Bits, Bytes, and Integers

15-213: Introduction to Computer Systems
2nd and 3rd Lectures, Jan 17 and Jan 22, 2013

Instructors:
Seth Copen Goldstein, Anthony Rowe, Greg Kesden

MLK recitations

- No recitations after 12:30, so ...
- The TAs have been kind enough to create some temporary sections:
  - GHC 4215: 10:30 & 11:30
  - GHC 4102: 11:30
  - GHC 4101: 9:30 & 10:30

Waitlist

- Please be patient.
- If you register for autolab, get the work done → you will be ready when you get into the class

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 \(1110110110110_{2}\)
  - Represent \(1.20_{10}\) as \(1.0011001100110011_{2}\)
  - Represent \(1.5213 \times 10^4\) as \(1.110110110110_{2} \times 2^{13}\)

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

```
3.3V - 0 - 1 - 0 - 2.8V
2.8V
0.5V
0.0V
```

Encoding Byte Values

- **Byte = 8 bits**
  - Binary: \(00000000_{2}\) to \(11111111_{2}\)
  - Decimal: \(0_{10}\) to \(255_{10}\)
  - Hexadecimal: \(0_{16}\) to \(FF_{16}\)
  - Use characters \('0' to '9' and 'A' to 'F'\)
  - Write \(FA1D37B_{16}\) 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>4</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 \( B=1 \)

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

  **Not**
  - \( \sim A = 1 \) when \( A=0 \)

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

### General Boolean Algebras

- Operate on Bit Vectors
  - Operations applied bitwise

<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>

<table>
<thead>
<tr>
<th>( \mid )</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>

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

<table>
<thead>
<tr>
<th>( \oplus )</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>

### 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
    - \( 01000001 \) \( \{0, 6\} \)
  - \mid Union
    - \( 01111101 \) \( \{0, 2, 3, 4, 5, 6\} \)
  - \( \oplus \) Symmetric difference
    - \( 00111100 \) \( \{2, 3, 4, 5\} \)
  - \( \sim \) 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)**
  - \( \sim 0x41 \rightarrow 0xBE \)
  - \( \sim 0x00 \rightarrow 0xFF \)
  - \( 0x69 \& 0x55 \rightarrow 0x41 \)
  - \( 0x69 \mid 0x55 \rightarrow 0x7D \)
  - \( 0x69 \oplus 0x55 \rightarrow 0x41 \)
  - \( 01101001 \& 01010101 \rightarrow 01000001 \)
  - \( 01101011 \mid 01010101 \rightarrow 01111101 \)
  - \( 01101011 \oplus 01010101 \rightarrow 01111100 \)
  - \( 01101011 \sim 01010101 \rightarrow 01111100 \)
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)

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 ≥ 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

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

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

- C short is 2 bytes long
- Short int x = 15213;
- Short int y = -15213;
- For 2's complement, most significant bit indicates sign
  - 0 for nonnegative
  - 1 for negative

### Numeric Ranges

#### Unsigned Values

- **UMin** = 0
  
  000...0

- **UMax** = \(2^w - 1\)
  
  111...1

#### Two’s Complement Values

- **TMin** = \(-2^{w-1}\)
  
  100...0

- **TMax** = \(2^{w-1} - 1\)
  
  011...1

#### Other Values

- Minus 1
  
  111...1

### Values for 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
  - **UMax** = \(2 \cdot TMax + 1\)

### C Programming

- C 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 a unique integer value
  - Each representable integer has a 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

<table>
<thead>
<tr>
<th>X</th>
<th>T2B</th>
<th>B2U</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>T2U</td>
<td>B2T</td>
</tr>
</tbody>
</table>

- Maintain Same Bit Pattern

Unsigned

<table>
<thead>
<tr>
<th>UX</th>
<th>U2T</th>
<th>B2T</th>
</tr>
</thead>
<tbody>
<tr>
<td>UX</td>
<td>U2B</td>
<td>B2U</td>
</tr>
</tbody>
</table>

- 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

\[ x \quad \text{T2U} \quad \text{U2T} \quad x \quad \text{B2U} \quad \text{Unsigned} \]

Maintain Same Bit Pattern

\[ \text{Large negative weight becomes Large positive weight} \]

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
    - `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 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 $W = 32$: $T_{\text{MIN}} = -2,147,483,648$, $T_{\text{MAX}} = 2,147,483,647$

