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
  
  ![0.0V, 0.2V, 0.9V, 1.1V]  

For example, can count in binary

- Base 2 Number Representation
  - Represent 15213\text{_{10}} as 11101101101101\text{_{2}}
  - Represent 1.20\text{_{10}} as 1.0011001100110011\ldots\text{_{2}}
  - Represent 1.5213 \times 10^{4}\text{_{10}} as 1.1101101101101\text{_{2}} \times 2^{13}

Encoding Byte Values

- Byte = 8 bits
  - Binary 00000000\text{_{2}} to 11111111\text{_{2}}
  - Decimal: 0 to 255\text{_{10}}
  - Hexadecimal 00\text{_{16}} to FF\text{_{16}}
  - Base 16 number representation
  - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
  - Write FA1D37Bu in C as
    - 0xFA1D37
    - 0xFA1D37b

Aside: ASCII table

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>ASCII</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0x2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>127</td>
<td>0x7F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>0x80</td>
<td></td>
<td></td>
</tr>
<tr>
<td>129</td>
<td>0x81</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>255</td>
<td>0xFF</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: \( \wedge \))
    - \( A \wedge B = 1 \) when both \( A = 1 \) and \( B = 1 \)
  - Or (math: \( \vee \))
    - \( A \vee B = 1 \) when either \( A = 1 \) or \( B = 1 \)
  - Not (math: \( \neg \))
    - \( \neg A = 1 \) when \( A = 0 \)
    - \( A \oplus B = 1 \) when either \( A = 1 \) or \( B = 1 \), but not both

### General Boolean Algebras

- Operate on bit vectors
  - Operations applied bitwise
  - All of the properties of Boolean algebra apply

### Example: Representing & Manipulating Sets

- **Representation**
  - Width \( w \) bit vector represents subsets of \( \{0, \ldots, w-1\} \)
  - \( a_i = 1 \) if \( i \in A \)

  - \( \begin{bmatrix} 01101001 & \{0, 3, 5, 6\} \\ 76543210 & 0 \end{bmatrix} \)

  - \( \begin{bmatrix} 01010101 & \{0, 2, 4, 6\} \\ 76543210 & 0 \end{bmatrix} \)

- **Operations**
  - \& Intersection \( 01000001 \) \( \{0, 6\} \)
  - | Union \( 01111101 \) \( \{0, 2, 3, 4, 6\} \)
  - \( ^\wedge \) 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)**
  - \( \sim 0x41 \rightarrow 0xBE \)
  - \( \neg 0x00 \rightarrow 0xFF \)
  - \( \sim 0x00000000 \rightarrow 0xFFFFFFFF \)
  - \( 0x69 \& 0x55 \rightarrow 0x41 \)
  - \( 0x1001011_2 \| 01010101_2 \rightarrow 01111110_2 \)
  - \( 0x09 \| 0x15 \rightarrow 0x1D \)
  - \( 0x1010101_2 \| 01010101_2 \rightarrow 01111110_2 \)
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 && 0x55 → 0x01

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
  - Signed shift into or out of sign bit (i.e., arith. behavior not assured)

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

<table>
<thead>
<tr>
<th>Unsigned</th>
<th>Two’s Complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>R2U(x) = \sum_{i=0}^{w-1} x_i 2^i</td>
<td>R2C(x) = -x_{w-1} 2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i</td>
</tr>
</tbody>
</table>

- 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>00111011 01010101</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>04 99 11000100 10010111</td>
</tr>
</tbody>
</table>

- Sign Bit
  - For 2’s complement, most significant bit indicates sign
  - 0 for nonnegative
  - 1 for negative

Binary Number Property

Claim

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

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

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

= 2^w

Four-bit Example, unsigned

\[
\begin{array}{cccccccc}
8 & 4 & 2 & 1 \\
1 & 1 & 1 & 1 & = 15_{10}
\end{array}
\]

This approach can represent 0 through 15
Four-bit Example, sign + magnitude

**Sign + magnitude:**

1 1 1 1 = 15 _10_

**Negate if 1**

This approach can represent -7 through 7.

Disadvantages: special cases, 0.

