Integers

Today
- Numeric Encodings
- Programming Implications
- Basic operations
- Programming Implications

Next time
- Floats
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;
```

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

**Unsigned**

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

(Binary To Unsigned)

**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>00111011 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

**x = 15213:**  
00111011 01101101

**y = -15213:**  
11000100 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>
</tbody>
</table>

**Sum**  
15213  -15213
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

- **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>$U_{\text{max}}$</td>
<td>65535</td>
<td>FF FF 11111111 11111111</td>
<td></td>
</tr>
<tr>
<td>$T_{\text{max}}$</td>
<td>32767</td>
<td>7F FF 01111111 11111111</td>
<td></td>
</tr>
<tr>
<td>$T_{\text{min}}$</td>
<td>-32768</td>
<td>80 00 10000000 00000000</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>FF FF 11111111 11111111</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00 00000000 00000000</td>
<td></td>
</tr>
</tbody>
</table>
Values for other 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
  - $|TMin| = |TMax| + 1$
    - Asymmetric range
  - $UMax = 2 \times TMax + 1$

- C Programming
  - `#include <limits.h>`
  - Declares constants, e.g.,
    - ULONG_MAX, UINT_MAX
    - LONG_MAX, INT_MAX
    - LONG_MIN, INT_MIN
  - Values platform-specific; for Java this is specified
Unsigned & signed numeric values

- **Equivalence**
  - Same encodings for nonnegative values

- **Uniqueness (bijections)**
  - 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>
C allows conversions from signed to unsigned

<table>
<thead>
<tr>
<th>short int</th>
<th>x = 15213;</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned short int ux = (unsigned short) x;</td>
<td></td>
</tr>
<tr>
<td>short int</td>
<td>y = -15213;</td>
</tr>
<tr>
<td>unsigned short int uy = (unsigned short) y;</td>
<td></td>
</tr>
</tbody>
</table>

Resulting value

- No change in bit representation
- Non-negative values unchanged
  - \( ux = 15213 \)
- Negative values change into (large) positive values
  - \( uy = 50323 \)
Relation between signed & unsigned

Casting from signed to unsigned

Two’s Complement

\[ x \rightarrow T2B \rightarrow B2U \rightarrow uX \]

Maintain same bit pattern

Consider B2U and B2T equations

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

and a bit pattern X; compute \( B2U(X) - B2T(X) \)

weighted sum of for bits from 0 to \( w - 2 \) cancel each other

\[ B2U(X) - B2T(X) = x_{w-1}(2^{w-1} - 2^{w-1}) = x_{w-1}2^w \]
\[ B2U(X) = x_{w-1}2^w + B2T(X) \]

If we let \( B2T(X) = x \)

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

\[
ux = \begin{cases} 
x & x \geq 0 \\
x + 2^w & x < 0 
\end{cases}
\]
Relation between signed & unsigned

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

<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>
<tr>
<td>Sum</td>
<td>-15213</td>
<td>50323</td>
</tr>
</tbody>
</table>

\[ ux = x + 2^{16} = -15213 + 65536 \]
Conversion - graphically

- 2’s Comp. → Unsigned
  - Ordering inversion
  - Negative → Big positive

2’s Comp. Range

Unsigned Range

TMax

0111

1111

TMax

TMax + 1

UMax

UMax – 1

0

0000

1000

0

2’s Comp. Range
Signed vs. unsigned in C

- **Constants**
  - By default are considered to be signed integers
  - Unsigned if have “U/u” as suffix
    
    0U, 4294967259U

- **Casting**
  - Explicit casting bet/ 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 & 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 $W = 32$

- **Expression**
<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Eval</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0 == 0U$</td>
<td><strong>unsigned</strong></td>
<td>1</td>
</tr>
<tr>
<td>$-1 &lt; 0$</td>
<td><strong>signed</strong></td>
<td>1</td>
</tr>
<tr>
<td>$-1 &lt; 0U$</td>
<td><strong>unsigned</strong></td>
<td>0</td>
</tr>
<tr>
<td>$2147483647 &gt; -2147483647-1$</td>
<td><strong>signed</strong></td>
<td>1</td>
</tr>
<tr>
<td>$2147483647U &gt; -2147483647-1$</td>
<td><strong>unsigned</strong></td>
<td>0</td>
</tr>
<tr>
<td>$2147483647 &gt; (\text{int}) 2147483648U$</td>
<td><strong>signed</strong></td>
<td>1</td>
</tr>
<tr>
<td>$-1 &gt; -2$</td>
<td><strong>signed</strong></td>
<td>1</td>
</tr>
<tr>
<td>$(\text{unsigned}) -1 &gt; -2$</td>
<td><strong>unsigned</strong></td>
<td>1</td>
</tr>
</tbody>
</table>

$2^{32}-1-1 = 2147483647$
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:
  - \( X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0 \)

\[ \begin{array}{c}
\text{k copies of MSB} \\
\end{array} \]
Sign extension example

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