<table>
<thead>
<tr>
<th>Constant 1</th>
<th>Constant 2</th>
<th>Relation</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0U</td>
<td><code>==</code></td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td><code>&lt;</code></td>
<td>signed</td>
</tr>
<tr>
<td>-1</td>
<td>0U</td>
<td><code>&gt;</code></td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>-2147483647-1</td>
<td><code>&gt;</code></td>
<td>signed</td>
</tr>
<tr>
<td>2147483647U</td>
<td>-2147483647-1</td>
<td><code>&lt;</code></td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>-2</td>
<td><code>&gt;</code></td>
<td>unsigned</td>
</tr>
<tr>
<td>(unsigned)-1</td>
<td>-2</td>
<td><code>&gt;</code></td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>2147483648U</td>
<td><code>&lt;</code></td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>(int) 2147483648U</td>
<td><code>&gt;</code></td>
<td>signed</td>
</tr>
</tbody>
</table>

Summary

Casting Signed $\leftrightarrow$ 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:
    - $X' = x_{w-1}, \ldots, x_{w-1}, x_{w-2}, \ldots, x_0$
  - Diagram:
    - $X$ to $X'$
Sign Extension Example

- Converting from smaller to larger integer data type
- C automatically performs sign extension

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x 15213</td>
<td>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>ix 15213</td>
<td>00 00 3B 6D</td>
<td>00000000 00000000 00111011 01101101</td>
</tr>
<tr>
<td>y -15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>iy -15213</td>
<td>FF FF C4 93</td>
<td>11111111 11111111 11000100 10010011</td>
</tr>
</tbody>
</table>

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

Fake real world example

- Acme, Inc. has developed a state of the art voltmeter they are connecting to a pc. It is precise to the millivolt and does not drain the unit under test.
- Your job is to develop the driver software.

```c
printf("%d\n", getValue());
```

Fake real world example

- Acme, Inc. has developed a state of the art voltmeter they are connecting to a pc. It is precise to the millivolt and does not drain the unit under test.
- Your job is to develop the driver software.

```c
printf("%d\n", getValue());
```
### Lets run some tests

```c
printf("%d\n", getValue());
```

- 50652
- 1500
- 9692
- 26076
- 17884
- 42460
- 34268
- 50652

### Only care about least significant 12 bits

```c
int x=getValue();
x=(x & 0x0fff); printf("%d\n",x);
```

```c
printf("%x\n", x);
```

---

### Lets run some tests

```c
int x=getValue();
printf("%d %08x\n",x, x);
```

```c
Those darn engineers!
```

- 50652 0000c5dc
- 1500 000005dc
- 9692 000025dc
- 26076 000065dc
- 17884 000045dc
- 42460 0000a5dc
- 34268 000085dc
- 50652 0000c5dc

---

### Only care about least significant 12 bits

```c
int x=getValue();
x=x(&0x0fff);
printf("%d\n",x);
```

```c
hmm?
```

```c
printf("%x\n", x);
```
Must sign extend

```c
int x=getValue();
x=(x&0x00fff) | (x&0x0800?0xfffff000:0);
printf("%d\n",x);
```

There is a better way.

Because you graduated from 213

```c
int x=getValue();
x=(x&0x00fff) | (x&0x0800?0xfffff000:0);
printf("%d\n",x);
```

huh?

Lets be really thorough

```c
int x=getValue();
x=(x&0x00fff) | (x&0x0800?0xfffff000:0);
printf("%d\n",x);
```

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**
  \[ 2^{w+1} \]

- **Modular Sum**
  \[ 2^w \]

**Two’s Complement Addition**

- **Operands**: $w$ bits
- **True Sum**: $w+1$ bits
- **Discard Carry**: $w$ bits

- **TAdd and UAdd have Identical Bit-Level Behavior**
  - Signed vs. unsigned addition in C:
    \[
    \begin{align*}
    \text{int } s, t, u, v; \\
    s &= \text{(int) } \text{(unsigned) } u + \text{(unsigned) } v; \\
    t &= u + v
    \end{align*}
    \]
  - Will give $s == t$
TAdd Overflow

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

<table>
<thead>
<tr>
<th>True Sum</th>
<th>TAdd Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 111...1</td>
<td>2^{w-1}</td>
</tr>
<tr>
<td>0 100...0</td>
<td>2^{w-1} - 1</td>
</tr>
<tr>
<td>0 000...0</td>
<td>0</td>
</tr>
<tr>
<td>1 011...1</td>
<td>-2^{w-1}</td>
</tr>
<tr>
<td>1 000...0</td>
<td>-2^w</td>
</tr>
</tbody>
</table>