Four-bit Example, two’s complement

**Two’s complement:**

1 1 1 1 = -1 _10_

-8 4 2 1

This approach can represent -8 through 7.

Four-bit Example

| Unsigned: 1 1 1 1 | = 15 _10_ |
|-------------------|
| Two’s complement: 1 1 1 1 | = -1 _10_ |
|                   -8 4 2 1 |

Encoding Integers

<table>
<thead>
<tr>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two’s Complement</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C short 2 bytes long</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sign Bit</td>
</tr>
</tbody>
</table>

For 2’s complement, most significant bit indicates sign
- 0 for nonnegative
- 1 for negative

Four-complement Encoding Example (Cont.)

<table>
<thead>
<tr>
<th>x</th>
<th>= 15213: 00111011 01101101</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Weight</th>
<th>15213</th>
<th>15213</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>32</td>
</tr>
<tr>
<td>64</td>
<td>1</td>
<td>64</td>
</tr>
<tr>
<td>128</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>256</td>
<td>1</td>
<td>256</td>
</tr>
<tr>
<td>512</td>
<td>1</td>
<td>512</td>
</tr>
<tr>
<td>1024</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2048</td>
<td>1</td>
<td>2048</td>
</tr>
<tr>
<td>4096</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8192</td>
<td>1</td>
<td>8192</td>
</tr>
<tr>
<td>16384</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>32768</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Values for W = 16

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>00000000</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>00000001</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td>00000010</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td>00000011</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>00000100</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>00000101</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>00000110</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>00000111</td>
</tr>
</tbody>
</table>

Values for W = 32

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>00000000</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>00000001</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td>00000010</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td>00000011</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>00000100</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>00000101</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>00000110</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>00000111</td>
</tr>
</tbody>
</table>

Sign Bit

- 0 for nonnegative
- 1 for negative
**Values for Different Word Sizes**

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

**Observations**
- $|TMin| = TMax + 1$
- Asymmetric range
- $U\text{Max} = 2 \times T\text{Max} + 1$

**C Programming**
- Include <limits.h>
- Declares constants, e.g.,
  - ULONG_MAX
  - LONG_MAX
  - LONG_MIN
- Values platform specific

**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

**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 complement integer

**Mapping Between Signed & Unsigned**

- **Two’s Complement**
  - $x \rightarrow \text{T2U}(x)$
  - $\text{T2U}(x) \rightarrow \text{U2T}(x)$

- **Unsigned**
  - $x \rightarrow \text{U2B}(x)$
  - $\text{U2B}(x) \rightarrow \text{B2U}(x)$

- **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>-7</td>
</tr>
<tr>
<td>1010</td>
<td>-6</td>
<td>-6</td>
</tr>
<tr>
<td>1011</td>
<td>-5</td>
<td>-5</td>
</tr>
<tr>
<td>1100</td>
<td>-4</td>
<td>-4</td>
</tr>
<tr>
<td>1101</td>
<td>-3</td>
<td>-3</td>
</tr>
<tr>
<td>1110</td>
<td>-2</td>
<td>-2</td>
</tr>
<tr>
<td>1111</td>
<td>-1</td>
<td>-1</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>-7</td>
</tr>
<tr>
<td>1010</td>
<td>-6</td>
<td>-6</td>
</tr>
<tr>
<td>1011</td>
<td>-5</td>
<td>-5</td>
</tr>
<tr>
<td>1100</td>
<td>-4</td>
<td>-4</td>
</tr>
<tr>
<td>1101</td>
<td>-3</td>
<td>-3</td>
</tr>
<tr>
<td>1110</td>
<td>-2</td>
<td>-2</td>
</tr>
<tr>
<td>1111</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

**Maintain Same Bit Pattern**

- $\text{B2U}(X) \rightarrow \text{U2B}(X)$
- $\text{B2T}(X) \rightarrow \text{T2B}(X)$

**Evaluate**
- Same encodings for nonnegative values
- Every bit pattern represents unique integer value
- Each representable integer has unique bit encoding
- $U2B(x) = B2U^{-1}(x)$
- Bit pattern for unsigned integer
- $T2B(x) = B2T^{-1}(x)$
- Bit pattern for two’s complement integer

