \cdot & \cdot & 1 & 1 & 1 & \cdot & \cdot & \cdot & 1 & 1 \\
\end{array}
\]

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

Case 2: Rounding

Dividend:

\[ x \]
\[ +2^k + 1 \]

Divisor:

\[ \div 2^k \]

\[ \left\lfloor \frac{x}{2^k} \right\rfloor \]

\[ k \]

Biasing adds 1 to final result

Incremented by 1

Binary Point

Incremented by 1
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.,
  
  \[
  \begin{align*}
  u > 0 & \quad \Rightarrow \quad u + v > v \\
  u > 0, \ v > 0 & \quad \Rightarrow \quad u \cdot v > 0
  \end{align*}
  \]
- These properties are not obeyed by two’s comp. arithmetic
  
  \[
  \begin{align*}
  T_{\text{Max}} + 1 & \quad = \quad T_{\text{Min}} \\
  -39 & \quad \Rightarrow \quad 15213 \times 30426 \quad = \quad -10030 \quad (16\text{-bit words})
  \end{align*}
  \]
C Puzzle Answers

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

- \( x < 0 \) \( \Rightarrow \) \((x \times 2) < 0\) \( \text{False: } TMin\)
- \( u x >= 0 \) \( \text{True: } 0 = UMin\)
- \( x \& 7 == 7 \) \( \Rightarrow \) \((x \ll 30) < 0\) \( \text{True: } x_1 = 1\)
- \( u x > -1 \) \( \text{False: } 0\)
- \( x > y \) \( \Rightarrow \) \(-x < -y\) \( \text{False: } -1, TMin\)
- \( x \times x >= 0 \) \( \text{False: } 30426\)
- \( x > 0 && y > 0 \) \( \Rightarrow \) \(x + y > 0\) \( \text{False: } TMax, TMax\)
- \( x >= 0 \) \( \Rightarrow \) \(-x <= 0\) \( \text{True: } -TMax < 0\)
- \( x <= 0 \) \( \Rightarrow \) \(-x >= 0\) \( \text{False: } TMin\)