Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
- Summary
- Representations in memory, pointers, strings

Everything is bits

- Each bit is 0 or 1
- By encoding/interpreting sets of bits in various ways
  - Computers determine what to do (instructions)
  - ... and represent and manipulate numbers, sets, strings, etc...
- Why bits? Electronic Implementation
  - Easy to store with bistable elements
  - Reliably transmitted on noisy and inaccurate wires

For example, can count in binary

- Base 2 Number Representation
  - Represent 15213 as 111011011011012
  - Represent 1.20 as 1.0011001100110011[0011]2
  - Represent 1.5213 \times 10^4 as 1.11011011011012 \times 2^{13}

Encoding Byte Values

- Byte = 8 bits
  - Binary 00000000 to 11111111
  - Decimal: 0 to 255
  - Hexadecimal: 00 to FF
    - Base 16 number representation
    - Use characters '0' to '9' and 'A' to 'F'
    - Write FA1D37Bu in C as
      - 0xFA1D37B
      - 0xfa1d37b

Aside: ASCII table

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>Octal</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>Decimal</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>Hex</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td>ASCII</td>
<td>@</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
<td>O</td>
</tr>
<tr>
<td></td>
<td>P</td>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>V</td>
<td>W</td>
<td>X</td>
<td>Y</td>
<td>Z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>:</td>
<td>:</td>
<td>:</td>
<td>=</td>
<td>&gt;</td>
<td>?</td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
<td>l</td>
<td>k</td>
<td>m</td>
<td>n</td>
<td>o</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>p</td>
<td>q</td>
<td>r</td>
<td>s</td>
<td>t</td>
<td>u</td>
<td>v</td>
<td>w</td>
<td>x</td>
<td>y</td>
<td>z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example Data Representations

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Typical 32-bit</th>
<th>Typical 64-bit</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>4</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>–</td>
<td>–</td>
<td>10/16</td>
</tr>
<tr>
<td>pointer</td>
<td>4</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

Today: Bits, Bytes, and Integers
- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
- Representations in memory, pointers, strings

Boolean Algebra
- Developed by George Boole in 19th Century
  - Algebraic representation of logic
    - Encode "True" as 1 and "False" as 0
- And (math: ∧)
  - \( A \land B = 1 \) when both \( A=1 \) and \( B=1 \)
  - \( \begin{array}{c|c|c}
  \hline
  \& & 0 & 1 \\
  0 & 0 & 0 \\
  1 & 0 & 1 \\
  \hline
  \end{array} \)
- Or (math: ∨)
  - \( A \lor B = 1 \) when either \( A=1 \) or \( B=1 \)
  - \( \begin{array}{c|c|c}
  \hline
  | & 0 & 1 \\
  0 & 0 & 1 \\
  1 & 1 & 1 \\
  \hline
  \end{array} \)
- Not (math: ¬)
  - \( \lnot A = 1 \) when \( A=0 \)
  - \( \lnot A \land B = 1 \) when either \( A=1 \) or \( B=1 \), but not both
  - \( \begin{array}{c|c|c}
  \hline
  \neg & 0 & 1 \\
  0 & 0 & 1 \\
  1 & 1 & 0 \\
  \hline
  \end{array} \)

General Boolean Algebras
- Operate on Bit Vectors
  - Operations applied bitwise
  - \( \begin{array}{c}
  \hline
  \& & 01101001 & 01101001 & 01101001 & 01101001 & 01101001 & 01101001 & 01101001 & 01101001 \\
  \hline
  01010101 & 00111101 & 00111100 & 10101010 & 01000001 & 01111101 & 00111100 & 10101010 \\
  \hline
  \end{array} \)
- All of the Properties of Boolean Algebra Apply

Example: Representing & Manipulating Sets
- Representation
  - Width \( w \) bit vector represents subsets of \( \{0, ..., w-1\} \)
  - \( a_j = 1 \) if \( j \in A \)
  - \( \begin{array}{c|c}
  \hline
  01101001 & \{0, 3, 5, 6\} \\
  76543210 & \{0, 2, 4, 6\} \\
  \hline
  \end{array} \)
- Operations
  - \& Intersection
  - | Union
  - ^ Symmetric difference
  - ~ Complement
  - \( \begin{array}{c|c|c|c|c}
  \hline
  01000001 & 01111101 & 10101010 & 10000000 & 00111111 \\
  \hline
  \end{array} \)

