Integers

Today

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

Next time

- Floats
Checkpoint
Encoding integers in binary

- Positive integers, easy

$$B2U(X) = \sum_{i=0}^{w-1} x_i \times 2^i$$

- What about negative integers?
Encoding integers in binary

- **Idea #1: sign bit**
  - use 1 in the most significant (leftmost) bit like a minus sign
    - $3 = 0011$, $-3 = 1011$
  - intuitive, but simple arithmetic is complicated
    - $5 + -3 = 0101 + 1011 = a \text{miracle occurs} = 0010$

- **Idea #2: ones' complement**
  - flip all bits for negatives
    - $3 = 0011$, $-3 = 1100$
  - addition not too bad (just add and then add carry bit if any)
    - $5 + -3 = 0101 + 1100 = 0001 + 1 (\text{carry}) = 0010$
Encoding integers

• Both ideas lead to two representations of zero, positive and negative:
  – sign bit: 0000 and 1000
  – ones' complement: 0000 1111
  – 5 + -5 = 0101 + 1010 = 1111 = -0
Encoding integers

• Idea #3: Two’s complement
  – Informal encoding view:
  – To encode \( -N \), encode \( N \), flip all bits, add 1
    • \( 5 = 0101 \),
    • \( -5 = 1010 + 1 = 1011 \)
  – More formally, given \( w \) bits \([x_{w-1}, x_{w-2}, \ldots, x_1, x_0]\),
    • \( N = -(2^{w-1})*x_{w-1} + \sum 2^i * x_i \) for \( i \) from 0 to \( w-2 \)
    • \( 1011 = -2^3 + 3 = -8 + 3 = -5 \)

• Addition is now simple: always add, ignore overflow
  – \( 5 + -5 = 0101 + 1011 = 0000 \)

• Only one zero (why?)
• Significant bit still serves as sign bit
Encoding integers

**Unsigned**

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

**Two’s Complement**

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

**C short 2 bytes long**

```c
short int x = 15213;
short int y = -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>y</td>
<td>-15213</td>
<td>C4 93 11000100 10010011</td>
</tr>
</tbody>
</table>
**Encoding example**

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

<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>1</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>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>1</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>1</td>
</tr>
</tbody>
</table>

**Sum** | **15213** | **-15213**

---

EECS 213 Introduction to Computer Systems  
Northwestern University  

Tuesday, September 27, 2011
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

---

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 other 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
  - $UMax = 2 \cdot TMax + 1$

- C constants
  - `#include <limits.h>`
  - Declares
    - `ULONG_MAX`
    - `INT_MAX, INT_MIN`
    - `LONG_MAX, LONG_MIN`
  - Values platform-specific
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

```
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
- 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**

\[ x \rightarrow T2B \rightarrow B2U \rightarrow u_x \]

Maintain same bit pattern

Consider B2U and B2T equations

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

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 \( B2T(X) = x \)

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

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

Sum

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

- 2’s Comp. → Unsigned
  - Ordering inversion
  - Negative → Big positive
Signed and unsigned in C

- **Constants**
  - By default are considered to be signed integers
  - Unsigned if have “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
    - tx = ux;
    - uy = ty;
  - Mixed expressions – cast to *unsigned* first
    - tx + ux;
    - uy < ty;
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:
  - \( X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0 \)

\[ \begin{array}{c}
\text{k copies of MSB} \\
\hline
\end{array} \]

\[ X \rightarrow X' \]

\[ \begin{array}{c}
k \quad w \\
\hline
\end{array} \]
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>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>ix</td>
<td>15213</td>
<td>00000000 00000000 00111011 01101101</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>iy</td>
<td>-15213</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

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

- 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
Checkpoint
Negating with complement & increment

- **Claim:** Following holds for 2’s complement
  - \( \sim x + 1 == -x \)

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

- **Increment**
  - \( -1 \)
  - \( \sim x + x + (-x + 1) == -1 + (-x + 1) \)
  - \( \sim x + 1 == -x \)
### 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>( \sim x )</td>
<td>-15214</td>
<td>C4 92</td>
<td>11000100 10010010</td>
</tr>
<tr>
<td>( \sim x + 1 )</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>( y )</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \sim 0 )</td>
<td>-1</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>( \sim 0 + 1 )</td>
<td>0</td>
<td>00 00</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>
Unsigned addition

- **Standard addition function**
  - Ignores carry output

- **Implements modular arithmetic**
  - \( s = \text{UAdd}_w(u, v) = u + v \mod 2^w \)

