Ints are not Integers

BIL 341–Systems Programming
Fall 2012

Instructors:
Aykut & Erkut Erdem

Acknowledgement: The course slides are adapted from the slides prepared by R.E. Bryant, D.R. O’Hallaron, G. Kesden and Markus Püschel of Carnegie-Mellon Univ.
Today: Integers

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

- Summary
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>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 (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>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>1</td>
</tr>
<tr>
<td>-32768</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

\[
\text{Sum} \quad 15213 \quad -15213
\]
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>Values</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**

- $|T_{min}| = T_{max} + 1$
  - Asymmetric range
- $U_{max} = 2 \times T_{max} + 1$

**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

- $\Rightarrow$ **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
Today: Integers

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

- Summary
Mappings between unsigned and two’s complement numbers:
keep bit representations and reinterpert
### 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>

Diagram:
- **T2U** (Signed to Unsigned)
- **U2T** (Unsigned to Signed)
# Mapping Signed $\leftrightarrow$ 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>

$\quad = \quad +/\ - \ 16$
Relation between Signed & Unsigned

Two’s Complement

Maintain Same Bit Pattern

Unsigned

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

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**
  - 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 and procedure calls
    ```c
    tx = ux;
    uy = ty;
    ```
Casting Surprises

- **Expression Evaluation**
  - If there is a mix of unsigned and signed in single expression, *signed values implicitly cast to unsigned*
  - Including comparison operations `<`, `>`, `==`, `<=`, `>=`
  - Examples for $W = 32$: \( \text{TMIN} = -2,147,483,648 \), \( \text{TMAX} = 2,147,483,647 \)

<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><code>==</code></td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td><code>&lt;</code></td>
<td>signed</td>
</tr>
<tr>
<td>-1</td>
<td>0U</td>
<td><code>&gt;</code></td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>-2147483647-1</td>
<td><code>&gt;</code></td>
<td>signed</td>
</tr>
<tr>
<td>2147483647U</td>
<td>-2147483647-1</td>
<td><code>&lt;</code></td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>-2</td>
<td><code>&gt;</code></td>
<td>signed</td>
</tr>
<tr>
<td>(unsigned)-1</td>
<td>-2</td>
<td><code>&gt;</code></td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>2147483648U</td>
<td><code>&lt;</code></td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>(int) 2147483648U</td>
<td><code>&gt;</code></td>
<td>signed</td>
</tr>
</tbody>
</table>
Code Security Example

/* 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;
}

- Similar to code found in FreeBSD’s implementation of getpeername
- There are legions of smart people trying to find vulnerabilities in programs
Typical 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);
    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);
  . . .
}

A typical definition of size_t (in stdio.h):

typedef unsigned long int size_t;
Summary

Casting Signed ↔ 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!!
Today: Integers

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

- Summary
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:
  - $X' = x_{w-1}, ..., x_{w-1}, x_{w-1}, x_{w-2}, ..., x_0$

![Diagram showing sign extension process]
Sign Extension Example

```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>

- 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 (note that sign can change!)
  - For small numbers yields expected behaviour
Today: Integers

- **Integers**
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting

- **Summary**
Negation: Complement & Increment

- **Claim:** Following holds for 2’s Complement
  \[ \sim x + 1 \equiv -x \]

- **Complement**
  - Observation: \( \sim x + x \equiv 1111\ldots111 \equiv -1 \)

  \[
  \begin{array}{c}
  x \\
  + \sim x \\
  \hline
  -1
  \end{array}
  \]

  \[
  \begin{array}{cccccccc}
  & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\
  \times & \hline
  & \hline
  & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\
  \hline
  & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1
  \end{array}
  \]

- **Complete Proof?**
Complement & Increment 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>

\( x = 0 \)

<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>( \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</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>
Unsigned Addition

Operands: \( w \) bits

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

True Sum: \( w+1 \) bits

\[
\frac{u + v \text{ mod } 2^w}{\text{Discard Carry: } w \text{ bits}}
\]

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

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

- **Implements Modular Arithmetic**
  \[
  s = \text{UAdd}_w(u, v) = u + v \text{ mod } 2^w
  \]

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

![Add4(u, v)](image-url)
Visualizing Unsigned Addition

Wraps Around
- If true sum ≥ 2^w
- At most once

True Sum
- 2^{w+1}
- 2^w
- 0

Modular Sum

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

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

True Sum: \( w+1 \) bits

Discard Carry: \( w \) bits

\( \text{TAdd}_w(u, v) \)

- **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 \( s == t \)
TAdd Overflow

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

**True Sum**

<table>
<thead>
<tr>
<th>True Sum</th>
<th>TAdd Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 111...1</td>
<td>2^{w-1} PosOver 011...1</td>
</tr>
<tr>
<td>0 100...0</td>
<td>2^{w-1} 000...0</td>
</tr>
<tr>
<td>0 000...0</td>
<td>0 10...0</td>
</tr>
<tr>
<td>1 011...1</td>
<td>(-2^{w-1}-1) NegOver 100...0</td>
</tr>
<tr>
<td>1 000...0</td>
<td>(-2^w)</td>
</tr>
</tbody>
</table>
Visualizing 2’s Complement 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
Characterizing TAdd

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

\[
TAdd(u, v) = \begin{cases} 
  u + v + 2^w & u + v < TMin_w \quad \text{(NegOver)} \\
  u + v & TMin_w \leq u + v \leq TMax_w \\
  u + v - 2^w & TMax_w < u + v \quad \text{(PosOver)} 
\end{cases}
\]
Mathematical Properties of TAdd

- **Isomorphic Group to unsigneds with UAdd**
  - $\text{TAdd}_w(u, v) = \text{U2T(UAdd}_w(\text{T2U}(u), \text{T2U}(v)))$
  - 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

\[ T\text{Comp}_w(u) = \begin{cases} 
  -u & u \neq \text{TMin}_w \\
  \text{TMin}_w & u = \text{TMin}_w 
\end{cases} \]
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

True Product: $2^w$ bits

Discard $w$ bits: $w$ bits

- **Standard Multiplication Function**
  - Ignores high order $w$ bits

- **Implements Modular Arithmetic**

  $$\text{UMult}_w(u, v) = u \cdot v \mod 2^w$$
Code Security Example #2

- **SUN XDR library**
  - Widely used library for transferring data between machines