Bit-Level Operations in C
- Operations \&, |, ~, ^ Available in C
  - Apply to any "integral" data type
  - long, int, short, char, unsigned
  - View arguments as bit vectors
  - Arguments applied bit-wise
- Examples (Char data type)
  - \( \sim 0x41 \rightarrow 0xBE \)
  - \( \sim 0x00 \rightarrow 0xFF \)
  - \( \sim 0x69 \& 0x55 \rightarrow 0x41 \)
  - \( \sim 0x10100101 \rightarrow 01111111 \)
  - \( \sim 0x01010101 \rightarrow 01111111 \)
  - \( 0x69 \& 0x55 \rightarrow 0x41 \)
  - \( 0x10100101 \& 01111111 \rightarrow 01100001 \)
  - \( 0x69 \& 0x55 \rightarrow 0x41 \)
  - \( 0x10100101 \& 01111111 \rightarrow 01100001 \)
Contrast: Logic Operations in C

- **Contrast to Logical Operators**
  - &&, ||, !
  - View 0 as "False"
  - Anything nonzero as "True"
  - Always return 0 or 1
  - Early termination (AKA "short-circuit evaluation")

- **Examples (char data type)**
  - !0x41 → 0x00
  - !0x00 → 0x01
  - !!0x41 → 0x01
  - 0x69 && 0x55 → 0x01
  - 0x69 || 0x55 → 0x01
  - p && *p (avoids null pointer access)

Shift Operations

- **Left Shift:** x << y
  - Shift bit-vector x left y positions
  - Throw away extra bits on left
  - Fill with 0's on right

- **Right Shift:** x >> y
  - Shift bit-vector x right y positions
  - Throw away extra bits on right
  - Logical shift
  - Fill with 0's on left
  - Arithmetic shift
  - Replicate most significant bit on left

- **Undefined Behavior**
  - Shift amount < 0 or ≥ word size

Interlude: Self-introduction

- **Name:** Stephen McCamant
- **Have been teaching CS at UMN since 2012**
- **Primary research interest:** binary analysis for security
- **Office:** 4-225E Keller Hall (upstairs from this room)
- **Office hours:**
  - Mondays 10-11am
  - Wednesdays 2-3pm

Interlude: ChimeIn

- I’ll periodically break up lectures with opportunities for you to think about the material and maybe talk with the people sitting next to you
- To anonymously submit answers, we’ll use ChimeIn
- If you have a laptop with you, please go to: [http://chimein.cla.umn.edu/](http://chimein.cla.umn.edu/)
- And answer today’s (non-CS) question
- (Can also supposedly set up to answer with a cell-phone)

Homework turn-in process

- **For full credit:** turn in at the beginning of class on the due date (green box)
  - On-time = 3:35pm, or when I start lecturing, whichever is later
  - Yes, this means you have to come to class on time (that day)
- **We strongly recommend typing your assignments on a computer, not hand-writing**
- Late submissions possible though end of class (red box)
- Late submissions for 24 hours after class: bring to my office (4-225E Keller Hall), slip under door if closed
- Late submission are worth 85% credit

Course registration update

- **OK with space in lecture hall (as you can see)**
- **Limiting factor is discussion sections and TAs**
  - All sections full or close to full
  - But can probably squeeze in a handful more people
- **If you are not properly registered but want to be, email me with:**
  - Which discussion section(s) work with your schedule
  - And why the others wouldn’t
  - **As space allows, we’ll provide permissions numbers**
  - Use by Monday
Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
- Summary
- Representations in memory, pointers, strings
- Summary

**Binary Number Property**

Claim

\[ 1 + 1 + 2 + 4 + 8 + \ldots + 2^{w-1} = 2^w \]

\[ 1 + \sum_{i=0}^{w-1} 2^i = 2^w \]

- \( w = 0 \):
  - \( 1 = 2^0 \)
- Assume true for \( w-1 \):
  - \[ 1 + 1 + 2 + 4 + 8 + \ldots + 2^{w-1} + 2^w = 2^w + 2^w = 2^{w+1} \]

\[ 0 \]

**Encoding Integers**

- \texttt{short}
- \texttt{int}
- \[ x = 15213; \]
- \[ y = -15213; \]
- \texttt{C short 2 bytes long}
- Sign Bit
  - For 2's complement, most significant bit indicates sign
  - 0 for nonnegative
  - 1 for negative