**Can Invert Mappings**

- Summary
Relation between Signed & Unsigned

Two's Complement

<table>
<thead>
<tr>
<th>x</th>
<th>T2B</th>
<th>U2X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>-1</td>
<td>111</td>
<td>111</td>
</tr>
</tbody>
</table>

Maintain Same Bit Pattern

Large negative weight

becomes

Large positive weight

Conversion Visualized

- Two's Comp. → Unsigned
  - Ordering Inversion
  - Negative → Big Positive

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: TMIN = -2,147,483,648, TMAX = 2,147,483,647

Casting Signed ↔ Unsigned: Basic Rules

- Bit pattern is maintained
- But reinterpreted
- Can have unexpected effects: adding or subtracting 2^w

Summary

- 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:
    \[
    X' = \begin{cases} \end{cases} x_{w-1}, \ldots, x_{w-1}, x_{w-2}, \ldots, x_0
    \]
    \( k \) copies of MSB

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>15213</td>
<td>0D 6D</td>
<td>00111101 01101101</td>
</tr>
<tr>
<td>-15213</td>
<td>C4 93</td>
<td>11111111 11111111</td>
</tr>
</tbody>
</table>

Summary: Expanding, Truncating: Basic Rules

- Expanding (e.g., short int to int)
  - Signed: sign extension
  - Both yield expected result
- Truncating (e.g., unsigned to unsigned short)
  - Both yield expected result

Unsigned Addition

- Operands: \( w \) bits
- True Sum: \( w+1 \) bits
- Discard Carry: \( w \) bits
- Standard Addition Function
  - Ignores carry output
- Implements Modular Arithmetic
  \[
  s = U\text{Add}_w(u, v) = u + v \mod 2^w
  \]
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

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 $\text{UAdd}_w(0, u) = u$
  - Every element has additive inverse
    - Let $\text{UComp}_w(u) = 2^w - u$
    - $\text{UAdd}_w(u, \text{UComp}_w(u)) = 0$

Two’s Complement Addition

- 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

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 \( T_{\text{Add}} \)

- Isomorphic Group to unsigned with \( U_{\text{Add}} \)
  \[ T_{\text{Add}}(u, v) = U_{\text{Add}}(u_2 U(u), U(v)) \]
  \( - \) Since both have identical bit patterns

- Two’s Complement Under \( T_{\text{Add}} \) Forms a Group
  \( - \) Closed, Commutative, Associative, 0 is additive identity
  \( - \) Every element has additive inverse
  \[ T_{\text{Comp}}(u) = \begin{cases} -u & u \neq T_{\text{Min}}w \\ T_{\text{Min}}w & u = T_{\text{Min}}w \end{cases} \]

Characterizing \( T_{\text{Add}} \)

- 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)

Sign bit, signed ordering

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

Negation: Complement & Increment

- Claim: Following Holds for 2’s Complement
  \[ -x + 1 = -x \]
- Complement
  \( - \) Observation: \( -x + x == 1111...111 == -1 \)
  \[ x \| \overline{1111...111} \]
  \[ + -x \| \overline{1111...111} \]
  \[ -1 \| 1111...111 \]
### 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>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>~x = -15214</td>
<td>C4 92</td>
<td>11000100 10010010</td>
</tr>
<tr>
<td>x+1 = -15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
</tbody>
</table>

**y = -15213**

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00000000 00000000</td>
</tr>
<tr>
<td>-1</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>-1+1</td>
<td>0</td>
<td>00000000 00000000</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 \( 2^w \) bits
      - Exact results range: \( 0 \leq x \cdot y \leq (2^w - 1) \cdot 2^w = 2^{2w} - 2^w + 1 \)
    - Two’s complement min (negative): Up to \( 2^w - 1 \) bits
      - Exact results range: \( x \cdot y \geq (-2^w - 1) \cdot (2^w - 1) = -2^{2w} - 2^w + 1 \)
  - Two’s complement max (positive): Up to \( 2^w \) bits, but only for \( (T_{\text{Min}})^w \)
      - Exact results range: \( x \cdot y \leq (2^w - 1) \cdot 2^w = 2^{2w} - 2^w + 1 \)

- **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

- **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

- **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

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

### XDR Code

```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 */
    void *result = malloc(ele_cnt * ele_size);
    if (result == NULL)
        /* malloc failed */
    else
        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 NULL;
}
```
XDR Vulnerability

malloc(ele_cnt * ele_size)

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

(2^{20} + 1) * 2^{12} = 2^{20} * 2^{12} + 2^{12} * 2^{12} ≡ 2^{12}

How can I make this function secure?