```c
void* copy_elements(void *ele_src[], int ele_cnt, size_t ele_size);
```

```
malloc(ele_cnt * ele_size)
```
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 locations designated by ele_src
     */
    void *result = malloc(ele_cnt * ele_size);
    if (result == NULL)
        /* malloc failed */
        return NULL;
    void *next = result;
    int i;
    for (i = 0; i < ele_cnt; i++) {
        /* Copy object i to destination */
        memcpy(next, ele_src[i], ele_size);
        /* Move pointer to next memory region */
        next += ele_size;
    }
    return result;
}
XDR Vulnerability

```c
malloc(ele_cnt * ele_size)
```

- **What if:**
  - `ele_cnt` = $2^{20} + 1$
  - `ele_size` = 4096 = $2^{12}$
  - Allocation = ??

- **How can I make this function secure?**
Signed Multiplication in C

Operands: \( w \) bits

True Product: \( 2w \) bits

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

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

<table>
<thead>
<tr>
<th>$u$</th>
<th>0</th>
<th>⋯</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>⋯</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^k$</td>
<td>0</td>
<td>⋯</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>⋯</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Operands: $w$ bits

True Product: $w+k$ bits

Discard $k$ bits: $w$ bits

- **Examples**
  - $u << 3 == u \times 8$
  - $u << 5 - u << 3 == u \times 24$
  - Most machines shift and add faster than multiply
    - Compiler generates this code automatically
Compiled Multiplication Code

C Function

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

Compiled Arithmetic Operations

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

Explanation

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

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

## Operands:

<table>
<thead>
<tr>
<th>$u$</th>
<th>$2^k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[\ldots \ldots]$</td>
<td>$[0 \ldots 010 \ldots 00]$</td>
</tr>
</tbody>
</table>

## Division:

<table>
<thead>
<tr>
<th>$u / 2^k$</th>
<th>$\lfloor u / 2^k \rfloor$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[\ldots \ldots]$</td>
<td>$[0 \ldots 00 \ldots]$</td>
</tr>
</tbody>
</table>

## Result:

<table>
<thead>
<tr>
<th>$\lfloor u / 2^k \rfloor$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[0 \ldots 00 \ldots]$</td>
</tr>
</tbody>
</table>

## Table:

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

![Diagram](image)

<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: \((x + (1<<k)-1) >> k\)
    - Biases dividend toward 0

Case 1: No rounding

Dividend: \( \frac{u}{2^k} \)

<table>
<thead>
<tr>
<th>Dividend:</th>
<th>Divisor:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(u)</td>
<td>(2^k)</td>
</tr>
<tr>
<td>(u \div 2^k)</td>
<td>}</td>
</tr>
</tbody>
</table>

Biases have no effect
Correct Power-of-2 Divide (Cont.)

Case 2: Rounding

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

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

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

```
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

- Unsigned ints, 2’s complement ints are isomorphic rings: isomorphism = casting

- 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
Today: Integers

- Representation: unsigned and signed
- Conversion, casting
- Expanding, truncating
- Addition, negation, multiplication, shifting
- Summary
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 (Mathematical) Integer Arithmetic**
  - Both are rings
  - Integers obey ordering properties, e.g.,
    - \( u > 0 \) \( \Rightarrow \) \( u + v > v \)
    - \( u > 0, v > 0 \) \( \Rightarrow \) \( u \cdot v > 0 \)
  - These properties are not obeyed by two’s comp. arithmetic
    - \( T\text{Max} + 1 = T\text{Min} \)
    - \( 15213 \times 30426 = -10030 \) (16-bit words)
Why Should I Use Unsigned?

- *Don’t Use Just Because Number Nonnegative*
  - Easy to make mistakes
    ```c
    unsigned i;
    for (i = cnt-2; i >= 0; i--)
        a[i] += a[i+1];
    ```
  - Can be very subtle
    ```c
    #define DELTA sizeof(int)
    int i;
    for (i = CNT; i-DELTA >= 0; i-= DELTA)
        . . .
    ```

- *Do Use When Performing Modular Arithmetic*
  - Multiprecision arithmetic

- *Do Use When Using Bits to Represent Sets*
  - Logical right shift, no sign extension
Integer C Puzzles

• Assume machine with 32 bit word size, two’s comp. integers

• \( x < 0 \) \( \Rightarrow \) \((x*2) < 0\)
• \( ux >= 0 \)
• \( x \& 7 == 7 \) \( \Rightarrow \) \((x<<30) < 0\)
• \( ux > -1 \)
• \( x > y \) \( \Rightarrow \) \(-x < -y\)
• \( x * x >= 0 \)
• \( x > 0 \&\& y > 0 \) \( \Rightarrow \) \( x + y > 0 \)
• \( x >= 0 \) \( \Rightarrow \) \(-x <= 0\)
• \( x <= 0 \) \( \Rightarrow \) \(-x >= 0\)
• \((x|-x)>>31 == -1\)
• \( ux >> 3 == ux/8\)
• \( x >> 3 == x/8\)
• \( x \& (x-1) != 0\)

Initialization

```c
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```