Representing Integers

Last Time: Bits & Bytes

- Bits, Bytes, Words
- Decimal, binary, hexadecimal representation
- Virtual memory space, addressing, byte ordering
- Boolean algebra
- Bit versus logical operations in C
Today: 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><code>x</code></td>
<td>15213</td>
<td>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td><code>y</code></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.)

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

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

Numeric Ranges

- **Unsigned Values**
  - \( U_{Min} = 0 \)
    - 000...0
  - \( U_{Max} = 2^w - 1 \)
    - 111...1

- **Two's Complement Values**
  - \( T_{Min} = -2^{w-1} \)
    - 100...0
  - \( T_{Max} = 2^{w-1} - 1 \)
    - 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**
  - $|TMin| = Tmax + 1$
  - Asymmetric range
  - $U_{Max} = 2 \times Tmax + 1$

- **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
  - $U_{2B}(x) = B2U^{-1}(x)$
    - Bit pattern for unsigned integer
  - $T_{2B}(x) = B2T^{-1}(x)$
    - Bit pattern for two’s comp integer
Today: Integers

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

Mapping Between Signed & Unsigned

- Mappings been unsigned and two's complement numbers:
  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>

The mapping process can be represented as:

- **T2U**: Signed to Unsigned
- **U2T**: Unsigned to Signed

The signed values are shifted by 16 to convert from unsigned to signed, and the unsigned values are shifted by -16 to convert from signed to unsigned.
Relation between Signed & Unsigned

Two’s Complement

\[ x \rightarrow \text{T2B} \rightarrow \text{T2U} \rightarrow \text{B2U} \rightarrow u_x \]

Maintain Same Bit Pattern

Large negative weight becomes Large positive weight

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

Conversion Visualized

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

2’s Complement Range

\[ T_{\text{Max}} \]

\[ T_{\text{Min}} \]

\[ U_{\text{Max}} \]

\[ U_{\text{Max}} - 1 \]

\[ T_{\text{Max}} + 1 \]

\[ T_{\text{Max}} \]

Unsigned Range

\[ 0 \]

\[ -2 \]

\[ -1 \]

\[ 0 \]

\[ U_{\text{Max}} \]

\[ U_{\text{Max}} - 1 \]
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;
    ```

- 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 = \(2,147,483,648\)

<table>
<thead>
<tr>
<th>( \text{Constant}_1 )</th>
<th>( \text{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

```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;
}
```

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

---

Typical Usage

```c
/* Kernel memory region holding user-accessible data */
#define MSIZE 528

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

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

/* 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);
    . . .
}
```

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

- 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:
    - $X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0$

![Sign Extension Diagram]
Sign Extension Example

```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>3B 6D 00111011 01101101</td>
</tr>
<tr>
<td>ix</td>
<td>15213</td>
<td>00 00 3B 6D 00000000 00000000 00111011 01101101</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>C4 93 11000100 10010011</td>
</tr>
<tr>
<td>iy</td>
<td>-15213</td>
<td>FF FF C4 93 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
  - For small numbers yields expected behavior
Today: Integers

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

Negation: Complement & Increment

- Claim: Following Holds for 2’s Complement
  \(-x + 1 = -x\)
- Complement
  - Observation: \(-x + x = 111\ldots111 = -1\)

  \[
  \begin{array}{c}
  \times & 10011101 \\
  \hline
  \text{~}x & 01100010 \\
  \hline
  \text{-1} & 11111111
  \end{array}
  \]

- 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>(-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 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>(-0)</td>
<td>-1</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>(-0+1)</td>
<td>0</td>
<td>00</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>

Unsigned Addition

Operands: \(w\) bits

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

True Sum: \(w+1\) bits

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

Discard Carry: \(w\) bits

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

- **Standard Addition Function**
  - Ignores carry output
- **Implements Modular Arithmetic**
  \[
  S = \text{UAdd}_w(u, v) = u + v \mod 2^w
  \]

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

Visualizing Unsigned Addition

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

![UAdd4(u, v)](image)
Two’s Complement Addition

Operands: \( w \) bits

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

True Sum: \( w+1 \) bits

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

Discard Carry: \( w \) bits

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

- **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**

- \( 0 \) \( 1 \) \( 1 \) \( 1 \) \( \ldots \) \( 1 \)
- \( 0 \) \( 1 \) \( 0 \) \( 0 \) \( \ldots \) \( 0 \)
- \( 0 \) \( 0 \) \( 0 \) \( \ldots \) \( 0 \)
- \( 1 \) \( 0 \) \( 1 \) \( \ldots \) \( 1 \)
- \( 1 \) \( 0 \) \( 0 \) \( \ldots \) \( 0 \)

- \( 0 \) \( 1 \) \( 1 \) \( \ldots \) \( 1 \)
- \( 2^w - 1 \)
- \( 0 \)
- \( -2^{w-1} \)
- \( -2^w \)

**TAdd Result**

- \( 011\ldots1 \)
- \( 000\ldots0 \)
- \( 000\ldots0 \)
- \( 100\ldots0 \)
- \( 10\ldots1 \)

- \( \text{PosOver} \)
- \( \text{NegOver} \)
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

\[
TAdd_w(u, v) = \begin{cases} 
    u + v + 2^w & u + v < TMin_w \
    u + v & TMin_w \leq u + v \leq TMax_w \
    u + v - 2^w & TMax_w < u + v
\end{cases}
\]
Multiplication

- **Computing Exact Product of** $w$-bit numbers $x, y$
  - Either signed or unsigned

- **Ranges**
  - Unsigned: $0 \leq x \cdot y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
    - Up to 2w bits
  - Two’s complement min: $x \cdot y \geq (-2^{w-1}) \cdot (2^{w-1} - 1) = -2^{2w-2} + 2^w$
    - Up to 2w–1 bits
  - Two’s complement max: $x \cdot y \leq (-2^{w-1})^2 = 2^{2w-2}$
    - 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**

Operand: $w$ bits

$U \cdot V$

True Product: $2w$ bits

$\text{UMult}_w(u, v)$

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);
```

```c
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 Code
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

\[
\begin{array}{c}
\text{True Product: } 2^w \text{ bits} \\
\end{array}
\]

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

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

<table>
<thead>
<tr>
<th>Operands: ( w ) bits</th>
<th>( u \ll k \rightarrow 0 \ldots 110 \ldots 00 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>True Product: ( w+k ) bits</td>
<td>( u \times 2^k \rightarrow 0 \ldots 110 \ldots 00 )</td>
</tr>
<tr>
<td>Discard ( k ) bits: ( w ) bits</td>
<td>( \text{UMult}_w(u, 2^k) \rightarrow 0 \ldots 0 \ldots 0 )</td>
</tr>
</tbody>
</table>

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

Compiled Multiplication Code

**C Function**