**Two-complement Encoding Example (Cont.)**

<table>
<thead>
<tr>
<th>Weight</th>
<th>15213</th>
<th>-15213</th>
</tr>
</thead>
<tbody>
<tr>
<td>1536</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>768</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>384</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>192</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>96</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>48</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>24</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Four-bit Example**

- Unsigned: \[ 1111 = 15_{10} \]
- Two's complement: \[ 1111 = -1_{10} \]

**Numeric Ranges**

- **Unsigned Values**
  - \( U_{\text{Min}} = 0 \)
  - \( U_{\text{Max}} = 2^w - 1 \)
- **Two's Complement Values**
  - \( T_{\text{Min}} = -2^{w-1} \)
  - \( T_{\text{Max}} = 2^w - 1 \)
- **Other Values**
  - Minus 1

Values for \( W = 16 \)

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>65535</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>32767</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>-32768</td>
<td>80 FF</td>
<td>10000000 00000000</td>
</tr>
<tr>
<td>-1</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<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
  - $|\text{TMin}| = \text{TMax} + 1$
  - Asymmetric range
  - $\text{UMax} = 2 \times \text{TMax} + 1$

- 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>1010</td>
<td>2</td>
<td>-2</td>
</tr>
<tr>
<td>0111</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0011</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>1010</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>9</td>
</tr>
<tr>
<td>1010</td>
<td>10</td>
<td>-10</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>

### Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
- Representations in memory, pointers, strings

### Announcement: Data Lab posted

- First lab assignment went up earlier today
- “Data Lab” covers bitwise, integer, and floating-point operations
- Goal: expression operations using a small number of other operations:
  - E.g., check whether a value is negative without using `<`
- Due date: Monday, February 8th
- You can get started on the first questions now
- Please use the forum for non-spoiler questions
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: $X \rightarrow \overline{X}$

Unsigned: $X$

Maintain Same Bit Pattern

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`, `4294967295U`

- Casting
  - Explicit casting between signed & unsigned same as U2T and T2U
    - `int tx, ty;
      unsigned ux, uy;
      tx = (int) ux;
      uy = (unsigned) ty;`
  - Implicit casting also occurs via assignments and procedure calls
    - `tx = ux;
      uy = ty;`

Casting and Comparison 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: $T\text{MIN} = -2,147,483,648$, $T\text{MAX} = 2,147,483,647$

- Constant
  - `0` $=$ `0U`
  - `-1` $=$ `0U`
  - `2147483647` $=$ `2147483647U`
Summary

Casting Signed ↔ Unsigned: Basic Rules

- Bit pattern is maintained
- But reinterpreted
- Can have unexpected effects: adding or subtracting $2^n$
- Expression containing signed and unsigned int
  - int is cast to unsigned!!

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 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);
}
```

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:
    
    $$X' = x_{w-1} \cdots x_{w-k} x_{w-1} \cdots x_0$$

- Representations in memory, pointers, strings
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>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>y</td>
<td>C4 93</td>
<td>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: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
- Representations in memory, pointers, strings
- Summary

Unsigned Addition

Operands: w bits

True Sum: w+1 bits

Discard Carry: w bits

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

- Standard Addition Function
  - Ignores carry output

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

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

Visualizing Unsigned Addition

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

True Sum