Power-of-2 Multiply with Shift

Operation
- u << k gives u * 2^k
- Both signed and unsigned

Operands: w bits

<table>
<thead>
<tr>
<th>True Product: wk bits</th>
<th>TMult(w, 2^k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
<td>UMult(u, 2^k)</td>
</tr>
</tbody>
</table>

Discard k bits: w bits

<table>
<thead>
<tr>
<th>2^k</th>
<th>2^k</th>
</tr>
</thead>
<tbody>
<tr>
<td>2^w</td>
<td>2^w</td>
</tr>
</tbody>
</table>

Examples
- u << 3 == u * 8
- (u << 5) - (u << 3) == u * 24
- Most machines shift and add faster than multiply

Compiled Multiplication Code

C Function
long mul12(long x)
{
    return x*12;
}

Compiled Arithmetic Operations
- leaq (%rax, %rax, 2), %rax
- salq $2, %rax

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:
- round(x + k) = round(x) + k (k integer), "translation invariance"
- round(-x) = -round(x) "negation invariance"

Floor: \[ \lfloor x \rfloor \] always round down (to \( -\infty \)):
- \[ \lfloor 2.0 \rfloor = 2, \lfloor 1.7 \rfloor = 1, \lfloor -2.2 \rfloor = -3 \]

Ceiling: \[ \lceil x \rceil \] 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

Division 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/2 is un-representable, and so undefined
  - On x86 this too is a default-fatal exception

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
  - Overflow in signed operations
  - Shifts of negative values

Things you do in this section of the course!
### Unsigned Power-of-2 Divide with Shift

**Quotient of Unsigned by Power of 2**
- $u >> k$ gives $u / 2^k$
- Uses logical shift

<table>
<thead>
<tr>
<th>$u$</th>
<th>$u / 2^k$</th>
<th>Binary Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u$</td>
<td>$u &gt;&gt; k$</td>
<td></td>
</tr>
</tbody>
</table>

### Compiled Unsigned Division Code

**C Function**

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

**Compiled Arithmetic Operations**

```
shrq $3, rax
```

### Signed Power-of-2 Divide with Shift

**Quotient of Signed by Power of 2**
- $x >> k$ gives $x / 2^k$
- Uses arithmetic shift
- Rounds wrong direction when $u < 0$

<table>
<thead>
<tr>
<th>$x$</th>
<th>$x &gt;&gt; k$</th>
<th>Binary Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$x &gt;&gt; k$</td>
<td></td>
</tr>
</tbody>
</table>

### Compiled Signed Division Code

**C Function**

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

**Compiled Arithmetic Operations**

```
testq rax, rax
ja 14
l3:
    addq $7, rax
    jmp 13
```

### Correct Power-of-2 Divide

**Quotient of Negative Number by Power of 2**
- Want $[x / 2^k]$ (Round Toward 0)
- Compute as $\lfloor (x+2^{k-1}) / 2^k \rfloor$
- In C $(x + (1<k) - 1) >> k$
- Biases dividend toward 0

**Case 1: No rounding**

<table>
<thead>
<tr>
<th>$u$</th>
<th>$u / 2^k$</th>
<th>Binary Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u$</td>
<td>$u &gt;&gt; k$</td>
<td></td>
</tr>
</tbody>
</table>

**Case 2: Rounding**

<table>
<thead>
<tr>
<th>$u$</th>
<th>$u / 2^k$</th>
<th>Binary Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u$</td>
<td>$u &gt;&gt; k$</td>
<td></td>
</tr>
</tbody>
</table>