```c
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>
Justification for sign extension

- Prove correctness by induction on k
  - Induction Step: extending by single bit maintains value

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

- Key observation: \(-2^w + 2^{w-1} = -2^{w-1} = \)
- 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 unsigned?

- Don’t use just because number nonzero
  - C compilers on some machines generate less efficient code
  - Easy to make mistakes (e.g., casting)
  - Few languages other than C supports unsigned integers

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

- Complement
  - Observation: \( \sim x + x = 1111\ldots1_2 = -1 \)

- Increment
  - \( \sim x + x + (-x + 1) = -1 + (-x + 1) \)
  - \( \sim x + 1 = -x \)
### Comp. & incr. examples

**x = 15213**

<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>~x</td>
<td>-15214</td>
<td>C4 92</td>
<td>11000100 10010010</td>
</tr>
<tr>
<td>~x+1</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 100100111</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 100100111</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00 00 00000000 00000000</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</td>
<td>0</td>
<td>00 00 00000000 00000000</td>
</tr>
</tbody>
</table>
Unsigned addition

- Standard addition function
  - Ignores carry output
- Implements modular arithmetic
  - \( s = U\text{Add}_w(u, v) = u + v \mod 2^w \)

Operands: \( w \) bits

True Sum: \( w + 1 \) bits

Discard Carry: \( w \) bits

\[
U\text{Add}_w(u, v) = \begin{cases} 
  u + v, & u + v < 2^w \\
  u + w - 2^w, & 2^w \leq x + y < 2^{w+1} 
\end{cases}
\]
Visualizing integer addition

- 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
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

Overflow

$U\text{Add}_4(u, v)$
Two’s complement addition

- TAdd and UAdd have identical Bit-level behavior
  - Signed vs. unsigned addition in C:
    - int s, t, u, v;
    - s = (int) ((unsigned) u + (unsigned) v);
    - t = u + v
  - Will give s == t

Operands: w bits

True Sum: w+1 bits

Discard Carry: w bits

\[
\begin{array}{c}
\text{true sum: } u + v \\
\text{TAdd}_w(u, v)
\end{array}
\]
Characterizing TAdd

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

\[
TAdd_w(u,v) = \begin{cases} 
  u + v + 2^{w-1} & u + v < TMin_w \quad \text{(NegOver)} \\
  u + v & TMin_w \leq u + v \leq TMax_w \\
  u + v - 2^{w-1} & TMax_w < u + v \quad \text{(PosOver)}
\end{cases}
\]
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 \( s = T\text{Add}(u, v) \)
  - Determine if \( s = \text{Add}(u, v) \)
  - **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)
  
  \[
  \text{ovf} = (u < 0 == v < 0) && (u < 0 != s < 0); 
  \]
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 $2^w$ bits to represent
  - 2’s complement min: $x \times y \geq (-2^{w-1}) \times (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$
    - Up to $2^{w-1}$ bits
  - 2’s complement max: $x \times y \leq (-2^{w-1})^2 = 2^{2w-2}$
    - Up to $2^w$ bits

- 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

<table>
<thead>
<tr>
<th>Operands: $w$ bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u$</td>
</tr>
<tr>
<td>$v$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>True Product: $2^w$ bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u \cdot v$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Discard $w$ bits: $w$ bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{UMult}_w(u, v)$</td>
</tr>
</tbody>
</table>