**Visualizing 2’s Complement Addition**

- **Values**
  - 4-bit two’s comp.
  - Range from -8 to +7
- **Wraps Around**
  - If sum \( \geq 2^{w-1} \)
    - Becomes negative
    - At most once
  - If sum \( < -2^{w-1} \)
    - Becomes positive
    - At most once

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: \( x \cdot y \geq (-2^{w-1}) \cdot (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1} \)
  - Two’s complement max (positive): Up to \( 2w \) bits, but only for \( TMin_w \)\(^2 \)
    - Result range: \( x \cdot y \leq (-2^{w-1})^2 = 2^{2w-2} \)
- 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**

<table>
<thead>
<tr>
<th>Operands: ( w ) bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>( u )</td>
</tr>
</tbody>
</table>
| \( u \cdot v \mod 2^w \)

<table>
<thead>
<tr>
<th>True Product: ( 2w ) bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>( u \cdot v )</td>
</tr>
<tr>
<td>( \mod 2^w )</td>
</tr>
<tr>
<td>( \text{UMult}_w(u, v) )</td>
</tr>
</tbody>
</table>

- **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

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c}
\hline
& & & & & & & & & & \\
\hline
u & | & * & v & | & | & | & | & | & \\
\hline
\end{array}
\]

True Product: \( 2^w \) bits

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c}
\hline
& & & & & & & & & & \\
\hline
u \cdot v & | & | & | & | & | & | & | & | & \\
\hline
\end{array}
\]

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

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c}
\hline
& & & & & & & & & & \\
\hline
u & | & * & 2^k & | & | & | & | & | & \\
\hline
\end{array}
\]

Operands: \( w \) bits

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

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c}
\hline
& & & & & & & & & & \\
\hline
u & \cdot 2^k & | & | & | & | & | & | & | & \\
\hline
\end{array}
\]

**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 \( \lfloor u / 2^k \rfloor \)
- Uses logical shift

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c}
\hline
& & & & & & & & & & \\
\hline
u & | & k & | & | & | & | & | & | & \\
\hline
\end{array}
\]

Operands:

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c}
\hline
& & & & & & & & & & \\
\hline
u / 2^k & | & | & | & | & | & | & | & | & \\
\hline
\end{array}
\]

Result:

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c}
\hline
& & & & & & & & & & \\
\hline
\lfloor u / 2^k \rfloor & | & | & | & | & | & | & | & | & \\
\hline
\end{array}
\]

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 \( x < 0 \)

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c}
\hline
& & & & & & & & & & \\
\hline
x & | & k & | & | & | & | & | & | & \\
\hline
\end{array}
\]

Operands:

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c}
\hline
& & & & & & & & & & \\
\hline
x / 2^k & | & | & | & | & | & | & | & | & \\
\hline
\end{array}
\]

Result:

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c}
\hline
& & & & & & & & & & \\
\hline
\lfloor x / 2^k \rfloor & | & | & | & | & | & | & | & | & \\
\hline
\end{array}
\]

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>15213</td>
<td>15213</td>
<td>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>7606.5</td>
<td>7606</td>
<td>1D B6</td>
<td>00011101 10111010</td>
</tr>
<tr>
<td>950.8125</td>
<td>950</td>
<td>03 B6</td>
<td>00000001 10111010</td>
</tr>
<tr>
<td>59.4257813</td>
<td>59</td>
<td>00 3B</td>
<td>00000000 00111011</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>-15213</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>-7606.5</td>
<td>-7607</td>
<td>E2 49</td>
<td>11100010 01001001</td>
</tr>
<tr>
<td>-950.8125</td>
<td>-951</td>
<td>FC 49</td>
<td>11111100 01001001</td>
</tr>
<tr>
<td>-59.4257813</td>
<td>-60</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 $\left\lfloor x \div 2^k \right\rfloor$ (Round Toward 0)
  - Compute as $\left\lfloor (x+2^{k-1}) \div 2^k \right\rfloor$
    - In C: $(x + (1<<k)-1) >> k$
  - Biases dividend toward 0

Case 1: No rounding

<table>
<thead>
<tr>
<th>Dividend:</th>
<th>$u$ $\div 2^k$</th>
<th>$+2^{k-1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 1 ... 1 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Divisor:</th>
<th>$l$ $\div 2^k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 ... 1 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>

Biasing has no effect

Correct Power-of-2 Divide (Cont.)