### Correct Power-of-2 Divide (Cont.)

**Biasing has no effect**

### Compiled Signed Division Code

**C Function**

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

**Compiled Arithmetic Operations**

```
testq rax, rax
ja 14
l3:
    addq $7, rax
    jmp 13
```

### Compiled Unsigned Division Code

**C Function**

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

**Compiled Arithmetic Operations**

```
shrq $3, rax
```

### For Java Users

- Logical shift written as >>>
- For Java Users
  - Uses arithmetic shift for int
  - Arithmetic shift for int
  - Arith. shift written as >>
Remainder operator

- Written as % in C
- \( x \mod y \) is the remainder after division \( x / y \)
- E.g., \( x \mod 10 \) is the lowest digit of non-negative \( x \)
- Behavior for negative values matches /'s rounding toward zero
  * \( b \times (a / b) + (a \mod b) = a \)
- 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)

Why Should I Use Unsigned?

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

Counting Down with Unsigned

- Proper way to use unsigned as loop index
  \[
  \text{unsigned } i;\text{for (} i = \text{cnt-2}; i < \text{cnt}; i--) \
  \text{a[}i\text{]} += \text{a[}i+1\text{]};
  \]
- See Robert Seacord, Secure Coding in C and C++
  * C Standard guarantees that unsigned addition will behave like modular arithmetic
    * \( 0-1 \rightarrow UMax \)
- Even better
  \[
  \text{size_t } i;\text{for (} i = \text{cnt-2}; i < \text{cnt}; i--) \
  \text{a[}i\text{]} += \text{a[}i+1\text{]};
  \]
  * Data type size_t defined as unsigned value with length = word size
  * Code will work even if \( \text{cnt} = \text{UMax} \)
  * What if \( \text{cnt} \) is signed and < 0?

Why Should I Use Unsigned? (cont.)

- 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
- 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 X 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)

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>

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

Byte Ordering Example

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

Representing Integers

- Two’s complement representation

Decimal: 15213
Binary: 0011 1011 0110 1101
Hex: 3B 6D

int A = 15213:

long int C = 15213:

int B = -15213:
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, size_t len){
  size_t i;
  for (i = 0; i < len; i++)
    printf("%p \t0x%.2x \n",start+i, start[i]);
  printf("\n");}
```

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

**show_bytes Execution Example**

```c
int a = 15213;
void (*) &a, sizeof(int));
```

Result (Linux x86-64):

```c
int a = 15213;
0x7fffb7f71dbc 6d
0x7fffb7f71dbd 3b
0x7fffb7f71dbe 00
0x7fffb7f71dbf 00
```

Representing Pointers

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

<table>
<thead>
<tr>
<th>Sun</th>
<th>IA32</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>EF</td>
<td>AC</td>
<td>3C</td>
</tr>
<tr>
<td>FP</td>
<td>2B</td>
<td>1B</td>
</tr>
<tr>
<td>PR</td>
<td>P5</td>
<td>PE</td>
</tr>
<tr>
<td>2C</td>
<td>FF</td>
<td>B2</td>
</tr>
<tr>
<td></td>
<td>F0</td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>00</td>
<td>00</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 / has code 0x30+i
    - String should be null-terminated
      - Final character = 0
  - **Compatibility**
    - Byte ordering not an issue

Reading Byte-Reversed Listings

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Code</th>
<th>Assembly Rendition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0048365</td>
<td>5b</td>
<td>pop %ebx</td>
</tr>
<tr>
<td>0048366</td>
<td>81 c3 ab 12 00 00</td>
<td>add 0x12ab,%ebx</td>
</tr>
<tr>
<td>004836c</td>
<td>83 c3 28 00 00 00</td>
<td>cmpl 0x28,%ebx</td>
</tr>
</tbody>
</table>

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

Integer C Puzzles

1. \(x < 0 \implies (x\times 2) < 0\)
2. \(ux > -1\)
3. \(x > 0 \&\& y > 0 \implies x + y > 0\)

Initialization

```c
4. (x|-x)>>31 == -1
```

```c
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```
Bonus: More Integer C Puzzles

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

Initialization

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