```
\[
\text{Overflow} = 2^{w+1}
\]
```

```
\[
\text{Modular Sum} = u + v \mod 2^w
\]
```
Mathematical Properties

- Modular Addition Forms an Abelian Group
  - Closed under addition
    \[ 0 \leq UAdd(u, v) \leq 2^{n-1} \]
  - Commutative
    \[ UAdd(u, v) = UAdd(v, u) \]
  - Associative
    \[ UAdd(u, UAdd(v, w)) = UAdd(UAdd(u, v), w) \]
  - 0 is additive identity
    \[ UAdd(u, 0) = u \]
  - Every element has additive inverse
    \[ UAdd(u, UComp(u)) = 0 \]

- Closed, Commutative, Associative, 0 is additive identity

Two's Complement Addition

- Operands: \( w \) bits
  - \( u \)
  - \( v \)

- True Sum: \( w+1 \) bits
  - \( u \oplus v \)

- Discard Carry: \( w \) bits
  - \( TAdd(u, v) \)

- Signed vs. unsigned addition in C:
  - \( u + v \) vs. \((\text{unsigned}) u + (\text{unsigned}) v\)
  - \( s = (\text{unsigned}) u + (\text{unsigned}) v\)

- \( t = u + v \)

- \( a \equiv t \)

TAdd Overflow

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

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

Mathematical Properties of TAdd

- Isomorphic Group to unisgneds with UAdd
  - \( TAdd(u, v) = U2T(UAdd(T2U(u), T2U(v))) \)
  - 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
    \[ TComp(u) = \begin{cases} u + TMin_w, & u > TMin_w \\ u, & u < TMin_w \end{cases} \]

Characterizing TAdd

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

- **Unsigned:**
  - Overflow if carry out of last position
  - Also just called “carry” (C)
- **Signed:**
  - Result wrong if input signs are the same but output sign is different
  - In CPUs, unqualified “overflow” usually means signed (O or V)

Two's complement min (negative):

- Up to \(2^{n-1} - 1\) for \(n\)-bit numbers

Two's complement max (positive):

- Up to \(2^{n-1}\) for \(n\)-bit numbers

Example:

- \(x = 0\):

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>000000000</td>
</tr>
<tr>
<td>1</td>
<td>000</td>
<td>000000000</td>
</tr>
</tbody>
</table>

- \(x = 15213\):

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>15213</td>
<td>C493</td>
<td>1111111111</td>
</tr>
<tr>
<td>-15213</td>
<td>C493</td>
<td>1111111111</td>
</tr>
</tbody>
</table>

Sign bit table, signed ordering

<table>
<thead>
<tr>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>-3</td>
<td>-2</td>
<td>-1</td>
</tr>
<tr>
<td>-2</td>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>-3</td>
<td>-2</td>
</tr>
<tr>
<td>-3</td>
<td>-2</td>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>-3</td>
</tr>
<tr>
<td>-4</td>
<td>-3</td>
<td>-2</td>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>-5</td>
<td>-4</td>
<td>-3</td>
<td>-2</td>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Negation: Complement & Increment

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

- **Complement**

  - Observation: \(-x + x = 1111...111 = -1\)

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

Complement & Increment Examples

- \(x = 15213\):

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>15213</td>
<td>C493</td>
<td>1111111111</td>
</tr>
<tr>
<td>-15213</td>
<td>C493</td>
<td>1111111111</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>0000000000</td>
</tr>
<tr>
<td>1</td>
<td>000</td>
<td>0000000000</td>
</tr>
</tbody>
</table>

Multiplication

- **Goal:** Computing Product of \(w\)-bit numbers \(x, y\)
  - Either signed or unsigned
  - But, exact results can be bigger than \(w\) bits
    - Unsigned: up to \(2w\) bits
      - Result range: \(0 \leq x \cdot y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1\)
    - Two’s complement min (negative): Up to \(2w-1\) bits
      - Result range: \(x \cdot y \leq (2^{w-1})(2^{w-1}) = 2^{2w-2} + 2^{w-1}\)
    - Two’s complement max (positive): Up to \(2w\) bits, but only for \(|TMin|\)
      - Result range: \(x \cdot y \leq (2^{w-2}) \cdot 2 = 2^{2w-2}\)
  - So, maintaining exact results...
    - would need to keep expanding word size with each product computed
    - is done in software, if needed
      - e.g., 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
  \[ \text{UMult}_w(u, v) = u \cdot v \mod 2^w \]

Signed Multiplication in C

Operands: w bits

True Product: 2\*w 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

Code Security Example #2

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

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

XDR Code

