Today’s Topics

- Representation of integers: unsigned and signed
- Conversion, casting
- Expanding, truncating
- Addition, negation, multiplication, shifting

Encoding Integers

- 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>12345</td>
<td>30</td>
<td>00110000 00110001</td>
</tr>
<tr>
<td>y</td>
<td>-12345</td>
<td>C7</td>
<td>11001111 11000111</td>
</tr>
</tbody>
</table>

- Two’s Complement

- Sign Bit
  - For 2’s complement, most significant bit indicates sign
    - 0 for non-negative
    - 1 for negative

$-x \rightarrow \sim x + 1$
Encoding Example (Cont.)

\[
\begin{align*}
x &= 12345: 00110000 0011001 \\
y &= -12345: 11001111 11000111
\end{align*}
\]

<table>
<thead>
<tr>
<th>Weight</th>
<th>12345</th>
<th>-12345</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>32</td>
</tr>
<tr>
<td>64</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>128</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>256</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>512</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1024</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2048</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4096</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8192</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>16384</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>-32768</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Sum</td>
<td>12345</td>
<td>-12345</td>
</tr>
</tbody>
</table>

Shift Operations

- **Left shift:** \( x \ll y \)
  - Shift bit-vector \( x \) left by \( y \) positions
  - Throw away extra bits on left
  - Fill with 0s on right
  - Multiply by \( 2^y \)

- **Right shift:** \( x \gg y \)
  - Shift bit-vector \( x \) right by \( y \) positions
  - Throw away extra bits on right
  - Logical shift (for unsigned)
    - Fill with 0s on left
  - Arithmetic shift (for signed)
    - Replicate most significant bit on right
    - Maintain sign of \( x \)
  - Divide by \( 2^y \)

*Undefined behavior when \( y < 0 \) or \( y \geq \text{word\_size} \)
Using Shifts and Masks

- **Extract 2nd** most significant byte of an integer
  - First shift: \( x >> (2 \times 8) \)
  - Then mask: \(( x >> 16 ) \& 0xFF\)

<table>
<thead>
<tr>
<th>x</th>
<th>01100001</th>
<th>0110010</th>
<th>01100011</th>
<th>01100100</th>
</tr>
</thead>
<tbody>
<tr>
<td>x &gt;&gt; 16</td>
<td>00000000</td>
<td>00000000</td>
<td>01100001</td>
<td>01100010</td>
</tr>
<tr>
<td>(( x &gt;&gt; 16 ) &amp; 0xFF)</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>01100010</td>
</tr>
</tbody>
</table>

- **Extracting the sign bit**
  - \(( x >> 31 \) & 1 - need the “& 1” to clear out all other bits except LSB

- **Conditionals as Boolean expressions ( x is 0 or 1 )**
  - if \(( x) a = y \) else \( a = z \); which is the same as \( a = x \? y : z \);
  - Can be re-written as: \( a = ( x << 31 ) >> 31 \) & \( y + ( l x << 31 ) >> 31 \) & \( z \)

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>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>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
  - \(UMax = 2 * Tmax + 1\)

- **C Programming**
  - `#include <limits.h>`
  - 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
Mapping Between Signed & Unsigned

Two’s Complement → Unsigned

```
<table>
<thead>
<tr>
<th>Bits</th>
<th>Signed</th>
<th>Unsigned</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>-7</td>
<td>-7</td>
</tr>
<tr>
<td>1010</td>
<td>-6</td>
<td>-6</td>
</tr>
<tr>
<td>1011</td>
<td>-5</td>
<td>-5</td>
</tr>
<tr>
<td>1100</td>
<td>-4</td>
<td>-4</td>
</tr>
<tr>
<td>1101</td>
<td>-3</td>
<td>-3</td>
</tr>
<tr>
<td>1110</td>
<td>-2</td>
<td>-2</td>
</tr>
<tr>
<td>1111</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>
```

Maintain Same Bit Pattern

- Keep bit representations and reinterpret
Mapping Signed ↔ Unsigned

<table>
<thead>
<tr>
<th>Bits</th>
<th>Signed</th>
<th>Unsigned</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>-7</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>-6</td>
<td>10</td>
</tr>
<tr>
<td>1011</td>
<td>-5</td>
<td>11</td>
</tr>
<tr>
<td>1100</td>
<td>-4</td>
<td>12</td>
</tr>
<tr>
<td>1101</td>
<td>-3</td>
<td>13</td>
</tr>
<tr>
<td>1110</td>
<td>-2</td>
<td>14</td>
</tr>
<tr>
<td>1111</td>
<td>-1</td>
<td>15</td>
</tr>
</tbody>
</table>

Relation between Signed & Unsigned

Two's Complement

\[ x \rightarrow T2B \rightarrow T2U \rightarrow Ux \]

Maintain Same Bit Pattern

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

Large negative weight becomes Large positive weight
Conversion Visualized

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

Signed vs. Unsigned in C

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

- Casting
  - int tx, ty;
    unsigned ux, uy;
  - Explicit casting between signed & unsigned same as U2T and T2U
    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 \( W = 32 \): \( TMIN = -2,147,483,648 \) \( TMAX = 2,147,483,647 \)

