Bits, Bytes, and Integers

15-213: Introduction to Computer Systems
2nd and 3rd Lectures, Sep 1 and Sep 6, 2011

Instructors:
Dave O’Hallaron, Greg Ganger, and Greg Kesden
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
Binary Representations

- **Base 2 Number Representation**
  - Represent $15213_{10}$ as $11101101101101_2$
  - Represent $1.20_{10}$ as $1.0011001100110011[0011]..._2$
  - Represent $1.5213 \times 10^4$ as $1.1101101101101_2 \times 2^{13}$

- **Electronic Implementation**
  - Easy to store with bistable elements
  - Reliably transmitted on noisy and inaccurate wires

![Graph showing voltage levels and binary representation]
Encoding Byte Values

- **Byte = 8 bits**
  - Binary 00000000<sub>2</sub> to 11111111<sub>2</sub>
  - Decimal: 0<sub>10</sub> to 255<sub>10</sub>
  - Hexadecimal 00<sub>16</sub> to FF<sub>16</sub>
    - Base 16 number representation
    - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
    - Write FA1D37B<sub>16</sub> in C as
      - 0xFA1D37B
      - 0xfa1d37b

<table>
<thead>
<tr>
<th>Hex</th>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>0111</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>A</td>
<td>10</td>
<td>1010</td>
</tr>
<tr>
<td>B</td>
<td>11</td>
<td>1011</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>1100</td>
</tr>
<tr>
<td>D</td>
<td>13</td>
<td>1101</td>
</tr>
<tr>
<td>E</td>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>F</td>
<td>15</td>
<td>1111</td>
</tr>
</tbody>
</table>
## Data Representations

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Typical 32-bit</th>
<th>Intel IA32</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>4</td>
<td>8</td>
</tr>
<tr>
<td>long long</td>
<td>8</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>8</td>
<td>10/12</td>
<td>10/16</td>
</tr>
<tr>
<td>pointer</td>
<td>4</td>
<td>4</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
- \( A \& B = 1 \) when both \( A=1 \) and \( B=1 \)

<table>
<thead>
<tr>
<th>&amp;</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Or
- \( A | B = 1 \) when either \( A=1 \) or \( B=1 \)

<table>
<thead>
<tr>
<th>( )</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Not
- \(~A = 1 \) when \( A=0 \)

<table>
<thead>
<tr>
<th>( )</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Exclusive-Or (Xor)
- \( A ^ B = 1 \) when either \( A=1 \) or \( B=1 \), but not both

<table>
<thead>
<tr>
<th>^</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
General Boolean Algebras

- Operate on Bit Vectors
  - Operations applied bitwise

\[
\begin{array}{cccc}
01101001 & 01101001 & 01101001 \\
\& 01010101 & | 01010101 & ^ 01010101 & \sim 01010101 \\
01000001 & 01111101 & 00111100 & 10101010
\end{array}
\]

- 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 $j \in A$
    - $01101001$  $\{0, 3, 5, 6\}$
  - $76543210$
    - $01010101$  $\{0, 2, 4, 6\}$
    - $76543210$

■ Operations
  ▪ &  Intersection  $01000001$  $\{0, 6\}$
  ▪ |  Union  $01111101$  $\{0, 2, 3, 4, 5, 6\}$
  ▪ ^  Symmetric difference  $00111100$  $\{2, 3, 4, 5\}$
  ▪ ~  Complement  $10101010$  $\{1, 3, 5, 7\}$
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)
  - ~0x41 → 0xBE
    - ~01000001₂ → 10111110₂
  - ~0x00 → 0xFF
    - ~00000000₂ → 11111111₂
  - 0x69 & 0x55 → 0x41
    - 01101001₂ & 01010101₂ → 01000001₂
  - 0x69 | 0x55 → 0x7D
    - 01101001₂ | 01010101₂ → 01111101₂
Contrast: Logic Operations in C

- **Contrast to Logical Operators**
  - `&`, `|`, `!`
    - View 0 as “False”
    - Anything nonzero as “True”
    - Always return 0 or 1
    - Early termination

- **Examples (char data type)**
  - `!0x41` ➔ `0x00`
  - `!0x00` ➔ `0x01`
  - `!!0x41` ➔ `0x01`
  - `0x69 && 0x55` ➔ `0x01`
  - `0x69 || 0x55` ➔ `0x01`
  - `p && *p` (avoids null pointer access)
Contrast: Logic Operations in C

- Contrast to Logical Operators
  - &&, ||, !
  - View 0 as "False"
  - Anything nonzero as "True"
  - Always return 0 or 1
  - Early termination

