Integers

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

Next time
- Floats
Integers in C

- C supports several integral data types
  - Note the `unsigned` modifier
  - Also note the asymmetric ranges

<table>
<thead>
<tr>
<th>C data type (32b)</th>
<th>Size</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>-128</td>
<td>127</td>
</tr>
<tr>
<td>unsigned char</td>
<td>1</td>
<td>0</td>
<td>255</td>
</tr>
<tr>
<td>short int</td>
<td>2</td>
<td>-32,768</td>
<td>32,767</td>
</tr>
<tr>
<td>unsigned short int</td>
<td>2</td>
<td>0</td>
<td>65,535</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>-2,147,438,648</td>
<td>2,147,438,647</td>
</tr>
<tr>
<td>unsigned int</td>
<td>4</td>
<td>0</td>
<td>4,294,967,295</td>
</tr>
<tr>
<td>long int</td>
<td>4</td>
<td>-2,147,438,648</td>
<td>2,147,438,647</td>
</tr>
<tr>
<td>unsigned long int</td>
<td>4</td>
<td>0</td>
<td>4,294,967,295</td>
</tr>
<tr>
<td>long long int</td>
<td>8</td>
<td>-9,223,372,036,854,775,808</td>
<td>9,223,372,036,854,775,807</td>
</tr>
<tr>
<td>unsigned long long int</td>
<td>8</td>
<td>0</td>
<td>18,446,744,073,709,551,615</td>
</tr>
</tbody>
</table>
Encoding integers

**Unsigned**

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

(Binary To Unsigned)

e.g. \(B2U([1011]) = 1 \cdot 2^3 + 0\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 = 11\)