<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>-2147483647-1</td>
<td>&gt;</td>
<td>signed</td>
</tr>
<tr>
<td>2147483647U</td>
<td>-2147483647-1</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>

Code Security Example (revisited)

```c
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}

#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, MSIZE);
    printf("%s\n", mybuf);
}
```
Malicious Usage

/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}

#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, -MSIZE);
    . . .
}

/* Declaration of library function memcpy */
void *memcpy(void *dest, void *src, size_t n);

Summary
Casting Signed $\leftrightarrow$ Unsigned: Basic Rules

- Bit pattern is maintained
- But reinterpreted
- Can have unexpected effects: adding or subtracting $2^w$

- Expression containing signed and unsigned int
  - int is cast to unsigned!!
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:
  \[X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0\]

![Diagram showing sign extension](image)

Sign Extension Example

```c
short int x = 12345;
int  ix = (int) x;
short int y = -12345;
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>12345</td>
<td>30 39</td>
<td>00110000 01101101</td>
</tr>
<tr>
<td>(ix)</td>
<td>12345</td>
<td>00 00 30 39</td>
<td>00000000 00000000 00110000 01101101</td>
</tr>
<tr>
<td>(y)</td>
<td>–12345</td>
<td>CF C7</td>
<td>11001111 11000111</td>
</tr>
<tr>
<td>(iy)</td>
<td>–12345</td>
<td>FF FF CF C7</td>
<td>11111111 11111111 11001111 11000111</td>
</tr>
</tbody>
</table>

- Converting from smaller to larger integer data type
- C automatically performs sign extension
Summary:
Expanding, Truncating: Basic Rules

- Expanding (e.g., short int to int)
  - Unsigned: zeros added
  - Signed: sign extension
  - Both yield expected result

- Truncating (e.g., unsigned to unsigned short)
  - Unsigned/signed: bits are truncated
  - Result reinterpreted
  - Unsigned: mod operation
  - Signed: similar to mod
    - for small numbers only – why?

Negation: Complement & Increment

- Claim: Following Holds for 2’s Complement
  \(-x + 1 == -x\)

- Complement
  - Observation: \(-x + x == 111\ldots111 == -1\)

\[
\begin{array}{r}
& 10111101 \\
+ & 0100010 \\
\hline
& 11111111
\end{array}
\]

- Complete Proof?
### Complement & Increment Examples

\[ x = 12345 \]

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345</td>
<td>3039</td>
<td>00110000 00111001</td>
</tr>
<tr>
<td>-12346</td>
<td>CF C6</td>
<td>11001111 11000110</td>
</tr>
<tr>
<td>-12345</td>
<td>CF C7</td>
<td>11001111 11000111</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00000000 00000000</td>
</tr>
<tr>
<td>-1</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>

\[ x = 0 \]

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

### Unsigned Addition

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

\[
\begin{array}{c}
\text{u} \\
+ \text{v} \\
\UAdd{w}(u, v)
\end{array}
\]

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

\[ u + v \]

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

- **Standard Addition Function**
  - Ignores carry output

- **Implements Modular Arithmetic**

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

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

$$\text{Add}_4(u, v)$$

Visualizing Unsigned Addition

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

$$\text{UAdd}_4(u, v)$$
Visualizing 2’s Complement Addition

- **Values**
  - 4-bit two’s comp.
  - Range from -8 to +7
- **Wraps around** (at most once)
  - If sum $\geq 2^{w-1}$
    - Becomes negative
  - If sum $< -2^{w-1}$
    - Becomes positive

Characterizing TAdd

- **Functionality**
  - True sum requires $w+1$ bits
  - Drop off MSB
  - Treat remaining bits as 2’s complement integer
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: \( x \times y \geq (-2^{w-1})(2^{w-1} - 1) = -2^{2w-2} + 2^{w-1} \)
    - 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

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

True Product: \( 2^w \) bits

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

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

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

- Standard Multiplication Function
  - Ignores high order \( w \) bits
- Implements Modular Arithmetic
  \[UMulti_w(u, v) = u \cdot v \mod 2^w\]
Signed Multiplication in C

Operands: w bits

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

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

True Product: 2*w bits

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

Discard w bits: w bits

\[
\begin{array}{c}
\text{TMult}_w(\text{u} , \text{v}) \\
\end{array}
\]

- **Standard Multiplication Function**
  - Ignores high order w bits
  - Some of which are different for signed vs. unsigned multiplication
  - Lower bits are the same

Power-of-2 Multiply with Shift

- **Operation**
  - \( \text{u} \ll k \) gives \( \text{u} \times 2^k \)
  - Both signed and unsigned

Operands: w bits

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

\[
\begin{array}{c}
2^k \\
\end{array}
\]

True Product: w+k bits

\[
\begin{array}{c}
\text{u} \times 2^k \\
\end{array}
\]

Discard k bits: w bits

\[
\begin{array}{c}
\text{UMult}_w(\text{u} \times 2^k) \\
\text{TMult}_w(\text{u} \times 2^k) \\
\end{array}
\]

- **Examples**
  - \( \text{u} \ll 3 \) \( \Rightarrow \) \( \text{u} \times 8 \)
  - \( \text{u} \ll 5 - \text{u} \ll 3 \) \( \Rightarrow \) \( \text{u} \times 24 \)
  - Most machines shift and add faster than multiply
    - Compiler generates this code automatically
Compiled Multiplication Code

- C compiler automatically generates shift/add code when multiplying by constant

C Function