```c
int mull2(int x) {
    return x*12;
}
```

**Compiled Arithmetic Operations**

```
leal (%eax,%eax,2), %eax
sal $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:

\[
\begin{array}{c}
\text{Operand:} \\
\hline
u & \cdots & \cdots \\
\hline
\end{array}
\]

\[
\begin{array}{c}
\text{Divisor:} \\
\hline
2^k & \cdots & \cdots \\
\hline
\end{array}
\]

### Division:

\[
\begin{array}{c}
\text{Result:} \\
\hline
\lfloor u / 2^k \rfloor & \cdots & \cdots \\
\hline
\end{array}
\]

<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>00110011 01101101</td>
</tr>
<tr>
<td>( x &gt;&gt; 1 )</td>
<td>7606.5</td>
<td>1D B6</td>
<td>00011110 10110110</td>
</tr>
<tr>
<td>( x &gt;&gt; 4 )</td>
<td>950,8125</td>
<td>03 B6</td>
<td>00000001 10110110</td>
</tr>
<tr>
<td>( x &gt;&gt; 8 )</td>
<td>59,4257813</td>
<td>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 \( u < 0 \)

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

<table>
<thead>
<tr>
<th>( x )</th>
<th>Binary Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \ldots \ldots \ldots \quad 0 \quad \ldots \ldots \ldots )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( 2^k )</th>
<th>Binary Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \ldots \ldots \ldots \quad 0 \quad \ldots \ldots \ldots )</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>( y \gg 1 )</td>
<td>-15213</td>
<td>-15213</td>
<td>C4 93</td>
</tr>
<tr>
<td>( y \gg 4 )</td>
<td>-7606.5</td>
<td>-7607</td>
<td>E2 49</td>
</tr>
<tr>
<td>( y \gg 8 )</td>
<td>-950.8125</td>
<td>-951</td>
<td>FC 49</td>
</tr>
<tr>
<td>( y \gg 8 )</td>
<td>-59.4257813</td>
<td>-60</td>
<td>FF C4</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: \( (x + (1<<k) - 1) \gg k \)
    - Biases dividend toward 0

**Case 1: No rounding**

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

**Biasing has no effect**
Correct Power-of-2 Divide (Cont.)

Case 2: Rounding

\[
\begin{array}{c}
\text{Dividend:} \\
\begin{array}{c}
\text{Divisor:} \\
\end{array}
\end{array}
\]

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

\[
\begin{array}{c}
k \\
0 \ldots 0111 \ldots 11
\end{array}
\]

\[
\begin{array}{c}
k \\
0 \ldots 0111 \ldots 10
\end{array}
\]

Incremented by 1

Binary Point

Biasing adds 1 to final result

Compiled Signed Division Code

C Function

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

Compiled Arithmetic Operations

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

Explanation

```assembly
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
Today: Integers

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

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

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

#### Initialization

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