```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
        // Move pointer to next memory region
        next += ele_size;
    }
    return result;
}
```

XDR Vulnerability

malloc(ele_cnt * ele_size)

- What if:
  - ele_cnt = 2^{20} + 1
  - ele_size = 4096 = 2^{12}
  - Allocation is ??
- Chime in: https://chimein.cla.umn.edu/course/view/2021 (Question 18527)
- (2^{20} + 1) \cdot 2^{12} = 2^{20} \cdot 2^{12} + 2^{12} + 2^{12} \approx 2^{12}

- How can I make this function secure?

Power-of-2 Multiply with Shift

- Operation
  - \( u \ll k \) gives \( u \cdot 2^k \)
  - Both signed and unsigned

- Examples
  - \( u \ll 3 \) == \( u \cdot 8 \)
  - \( (u \ll 5) - (u \ll 3) \) == \( u \cdot 24 \)
  - Most machines shift and add faster than multiply
### Compiled Multiplication Code

**C Function**

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

### Compiled Arithmetic Operations

- **lasmq (trax, trax, 2), trax**
- **salq $2, trax**

- **t <= mwsp2;**
- **return t << 2;**

**Explanation**

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

---

### Background: Rounding in Math

- **How to round to the nearest integer?**

- **Cannot have both:**
  - $\text{round}(x + k) = \text{round}(x) + k$ (k integer), “translation invariance”
  - $\text{round}(-x) = -\text{round}(x)$ “negation invariance”

- **\( \lfloor x \rfloor \), read “floor”: always round down (to \(-\infty\)):**
  - $\lfloor 2.0 \rfloor = 2, \lfloor 1.7 \rfloor = 1, \lfloor -2.2 \rfloor = -3$

- **\( \lceil x \rceil \), read “ceiling”: always round up (to \(+\infty\)):**
  - $\lceil 2.0 \rceil = 2, \lceil 1.7 \rceil = 2, \lceil -2.2 \rceil = -2$

- C integer operators mostly use round to zero, which is like floor for positive and ceiling for negative

---

### Divison in C

- **Integer division `/`: rounds towards 0**
  - Choice (settled in C99) is historical, via FORTRAN and most CPUs

- **Division by zero: undefined, usually fatal**

- **Unsigned division: no overflow possible**

- **Signed division: overflow almost impossible**
  - Exception: TMin/-1 is un-representable, and so undefined
  - On x86 this too is a default-fatal exception

---

### 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:**
  - $u$
  - $2^k$

- **Binary Point**
  - $u$
  - $2^k$

- **Division:**
  - $u / 2^k$

- **Result:**
  - $\lfloor u / 2^k \rfloor$

---

### Divison Computed Hex Binary

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u / 2^1$</td>
<td>$u &gt;&gt; 1$</td>
<td>7606</td>
<td>0111 1010 0110 1101</td>
</tr>
<tr>
<td>$u / 2^4$</td>
<td>$u &gt;&gt; 4$</td>
<td>900</td>
<td>00001101 01010110</td>
</tr>
<tr>
<td>$u / 2^8$</td>
<td>$u &gt;&gt; 8$</td>
<td>59</td>
<td>00000000 00111001</td>
</tr>
</tbody>
</table>

### Compiled Unsigned Division Code

**C Function**

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

**Explanation**

- Uses logical shift for unsigned

- For Java Users
  - Logical shift written as >>>

---

### Undefined behavior

- **Many things you should not do are officially called “undefined” by the C language standard**
  - Meaning: compiler can do anything it wants

- **Examples:**
  - Accessing beyond the ends of an array
  - Dividing by zero

- Things you do in this section of the course!
  - Overflow in signed operations
  - Shifts of negative values
  - Gap between standard and lenient practical compilers not yet resolved

---

### Compiled Unsigned Division Code

**C Function**

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

**Explanation**

- 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>( \lfloor x / 2^k \rfloor )</td>
</tr>
</tbody>
</table>

\[ x \gg k = \lfloor x / 2^k \rfloor \]

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-16213)</td>
<td>-16213</td>
<td>C493</td>
<td>(11000100 , 10010011)</td>
</tr>
<tr>
<td>(-7606.5)</td>
<td>-7607</td>
<td>B249</td>
<td>(11100001 , 10010011)</td>
</tr>
<tr>
<td>(-950.8125)</td>
<td>-951</td>
<td>FC49</td>
<td>(11111100 , 10010011)</td>
</tr>
<tr>
<td>(-59.4257813)</td>
<td>-60</td>
<td>FF4C</td>
<td>(11111111 , 11000100)</td>
</tr>
</tbody>
</table>

Correct Power-of-2 Divide

- Quotient of Negative Number by Power of 2
  - Want \( \lceil x / 2^k \rceil \) (Round Toward 0)
  - Compute as \( \lceil (x+2^k-1) / 2^k \rceil \)
  - In C: \( x + (1<<k) - 1 \gg k \)
  - Biases dividend toward 0

<table>
<thead>
<tr>
<th>Dividend:</th>
<th>( \mu )</th>
<th>( -2^k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lfloor x / 2^k \rfloor )</td>
<td>( \lfloor \mu / 2^k \rfloor )</td>
<td></td>
</tr>
</tbody>
</table>

Biasing has no effect

Correct Power-of-2 Divide (Cont.)

Case 2: Rounding

<table>
<thead>
<tr>
<th>Dividend:</th>
<th>( x \gg k )</th>
<th>( x / 2^k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incremented by 1</td>
<td></td>
<td>Incremented by 1</td>
</tr>
</tbody>
</table>

Biasing adds 1 to final result

Compiled Signed Division Code

C Function

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

Compiled Arithmetic Operations