```
int mul12(int x)
{
    return x*12;
}
```

Compiled Arithmetic Operations

- `leal (%eax,%eax,2), %eax`
- `sall $2, %eax`

```
t <- x+x*2
return t << 2;
```

Explanation

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

<table>
<thead>
<tr>
<th>Operands: ( u / 2^k )</th>
<th>Division: ( u / 2^k )</th>
<th>Result: ( \lfloor u / 2^k \rfloor )</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u \gg k$</td>
<td>( 0 \quad 0 \quad 010 \quad \cdots \quad 00 )</td>
<td>( 0 \quad 0 \quad 0 \quad \cdots \quad \cdots )</td>
</tr>
<tr>
<td>Binary Point</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>12345</td>
<td>30 39</td>
<td>00110000 00111001</td>
</tr>
<tr>
<td>( x \gg 1 )</td>
<td>6172.5</td>
<td>18 1C</td>
<td>00011000 00011100</td>
</tr>
<tr>
<td>( x \gg 4 )</td>
<td>771.5625</td>
<td>03 03</td>
<td>00000011 00000011</td>
</tr>
<tr>
<td>( x \gg 8 )</td>
<td>48.2226163</td>
<td>00 30</td>
<td>00000000 00110000</td>
</tr>
</tbody>
</table>
Compiled Unsigned Division Code

C Function

```c
unsigned udiv8(unsigned x) {
    return x/8;
}
```

Compiled Arithmetic Operations

```
shrl $3, %eax
```

Explanation

# Logical shift

```
return x >> 3;
```

- Uses logical shift for unsigned
- For Java Users
  - Logical shift written as >>>

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

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>-12345</td>
<td>CF C7</td>
<td>11001111 11000111</td>
</tr>
<tr>
<td>y &gt;&gt; 1</td>
<td>-6172.5</td>
<td>E7 E3</td>
<td>11100111 11100011</td>
</tr>
<tr>
<td>y &gt;&gt; 4</td>
<td>-771.5625</td>
<td>FC FC</td>
<td>11111100 11111100</td>
</tr>
<tr>
<td>y &gt;&gt; 8</td>
<td>-48.226563</td>
<td>FF CF</td>
<td>11111111 11001111</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 \( \lceil (x+2^k-1) / 2^k \rceil \)
  - In C: \( x + (1<<k) - 1 >> k \)
  - Biases dividend toward 0

Case 1: No rounding

Dividend: \( u \)
\[
\begin{array}{c}
1 \ldots 0 \ldots 0 \\
+2^k - 1
\end{array}
\]

Divisor: \( l / 2^k \)
\[
\begin{array}{c}
0 \ldots 0 \ldots 0 \\
\end{array}
\]

Binary Point

Biasing has no effect

Case 2: Rounding

Dividend: \( x \)
\[
\begin{array}{c}
1 \ldots 0 \ldots 0 \\
+2^k - 1
\end{array}
\]

Divisor: \( l / 2^k \)
\[
\begin{array}{c}
0 \ldots 0 \ldots 0 \\
\end{array}
\]

Biasing adds 1 to final result
Compiled Signed Division Code

C Function

```c
int idiv8(int x)
{
    return x/8;
}
```

Compiled Arithmetic Operations

```
testl %eax, %eax
    js  L4
L3:       sarl $3, %eax
            ret
L4:       addl $7, %eax
            jmp  L3
```

Explanation

```c
if x < 0
    x += 7;
    # Arithmetic shift
    return x >> 3;
```

- Uses arithmetic shift for int
- For Java Users
  - Arith. shift written as `>>`

Arithmetic: Basic Rules

- **Addition:**
  - Unsigned/signed: Normal addition followed by truncate, same operation on bit level
  - Unsigned: addition mod $2^w$
    - Mathematical addition + possible subtraction of $2^w$
  - Signed: modified addition mod $2^w$ (result in proper range)
    - Mathematical addition + possible addition or subtraction of $2^w$

- **Multiplication:**
  - Unsigned/signed: Normal multiplication followed by truncate, same operation on bit level
  - Unsigned: multiplication mod $2^w$
  - Signed: modified multiplication mod $2^w$ (result in proper range)
Arithmetic: Basic Rules

- **Left shift**
  - Unsigned/signed: multiplication by $2^k$
  - Always logical shift

- **Right shift**
  - Unsigned: logical shift, div (division + round to zero) by $2^k$
  - Signed: arithmetic shift
    - Positive numbers: div (division + round to zero) by $2^k$
    - Negative numbers: div (division + round away from zero) by $2^k$
      Use biasing to fix