- C short 2 bytes long

  ```c
  short int 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>
</tbody>
</table>
Some examples

$2^3 = 8$
$2^2 = 4$
$2^1 = 2$
$2^0 = 1$

[0101] = 5
[1011] = 11
Encoding integers

Two’s Complement

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

- e.g. \( B2T([1011]) = -1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = -5 \)
- C short 2 bytes long
  
  short int \( y = -15213 \);

<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>( 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
Some examples

- $2^3 = -8$
- $2^2 = 4$
- $2^1 = 2$
- $2^0 = 1$

- $[0101] = 5$
- $[1011] = -5$
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</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>$T_{\text{max}}$</td>
<td>32767</td>
<td>7F FF</td>
<td>01111111 11111111</td>
</tr>
<tr>
<td>$T_{\text{min}}$</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>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
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
  - Not based on a numeric perspective
  - No change in bit representation
  - Non-negative values unchanged
    - \( ux = 15213 \)
  - Negative values change into (large) positive values
    - \( uy = 50323 \)
## 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 (bijectons)**
  - Every bit pattern represents unique integer value
  - Each representable integer has unique bit encoding

- **Hence, well defined inverses**
  - \( U2B(x) = B2U^{-1}(x) \)
    - Bit pattern for unsigned integer
  - \( T2B(x) = B2T^{-1}(x) \)
    - Bit pattern for two’s comp integer
Relation between signed & unsigned

Casting from signed to unsigned

Two’s Complement

| x | T2B | B2U | 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 \( B2T(X) = x \)

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

Sign bit

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

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

<table>
<thead>
<tr>
<th>Weight</th>
<th>15213</th>
<th>-15213</th>
<th>50323</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>01</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>01</td>
<td>16</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>64</td>
<td>1</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>128</td>
<td>0</td>
<td>01</td>
<td>128</td>
</tr>
<tr>
<td>256</td>
<td>1</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>512</td>
<td>1</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>1024</td>
<td>0</td>
<td>01</td>
<td>1024</td>
</tr>
<tr>
<td>2048</td>
<td>1</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>4096</td>
<td>1</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>8192</td>
<td>1</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>16384</td>
<td>0</td>
<td>01</td>
<td>16384</td>
</tr>
<tr>
<td>-/+32768</td>
<td>0</td>
<td>01</td>
<td>-32768</td>
</tr>
<tr>
<td>Sum</td>
<td>15213</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
Signed vs. unsigned in C

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

- **Casting**
  - Explicit casting between signed & unsigned same as U2T and T2U
    ```c
    int tx, ty;
    unsigned ux, uy;
    tx = (int) ux;
    uy = (unsigned) ty;
    ```
  - Implicit casting also occurs via assignments & procedure calls
    ```c
    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>unsigned</td>
<td>1</td>
</tr>
<tr>
<td>$-1 &lt; 0$</td>
<td>signed</td>
<td>1</td>
</tr>
<tr>
<td>$-1 &lt; 0U$</td>
<td>unsigned</td>
<td>0</td>
</tr>
<tr>
<td>$2^{32-1} - 1 = 2147483647$</td>
<td>signed</td>
<td>1</td>
</tr>
<tr>
<td>$(unsigned) -1 &gt; -2$</td>
<td>unsigned</td>
<td>1</td>
</tr>
</tbody>
</table>
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 of sign extension]
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></td>
<td></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></td>
<td></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:
  - \( X \quad -2^{w-1} x_{w-1} \)
  - \( X' \quad -2^w x_{w-1} + 2^{w-1} x_{w-1} = -2^{w-1} x_{w-1} \)
Why should I use unsigned?

- Don’t use just because number nonzero
  - C compilers on some machines generate less efficient code
  - 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: \( \sim x + x == 1111...11_2 == -1 \)

- **Increment**
  - \( \sim x + \bar{x} + (\sim x + 1) == \bar{-1} + (-x + 1) \)
  - \( \sim x + 1 == -x \)
### 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</td>
<td>6D 00111011 01101101</td>
</tr>
<tr>
<td>~x</td>
<td>-15214</td>
<td>C4</td>
<td>92 11000100 10010010</td>
</tr>
<tr>
<td>~x+1</td>
<td>-15213</td>
<td>C4</td>
<td>93 11000100 10010011</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>C4</td>
<td>93 11000100 10010011</td>
</tr>
</tbody>
</table>

**0**

<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>00</td>
<td>00000000 00000000</td>
</tr>
<tr>
<td>~0</td>
<td>-1</td>
<td>FF</td>
<td>FF 11111111 11111111</td>
</tr>
<tr>
<td>~0+1</td>
<td>0</td>
<td>00</td>
<td>00000000 00000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Unsigned addition

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

- **Implements modular arithmetic**
  - \( s = UAdd_w(u, v) = u + v \mod 2^w \)

<table>
<thead>
<tr>
<th>Operands: ( w ) bits</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( u )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( v )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( u + v )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( UAdd_w(u, v) )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

\[
UAdd_w(u, v) = \begin{cases} 
    u + v, & u + v < 2^w \\
    u + w - 2^w, & 2^w \leq u + v < 2^{w+1}
\end{cases}
\]
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

Overflow

Modular Sum

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

Overflow
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

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

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

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

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

- **Wraps Around**
  - If sum $\geq 2^{w-1}$
    - Becomes negative
    - At most once
  - If sum $< -2^{w-1}$
    - Becomes positive
    - At most once

![3D diagram showing TAdd₄(u, v) with标NegOver and PosOver labels.](image)
Detecting 2’s comp. overflow

- **Task**
  - Given \( s = TAddw(u, v) \)
  - Determine if \( s = Addw(u, v) \)
  - **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)

\[
\text{ovf} = (u < 0 == v < 0) \&\& (u < 0 != s < 0);
\]
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 to represent
  - 2’s complement min: $x \times y \geq (-2^{w-1})(2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$
    - Up to $2^{w-1}$ bits
  - 2’s complement max: $x \times y \leq (-2^{w-1})^2 = 2^{2w-2}$
    - Up to $2w$ 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
  \[
  \text{UMult}_w(u, v) = u \cdot v \mod 2^w
  \]
Security vulnerability in XDR

/*
 * 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 = TMultw(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 \( u \times 2^k \)
  - Both signed and unsigned

  
  Operands: \( w \) bits

  
  \[
  \begin{array}{c}
  \text{True Product: } w+k \text{ bits} \\
  \text{Discard } k \text{ bits: } w \text{ bits}
  \end{array}
  \]

- **Examples**
  - \( u << 3 == u \times 8 \)
  - \( u << 5 - u << 3 = u \times 24 \)
  - Most machines >> and + much faster than \( \times \) (1 to 12 cycles)
    - Compiler generates this code automatically
Unsigned power-of-2 divide with shift

- Quotient of unsigned by power of 2
  - \( u \gg k \) gives \( \lfloor u \div 2^k \rfloor \)
  - 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 \gg 1)</td>
<td>7606.5</td>
<td>7606</td>
<td>1D B6</td>
</tr>
<tr>
<td>( x \gg 4)</td>
<td>950.8125</td>
<td>950</td>
<td>03 B6</td>
</tr>
<tr>
<td>( x \gg 8)</td>
<td>59.4257813</td>
<td>59</td>
<td>00 3B</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 $x < 0$

### 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>$y &gt;&gt; 1$</td>
<td>-7606.5</td>
<td>E2 49</td>
<td>11100010 01001001</td>
</tr>
<tr>
<td>$y &gt;&gt; 4$</td>
<td>-950.8125</td>
<td>FC 49</td>
<td>11111100 01001001</td>
</tr>
<tr>
<td>$y &gt;&gt; 8$</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**

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

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

**Case 2: Rounding**

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

**Divisor:**
\[
/ 2^k
\]

**Biasing adds 1 to final result**
Visualizing integer addition

- **Integer addition**
  - 4-bit integers u, v
  - Compute true sum Add4(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
Encoding example

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

<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>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>0</td>
</tr>
<tr>
<td>64</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>128</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>256</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>512</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1024</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2048</td>
<td>1</td>
<td>0</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>0</td>
</tr>
<tr>
<td>-/+32768</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Sum</td>
<td>15213</td>
<td>-15213</td>
</tr>
</tbody>
</table>
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;
```

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

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

- $x < 0 \quad \Rightarrow \quad ((x*2) < 0)$ \quad False: $TMin$
- $ux >= 0$ \quad True: $0 = UMin$
- $x & 7 == 7 \quad \Rightarrow \quad (x<<30) < 0$ \quad True: $x_1 = 1$
- $ux > -1$ \quad False: $0$
- $x > y \quad \Rightarrow \quad -x < -y$ \quad False: $-1, TMin$
- $x * x >= 0$ \quad False: $30426$
- $x > 0 && y > 0 \quad \Rightarrow \quad x + y > 0$ \quad False: $TMax, TMax$
- $x >= 0 \quad \Rightarrow \quad -x <= 0$ \quad True: $-TMax < 0$
- $x <= 0 \quad \Rightarrow \quad -x >= 0$ \quad False: $TMin$