```c
tastq %rax, %rax
    jm L4
L3:
    sarq $3, %rax
    ret
L4:
    addq $7, %rax
    jmp L3
```

Explanation

- Uses arithmetic shift for int
- For Java Users
  - Arith. shift written as \( \gg \)

Remainder operator

- Written as \( \% \) in C
- \( x \% y \) is the remainder after division \( x / y \)
  - E.g., \( x \% 10 \) is the lowest digit of non-negative \( x \)
  - Behavior for negative values matches \( / \)'s rounding toward zero
    - \( b^a \cdot (a / b) + (a \% b) = a \)
  - I.e. sign of remainder matches sign of dividend
  - (Some other languages have other conventions: sign of result equals sign of divisor, sometimes distinguished as “modulo”, or always positive)

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
  - Representations in memory, pointers, strings
Arithmetic: Basic Rules

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

- **Multiplication:**
  - Unsigned/signed: Normal multiplication followed by truncate, same operation on bit level
  - Unsigned: multiplication mod 2
  - Positive numbers: div (division + round to zero) by 2
  - Isomorphic to ring of integers mod 2

Properties of Unsigned Arithmetic

- **Unsigned Multiplication with Addition Forms**
  - Commutative Ring
  - Addition is commutative group
  - Closed under multiplication
  - 0 ≤ MUlt(u, v) ≤ 2 - 1
  - Multiplication Commutative
    - MUlt(u, v) = MUlt(v, u)
  - Multiplication is Associative
    - MUlt(t, MUlt(u, v)) = MUlt(t, MUlt(u, v))
  - 1 is multiplicative identity
    - MUlt(u, 1) = u
  - Multiplication distributes over addition
    - MUlt(u, UAdd(u, v)) = UAdd(MUlt(t, u), MUlt(t, v))

Why Should I Use Unsigned?

- **Don’t use without understanding implications**
  - Easy to make mistakes
    - unsigned i;
    - for (i = cnt-2; i >= 0; i--)
      - a[i] = a[i+1];
  - Can be very subtle
    - #define DELTA sizeof(int)
      - int i;
      - for (i = cnt; i-DELTA >= 0; i = DELTA) - - -

Arithmetic: Basic Rules

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

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

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

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

- **Comparison to (Mathematical) Integer Arithmetic**
  - Both are rings
  - Integers obey ordering properties, e.g.,
    - u > 0
    - u + v > 0
    - These properties are not obeyed by two’s comp. arithmetic

  - TMax + 1 == TMin

  - 15213 * 30426 == -10030

  - (16-bit words)

Counting Down with Unsigned

- **Proper way to use unsigned as loop index**
  - unsigned i;
  - for (i = cnt-2; i < cnt; i--)
    - a[i] = a[i+1];

- **See Robert Seacord, Secure Coding in C and C++**
  - C Standard guarantees that unsigned addition will behave like modular arithmetic
    - 0 – 1 == UMax

- **Even better**
  - size_t i;
  - for (i = cnt-2; i < cnt; i--)
    - a[i] = a[i+1];
  - Data type size_t defined as unsigned value with length = word size
  - Code will work even if cnt = UMax
  - What if cnt is signed and < 0?
Why Should I Use Unsigned? (cont.)

- Do Use When Performing Modular Arithmetic
  - Multiplication arithmetic
- Do Use When Using Bits to Represent Sets
  - Logical right shift, no sign extension

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
- Summary
  - Representations in memory, pointers, strings

Byte-Oriented Memory Organization

- Programs refer to data by address
  - Conceptually, envision it as a very large array of bytes
  - In reality, it’s not, but can think of it that way
  - An address is like an index into that array
  - and, a pointer variable stores an address

Note: system provides private address spaces to each “process”

- Think of a process as a program being executed
- So, a program can clobber its own data, but not that of others

Machine Words

- Any given computer has a “Word Size”
  - Nominal size of integer-valued data
    - and of addresses
  - Until recently, most machines used 32 bits (4 bytes) as word size
    - Limits addresses to 4GB ($2^{32}$ bytes)
  - Increasingly, machines have 64-bit word size
    - Potentially, could have 18 EB (exabytes) of addressable memory
      - That’s $18.4 \times 10^{18}$
    - Machines still support multiple data formats
      - Fractions or multiples of word size
      - Always integral number of bytes

Word-Oriented Memory Organization

- Addresses Specify Byte Locations
  - Address of first byte in word
  - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Typical 32-bit</th>
<th>Typical 64-bit</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>4</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>–</td>
<td>–</td>
<td>10/16</td>
</tr>
<tr>
<td>pointer</td>
<td>4</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

Example Data Representations
Byte Ordering

- So, how are the bytes within a multi-byte word ordered in memory?
- Conventions
  - Big Endian: Sun, PPC Mac, Internet
  - Least significant byte has highest address
  - Little Endian: x86, ARM processors running Android, iOS, and Windows
    - Least significant byte has lowest address

Example

- Variable \( x \) has 4-byte value of 0x01234567
- Address given by \&\( x \) is 0x100

<table>
<thead>
<tr>
<th>Big Endian</th>
<th>Little Endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>01 23 45 67</td>
<td>67 45 23 01</td>
</tr>
</tbody>
</table>

Representing Integers

Int \( A = 15213; \)

<table>
<thead>
<tr>
<th>IA32, x86-64</th>
<th>Sun</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00 00 0D</td>
<td>FF</td>
</tr>
</tbody>
</table>

Int \( B = -15213; \)

<table>
<thead>
<tr>
<th>IA32, x86-64</th>
<th>Sun</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00 00 6D</td>
<td>FF</td>
</tr>
</tbody>
</table>

Examining Data Representations

- Code to Print Byte Representation of Data
  - Casting pointer to unsigned char * allows treatment as a byte array
  - Typedef unsigned char *pointer;