- Examples
  - !0x41 ➞ 0x00
  - !0x00 ➞ 0x01
  - !!0x41 ➞ 0x01
  - 0x69 && 0x55 ➞ 0x01
  - 0x69 || 0x55 ➞ 0x01
  - p && *p  (avoids null pointer access)

Watch out for && vs. & (and || vs. |)… one of the more common oopsies in C programming
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 \( \geq \) word size
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
Encoding Integers

Unsigned

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

Two’s Complement

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

C short 2 bytes long

<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>y</td>
<td>-15213</td>
<td>C4 93 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.)

\[ x = \begin{array}{c} 15213: 00111011 01101101 \\ y = \begin{array}{c} -15213: 11000100 10010011 \end{array} \end{array} \]

<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>

Sum: \[ 15213 \quad -15213 \]
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></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>65535</td>
<td>FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>Tmax</td>
<td>32767</td>
<td>7F</td>
<td>01111111 11111111</td>
</tr>
<tr>
<td>Tmin</td>
<td>-32768</td>
<td>80</td>
<td>10000000 00000000</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>
Values for Different Word Sizes

<table>
<thead>
<tr>
<th></th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>UMax</td>
<td>255</td>
</tr>
<tr>
<td></td>
<td>65,535</td>
</tr>
<tr>
<td></td>
<td>4,294,967,295</td>
</tr>
<tr>
<td></td>
<td>18,446,744,073,709,551,615</td>
</tr>
<tr>
<td>Tmax</td>
<td>127</td>
</tr>
<tr>
<td></td>
<td>32,767</td>
</tr>
<tr>
<td></td>
<td>2,147,483,647</td>
</tr>
<tr>
<td></td>
<td>9,223,372,036,854,775,807</td>
</tr>
<tr>
<td>Tmin</td>
<td>-128</td>
</tr>
<tr>
<td></td>
<td>-32,768</td>
</tr>
<tr>
<td></td>
<td>-2,147,483,648</td>
</tr>
<tr>
<td></td>
<td>-9,223,372,036,854,775,808</td>
</tr>
</tbody>
</table>

**Observations**

- \(|T_{\text{Min}}| = T_{\text{Max}} + 1\)
  - Asymmetric range
- \(U_{\text{Max}} = 2 * T_{\text{Max}} + 1\)

**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

- **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: 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
Mapping Between Signed & Unsigned

Two’s Complement

Unsigned

Maintain Same Bit Pattern

Mappings between 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>
# 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 → Signed