Case 2: Rounding

Dividend: $x$

$+2^k - 1$

$0 1 ... 1 0 0 0$

Divisor: $x \div 2^k$

$1 0 1 ... 1 0 0 0$

Incremented by 1

Binary Point

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

Integer C Puzzles

- Assume 32-bit word size, two’s complement integers
- For each of the following C expressions: true or false? Why?
  - `x < 0`  `⇒ (x*2 < 0)`
  - `ux >= 0`
  - `x & 7 == 7`  `⇒ (x<<30) < 0`
  - `ux > -1`
  - `x > y`  `⇒ -x < -y`
  - `x * x >= 0`
  - `x > 0 && y > 0`  `⇒ x+y > 0`
  - `x == 0`  `⇒ -x <= 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;
```

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)

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
  - Big Endian: Sun, PPC Mac, Internet
    - Least significant byte has highest address
  - Little Endian: 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

<table>
<thead>
<tr>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>23</td>
<td>45</td>
<td>67</td>
</tr>
</tbody>
</table>

Little Endian

<table>
<thead>
<tr>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td>67</td>
<td>45</td>
<td>23</td>
<td>01</td>
</tr>
</tbody>
</table>

Representing Integers

<table>
<thead>
<tr>
<th>Decimal: 15213</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary: 0011 1011 0110 1101</td>
</tr>
<tr>
<td>Hex: 3 B 6 D</td>
</tr>
</tbody>
</table>

```
int A = 15213;
```

```
IA32, x86-64 | Sun
```

```
6D 3B 00
```

```
IA32, x86-64 | Sun
```

```
93 C4 FF FF
```

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	0x%.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):

```
int a = 15213;
0x11ffffffcb8 0x6d
0x11ffffffcb9 0x3b
0x11ffffffcb0 0x00
0x11ffffffcb1 0x00
```
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>c3 ab 12 00 00</td>
<td>add $0x12ab,%ebx</td>
</tr>
<tr>
<td>804836c:</td>
<td>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

Representing Pointers

- 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

Code Security Example

- **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
malloc(ele_cnt * 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 */
        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?**

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

$$A\&\sim B \lor \sim A\& B$$

Connection when

$$A\&\sim B \lor \sim A\& B = A^B$$
### 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);
    ...
}

### 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

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

Mathematical Properties of TAdd

- **Isomorphic Group to unsigneds with UAdd**
  - \(TAdd_w(u, v) = U2T(UAdd_w(T2U(u), T2U(v)))\)
    - 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

\[
TComp_w(u) = \begin{cases} 
 -u & u \neq TMin_w \\
 TMin_w & u = TMin_w
\end{cases}
\]

Compiled Multiplication Code

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

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

Compiled Unsigned Division Code

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

- 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

<table>
<thead>
<tr>
<th>Test</th>
<th>Machine Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>test1 %eax, %eax</td>
<td>js L4</td>
</tr>
<tr>
<td>L3:</td>
<td>sarl $3, %eax</td>
</tr>
<tr>
<td>ret</td>
<td>addl $7, %eax</td>
</tr>
<tr>
<td>jmp L3</td>
<td></td>
</tr>
</tbody>
</table>

Explanation

- 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
  - $-x + 1 = -x$

- **Complement**
  - Observation: $-x + x = 1111...111 = -1$
  - $x = 10011101$
  - $-x = 01100010$
  - $-1 = 11111111$

- **Complete Proof?**

Complement & Increment Examples

$x = 15213$

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>11101010110000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$-x$</th>
<th>$-15214$</th>
<th>10010011000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y$</td>
<td>$-15213$</td>
<td>10010011000000</td>
</tr>
</tbody>
</table>

$x = 0$

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>00000000000000</td>
</tr>
<tr>
<td>$-0$</td>
<td>$-1$</td>
<td>FF FF FF FF FF</td>
</tr>
<tr>
<td>$-0+1$</td>
<td></td>
<td>11111111111111</td>
</tr>
</tbody>
</table>

| $-0+1$  |     | 00000000000000 |

```
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.,
    \[ u > 0 \implies u + v > v \]
    \[ u > 0, v > 0 \implies u \cdot v > 0 \]
  - These properties are not obeyed by two’s comp. arithmetic
    \[ \text{TMax} + 1 = \text{TMin} \]
    \[ 15213 \times 30426 = -10030 \] (16-bit words)