```c
typedef unsigned char *pointer;

void show_bytes(pointer start, size_t len){
    size_t i;
    for (i = 0; i < len; i++)
        printf(“%p %x\n”, start+i, start[i]);
}
```

Print directives:
- %p: Print pointer
- %x: Print hexadecimal

Representing Pointers

Int \( B = -15213; \)

<table>
<thead>
<tr>
<th>IA32</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>EF FF 28</td>
<td>3C 1B</td>
</tr>
<tr>
<td>FB FF</td>
<td>FE 82</td>
</tr>
</tbody>
</table>

Different compilers & machines assign different locations to objects

Even get different results each time run program
Representing Strings

- **Strings in C**
  - Represented by array of characters
  - Each character encoded in ASCII format
    - Standard 7-bit encoding of character set
    - Character "0" has code 0x30
    - Digit i has code 0x30+i
  - String should be null-terminated
    - Final character = 0

- **Compatibility**
  - Byte ordering not an issue

```
char S[6] = "182135";
```

Reading Byte-Reversed Listings

- **Disassembly**
  - Text representation of binary machine code
  - Generated by program that reads the machine code

- **Example Fragment**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Code</th>
<th>Assembly Rendition</th>
</tr>
</thead>
<tbody>
<tr>
<td>8048365: 5b</td>
<td>pop %ebx</td>
<td></td>
</tr>
<tr>
<td>8048366: 81 c3 ab 12 00 00</td>
<td>add $0x12ab,%ebx</td>
<td></td>
</tr>
<tr>
<td>804836c: 83 bb 28 00 00 00 00</td>
<td>cmpl $0x0,0x28(%ebx)</td>
<td></td>
</tr>
</tbody>
</table>

- **Deciphering Numbers**
  - Value: 0x12ab
  - Pad to 32 bits: 00000012ab
  - Split into bytes: 00 00 12 ab
  - Reverse: ab 12 00 00

Integer C Puzzles

1. \(x < 0 \implies ((x*2) < 0)\)
2. \(ux > -1\)
3. \(x > 0 \&\& y > 0 \implies x + y > 0\)
4. \((x|-x)>>31 == -1\)

Initialization

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

Bonus: More Integer C Puzzles

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

Initialization

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