Large negative weight becomes Large positive weight

unsigned

Maintain Same Bit Pattern

Two’s Complement

x

unsigned

+ + + ... + + +

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&lt;sub&gt;1&lt;/sub&gt;</th>
<th>Constant&lt;sub&gt;2&lt;/sub&gt;</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>
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: 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
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:
  ▪ $X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0$

```
X
...

X'

k copies of MSB
```

```

w

w

k
```
Sign Extension Example

```
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
  - For small numbers yields expected behaviour
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

**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

$2^{w+1}$

$2^w$

$0$

Modular Sum

Overflow

$U\text{Add}_4(u, v)$
Two’s Complement Addition

Operands: \( w \) bits

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

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**
  - $011\ldots1$ $2^{w-1}$
  - $010\ldots0$ $2^{w-1}$
  - $000\ldots0$ $0$
  - $101\ldots1$ $-2^{w-1}-1$
  - $100\ldots0$ $-2^w$

- **TAdd Result**
  - $011\ldots1$
  - $000\ldots0$
  - $100\ldots0$
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
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: $0 \leq x \times y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
  - Two’s complement min (negative): Up to $2w-1$ bits
    - Result range: $x \times y \geq (-2^{w-1}) \times (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$
  - Two’s complement max (positive): Up to $2w$ bits, but only for $(TMin_w)^2$
    - Result range: $x \times y \leq (-2^{w-1})^2 = 2^{2w-2}$

- **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**
  
  $UMult_w(u, v) = u \cdot v \mod 2^w$
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
Power-of-2 Multiply with Shift

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

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

- **Examples**
  - \( u \ll 3 \) \( \equiv \) \( u \times 8 \)
  - \( u \ll 5 - u \ll 3 \) \( \equiv \) \( u \times 24 \)
  - Most machines shift and add faster than multiply
    - Compiler generates this code automatically
UnSigned Power-of-2 Divide with Shift

- **Quotient of Unsigned by Power of 2**
  - \( u \gg k \) gives \( \left\lfloor \frac{u}{2^k} \right\rfloor \)
  - Uses logical shift

### Diagram

![Division Diagram](image)

- **Operands:**
  - \( u \)
  - \( \frac{1}{2^k} \)

- **Division:**
  - \( u \div 2^k \)

- **Result:**
  - \( \left\lfloor \frac{u}{2^k} \right\rfloor \)

### 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>15213</td>
<td>3B 6D 00111011 01101101</td>
</tr>
<tr>
<td>x \gg 1</td>
<td>7606.5</td>
<td>7606</td>
<td>1D B6 00011101 10110110</td>
</tr>
<tr>
<td>x \gg 4</td>
<td>950.8125</td>
<td>950</td>
<td>03 B6 00000011 10110110</td>
</tr>
<tr>
<td>x \gg 8</td>
<td>59.4257813</td>
<td>59</td>
<td>00 3B 00000000 00111011</td>
</tr>
</tbody>
</table>
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 \( \alpha < 0 \)

### Table

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y )</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>( y \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:
\[
\begin{array}{c}
\hline
u \quad 1 \ldots 0 \ldots 0 \\
+2^k - 1 \quad 0 \ldots 0 1 \ldots 1 \\
\hline
\end{array}
\]

Divisor:
\[
\begin{array}{c}
\hline
\frac{l}{2^k} \quad 0 \ldots 0 1 0 \ldots 0 \\
\hline
\end{array}
\]

\[ \left\lfloor \frac{u}{2^k} \right\rfloor \]

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

Case 2: Rounding

Dividend: \( x \)
\[
\begin{array}{c}
1 \cdots 1 \cdots 1 \\
+ 2^k - 1
\end{array}
\]
\[
\begin{array}{c}
0 \cdots 011 \cdots 11
\end{array}
\]

Divisor: \( l \)
\[
\begin{array}{c}
1 \cdots 1 \cdots 11 \cdots 11 \\
\downarrow \quad 2^k
\end{array}
\]
\[
\begin{array}{c}
0 \cdots 010 \cdots 00
\end{array}
\]

\[ x / 2^k \]

Incremented by 1

Binary Point

Incremented by 1

Biasing adds 1 to final result
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 $2^w$
    - 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)
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
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
  - Most current machines use 32 bits (4 bytes) as word size
    - Limits addresses to 4GB ($2^{32}$ bytes)
    - Becoming too small for memory-intensive applications
      - leading to emergence of computers with 64-bit word size
  - 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>32-bit Words</th>
<th>64-bit Words</th>
<th>Bytes</th>
<th>Addr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr = 0000</td>
<td>Addr = 0000</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>Addr = 0004</td>
<td></td>
<td>0001</td>
<td></td>
</tr>
<tr>
<td>Addr = 0008</td>
<td></td>
<td>0002</td>
<td></td>
</tr>
<tr>
<td>Addr = 0012</td>
<td></td>
<td>0003</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Addr = 0008</td>
<td>0004</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0005</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0006</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0007</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0008</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0009</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0010</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0011</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0012</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0013</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0014</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0015</td>
<td></td>
</tr>
</tbody>
</table>
For other data representations too ...

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Typical 32-bit</th>
<th>Intel IA32</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>4</td>
<td>8</td>
</tr>
<tr>
<td>long long</td>
<td>8</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>8</td>
<td>10/12</td>
<td>10/16</td>
</tr>
<tr>
<td>pointer</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>
Byte Ordering

- So, how are the bytes within a multi-byte word ordered in memory?

- Conventions
  - BigEndian: Sun, PPC Mac, Internet
    - Least significant byte has highest address
  - LittleEndian: x86
    - Least significant byte has lowest address
Byte Ordering Example

- **Example**
  - Variable x has 4-byte value of 0x01234567
  - Address given by &x is 0x100

**Big Endian**

```
0x100 0x101 0x102 0x103
  | 01 23 45 67 |
```

**Little Endian**

```
0x100 0x101 0x102 0x103
  | 67 45 23 01 |
```
Representing Integers

\[ \text{int } A = 15213; \]

\[ \text{long int } C = 15213; \]

\[ \text{int } B = -15213; \]

Two’s complement representation
Examining Data Representations

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

```c
typedef unsigned char *pointer;

void show_bytes(pointer start, int len){
    int i;
    for (i = 0; i < len; i++)
        printf("%p\t0x%.2x\n", start+i, start[i]);
    printf("\n");
}
```

**Printf directives:**
- %p: Print pointer
- %x: Print Hexadecimal
show_bytes Execution Example

```c
int a = 15213;
printf("int a = 15213;\n");
show_bytes((pointer) &a, sizeof(int));
```

Result (Linux):