- Standard multiplication function
  - Ignores high order $w$ bits
- Implements modular arithmetic

$$\text{UMult}_w(u, v) = u \cdot v \mod 2^w$$
Security vulnerability in XDR

```c
/*
 * Illustration of code vulnerability similar to
 * that found in Sun’s XDR library
 */

void* copy_elements(void *ele_src[], int ele_cnt, size_t ele_size) {
    /*
     * Allocate buffer for ele_cnt objects, each
     * of ele_size bytes and copy from ele_src
     */
    void *result = malloc(ele_cnt * ele_size);
    if (result == NULL) return NULL; /* malloc failed */
    void *next = result;
    int i;
    for (i = 0; i < ele_cnt; i++) {
        memcpy(next, ele_src[i], ele_size); /* Copy object i to dest */
        next += ele_size; /* Move pointer to next */
    }
    return result;
}
```

Call it with `ele_cnt = 2^{20}+1` and `ele_size = 2^{12}`

Then this overflows, allocating only 4096B

... and this for loop will write over the allocated buffer, corrupting other data structures!

US-CERT Vulnerability note
http://www.kb.cert.org/vuls/id/192995
Two’s complement multiplication

- Two’s complement multiplication
  ```c
  int x, y;
  int p = x * y;
  ```
  - Compute exact product of two w-bit numbers \( x, y \)
  - Truncate result to w-bit number \( p = \text{TMultw}(x, y) \)

- Relation
  - Signed multiplication gives same bit-level result as unsigned
    ```c
    int up == (unsigned) p
    ```
Operation
- \( u << k \) gives \( u \times 2^k \)
- Both signed and unsigned

Examples
- \( u << 3 == u \times 8 \)
- \( u << 5 - u << 3 = u \times 24 \)
- Most machines \( >> \) and \( + \) much faster than \( * \) (1 to 12 cycles)
  - 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

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

<table>
<thead>
<tr>
<th>Operands:</th>
<th>Division:</th>
<th>Result:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>( x / 2^k )</td>
<td>( \text{RoundDown}(x / 2^k) )</td>
</tr>
</tbody>
</table>

<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>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>( y \gg 1)</td>
<td>-7606.5</td>
<td>E2 49</td>
<td>11100010 01001001</td>
</tr>
<tr>
<td>( y \gg 4)</td>
<td>-950.8125</td>
<td>FC 49</td>
<td>11111100 01001001</td>
</tr>
<tr>
<td>( y \gg 8)</td>
<td>-59.4257813</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: \((x<0 ? (x + (1<<k)-1) : x) >> k\)
    - Biases dividend toward 0

- Case 1: No rounding

<table>
<thead>
<tr>
<th>Dividend: ( u )</th>
<th>( +2^k - 1 )</th>
<th>( +2^k + 1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 1 ) ( \cdots ) ( 0 ) ( \cdots ) ( 0 ) ( 0 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( 0 ) ( \cdots ) ( 0 ) ( 0 ) ( 1 ) ( \cdots ) ( 1 ) ( 1 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Divisor: ( )</th>
<th>( \div 2^k )</th>
<th>( \left\lfloor \frac{u}{2^k} \right\rfloor )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 0 ) ( \cdots ) ( 0 ) ( 1 ) ( \cdots ) ( 0 ) ( 0 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( 1 ) ( \cdots ) ( 1 ) ( 1 ) ( \cdots ) ( 1 ) ( 1 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Binary Point**

**Biasing has no effect**
Correct power-of-2 divide (Cont.)

Case 2: Rounding

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

\[ \frac{x}{2^k} \]

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

**Biasing adds 1 to final result**

Divisor: \[ \frac{x}{2^k} \]

\[ \frac{x}{2^k} \]

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

Incremented by 1

Binary Point
C Puzzle answers

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

- $x < 0 \implies ((x \times 2) < 0)$ False: $T_{Min}$
- $ux >= 0$ True: $0 = U_{Min}$
- $x \& 7 == 7 \implies (x << 30) < 0$ True: $x_1 = 1$
- $ux > -1$ False: 0
- $x > y \implies -x < -y$ False: $-1, T_{Min}$
- $x \times x >= 0$ False: 30426
- $x > 0 && y > 0 \implies x + y > 0$ False: $T_{Max}, T_{Max}$
- $x >= 0 \implies -x <= 0$ True: $-T_{Max} < 0$
- $x <= 0 \implies -x >= 0$ False: $T_{Min}$