**Operands:** \( w \) bits

**True Sum:** \( w+1 \) bits

**Discard Carry:** \( w \) bits

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

\[ 0 \rightarrow 2^w \rightarrow 2^{w+1} \]

Modular Sum

Overflow

\[ \text{Overflow} \]

\[ \text{UAdd}_4(u, v) \]

\[ u \rightarrow v \]

\[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 \]
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

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

\[
\text{TAdd}_w(u, v)
\]
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 $\text{sum} \geq 2^{w-1}$
    - Becomes negative
    - At most once
  - If $\text{sum} < -2^{w-1}$
    - Becomes positive
    - At most once
Detecting 2’s comp. overflow

- **Task**
  - Given \( s = \text{TAddw}(u, v) \)
  - Determine if \( s = \text{Addw}(u, v) \)
  - Example
    - \( \text{int } s, u, v; \)
    - \( s = u + v; \)

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

\[ \text{ovf} = (u < 0 == v < 0) \&\& (u < 0 != s < 0); \]
Checkpoint
Multiplication

- Computing exact product of \(w\)-bit numbers \(x, y\)
  - Either signed or unsigned
- Ranges
  - Unsigned: \(0 \leq x \times y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1\)
    - May need up to \(2w\) bits to represent
  - Two’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
  - Two’s complement max: \(x \times y \leq (-2^{w-1})^2 = 2^{2w-2}\)
    - Up to \(2w\) 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

- 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: \( up = ux \times uy \mod 2^w \)

- **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**
  
  \[
  \text{unsigned } \text{ux} = (\text{unsigned}) \ x;
  \]
  
  \[
  \text{unsigned } \text{uy} = (\text{unsigned}) \ y;
  \]
  
  \[
  \text{unsigned } \text{up} = \text{ux} \times \text{uy}
  \]

- **Two’s complement multiplication**
  
  \[
  \text{int } x, y;
  \]
  
  \[
  \text{int } p = x \times y;
  \]

- **Relation**
  
  – Signed multiplication gives same bit-level result as unsigned
  
  – \( \text{up} == (\text{unsigned}) \ p \)
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**
  - \( 3 \times a = a << 1 + a \)
  - Most machines shift and add much faster than multiply (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>15213</td>
<td>3B 6D</td>
</tr>
<tr>
<td>( x \ &gt;&gt; 1 )</td>
<td>7606.5</td>
<td>7606</td>
<td>1D B6</td>
</tr>
<tr>
<td>( x \ &gt;&gt; 4 )</td>
<td>950.8125</td>
<td>950</td>
<td>03 B6</td>
</tr>
<tr>
<td>( x \ &gt;&gt; 8 )</td>
<td>59.4257813</td>
<td>59</td>
<td>00 3B</td>
</tr>
</tbody>
</table>
Arithmetic Right Shift = Division by 2?

- Compare right-shifting 3-bit negative numbers to dividing by 2

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>-4</td>
</tr>
<tr>
<td>101</td>
<td>-3</td>
</tr>
<tr>
<td>110</td>
<td>-2</td>
</tr>
<tr>
<td>111</td>
<td>-1</td>
</tr>
<tr>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>010</td>
<td>2</td>
</tr>
<tr>
<td>011</td>
<td>3</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 \( u < 0 \)

### Division:

Operands:

\[
x
\]

\[
/ 2^k
\]

Division:

\[
x / 2^k
\]

Result:

\[
\text{RoundDown}(x / 2^k)
\]

<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>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 $\lfloor x / 2^k \rfloor$ (Round Toward 0)
  - Compute as $\lfloor (x + 2^k - 1) / 2^k \rfloor$
    - In C: $(x<0 ? (x + (1<<k) - 1) : x) >> k$
    - Biases dividend toward 0

- Case 1: No rounding

\[
\begin{array}{c|c}
\text{Dividend:} & 1 \cdots 0 \cdots 0 0 \\
\hline
\text{+2}^k \cdots 1 & 0 \cdots 0 0 1 \cdots 1 1 \\
\hline
\text{Divisor:} & 1 \cdots 1 1 1 \\
\hline
\text{u} / 2^k & 0 \cdots 0 \underline{1} 0 \cdots 0 0 \\
\hline
\text{[ u / 2^k ]} & 1 \cdots 1 1 1 \cdots 1 1 \\
\end{array}
\]

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

Case 2: Rounding

Dividend:

\[ \frac{x + 2^k + 1}{2^k} \]

Divisor:

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

Biasing adds 1 to final result

Incremented by 1