```c
int a = 15213;
0x11fffffcba 0x00
0x11ffffffcb8 0x6d
0x11ffffffcb9 0x3b
0x11ffffffcbb 0x00
```
Representing Pointers

```c
int B = -15213;
int *P = &B;
```

Different compilers & machines assign different locations to objects
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

```c
char S[6] = "18243";
```
### Integer C Puzzles

#### Initialization

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

- `x < 0` \[\Rightarrow (x \times 2 < 0)\]
- `ux >= 0` \[\Rightarrow (x << 30 < 0)\]
- `x & 7 == 7` \[\Rightarrow (x << 30 < 0)\]
- `ux > -1` \[\Rightarrow -x < -y\]
- `x > y` \[\Rightarrow -x < -y\]
- `x * x >= 0` \[\Rightarrow x + y > 0\]
- `x > 0 && y > 0` \[\Rightarrow -x <= 0\]
- `x <= 0` \[\Rightarrow -x >= 0\]
- `(x|x-x)>>31 == -1` \[\Rightarrow -x >= 0\]
- `ux >> 3 == ux/8` \[\Rightarrow x >> 3 == x/8\]
- `x >> 3 == x/8` \[\Rightarrow x & (x-1) != 0\]
Bonus extras
Application of Boolean Algebra

- Applied to Digital Systems by Claude Shannon
  - 1937 MIT Master’s Thesis
  - Reason about networks of relay switches
    - Encode closed switch as 1, open switch as 0

\[ A \land \neg B \lor \neg A \land B \]

Connection when

\[ A \land \neg B \lor \neg A \land B = A \lor B \]

Diagram:

- Connections for \( A \land \neg B \)
- Connections for \( \neg A \land B \)
- Connections for \( \neg A \land B \)
- Connections for \( A \lor B \)
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);
    . . .
}

/* Declaration of library function memcpy */
void *memcpy(void *dest, void *src, size_t n);
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 \[ \text{UComp}_w(u) = 2^w - u \]
    \[ \text{UAdd}_w(u, \text{UComp}_w(u)) = 0 \]
Characterizing TAdd

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

\[
\begin{array}{c|c|c|c|c}
\text{TAdd}(u,v) & >0 & <0 & v & <0 \\
\hline
>0 & & & & \\
<0 & & & & \\
v & & & & \\
<0 & & & & \\
\end{array}
\]

Negative Overflow

Positive Overflow

\[
\begin{align*}
TAdd_w(u,v) & \quad u \quad v \quad 2^w \\
& \quad u \quad v \quad TMin_w \quad (\text{NegOver}) \\
& \quad u \quad v \quad TMax_w \quad (\text{PosOver})
\end{align*}
\]
Mathematical Properties of TAdd

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

\[
\begin{array}{ccc}
\text{TComp}_w(u) & u & u & \text{TMin}_w \\
\text{TMin}_w & u & \text{TMin}_w & \text{TMin}_w
\end{array}
\]
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);
```

```
int main()
{
    int ele_cnt = 5;
    size_t ele_size = 10;
    void *ele_src = malloc(ele_cnt * ele_size);
    void *dst = malloc(ele_cnt * ele_size);
    void *result = copy_elements(ele_src, ele_cnt, ele_size);
    return 0;
}
```
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 */
        memcpy(next, ele_src[i], ele_size);
        /* 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 = ??

- **How can I make this function secure?**
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
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 >>>>
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

- 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
Negation: Complement & Increment

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

- **Complement**
  - **Observation:** \( \neg x + x = 1111\ldots111 = -1 \)

\[
\begin{array}{c}
10011101 \\
+ 01100010 \\
\hline
11111111
\end{array}
\]

- **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>( \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 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 00</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>
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.,
    \[
    u > 0 \quad \Rightarrow \quad u + v > v \\
    u > 0, \ v > 0 \quad \Rightarrow \quad u \cdot v > 0
    \]
  - These properties are not obeyed by two’s comp. arithmetic
    \[
    TMax + 1 = TMin \\
    15213 \times 30426 = -10030 \quad \text{(16-bit words)}
    \]
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:</td>
<td>5b</td>
<td>pop %ebx</td>
</tr>
<tr>
<td>8048366:</td>
<td>81 c3 ab 12 00 00</td>
<td>add $0x12ab,%ebx</td>
</tr>
<tr>
<td>804836c:</td>
<td>83 bb 28 00 00 00</td>
<td>cmpl $0x0,0x28(%ebx)</td>
</tr>
</tbody>
</table>

- **Deciphering Numbers**
  - Value: 0x12ab
  - Pad to 32 bits: 0x000012ab
  - Split into bytes: 00 00 12 ab
  - Reverse: ab 12 00 00