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 as 111011011011012
  - Represent 1.2010 as 1.0011001100110011[0011]_{-2}
  - Represent 1.5213 x 10^{4} as 1.1101101101101_{1} x 2^{13}

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

Encoding Byte Values

- **Byte = 8 bits**
  - Binary 00000000 to 111111112
  - Decimal: 0s to 255s
  - Hexadecimal 00_{16} to FF_{16}
    - Base 16 number representation
    - Use characters '0' to '9' and 'A' to 'F'
    - Write FA1D37B_{16} in C as
      - 0xFA1D378
      - 0xFA1D37B
# 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**

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

**Or**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Not**

<table>
<thead>
<tr>
<th>A</th>
<th>~A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Exclusive-Or (Xor)**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A ^ B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</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

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- 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 \(01000001\) \(\{0, 6\}\)
  - \(| Union \) \(01111101\) \(\{0, 2, 3, 4, 5, 6\}\)
  - \(^\wedge Symmetric difference\) \(00111100\) \(\{2, 3, 4, 5\}\)
  - \(~ Complement\) \(10101010\) \(\{1, 3, 5, 7\}\)

Bit-Level Operations in C

- Operations \&, \(|, \sim, \^\) Available in C
  - Apply to any "integral" data type
  - \(\{\text{long, int, short, char, unsigned}\}\)
  - View arguments as bit vectors
  - Arguments applied bit-wise

- Examples (Char data type)
  - \(~0x41\) \(\rightarrow 0xBE\)
  - \(~0x00\) \(\rightarrow 0xFF\)
  - \(~0x00\) \(\rightarrow 0xFF\)
  - \(~0x69 \& 0x55\) \(\rightarrow 0x41\)
  - \(~0x69 | 0x55\) \(\rightarrow 0x7D\)

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)
  - \(!0x41\) \(\rightarrow 0x00\)
  - \(!0x00\) \(\rightarrow 0x01\)
  - \(!0x41\) \(\rightarrow 0x01\)
  - \(0x69 \&\& 0x55\) \(\rightarrow 0x01\)
  - \(0x69 || 0x55\) \(\rightarrow 0x01\)
  - \(p \&\& \!p\) (avoids null pointer access)
Shift Operations

- **Left Shift:** $x << y$
  - Shift bit-vector $x$ left $y$ positions
  - Throw away extra bits on left
    - Fill with 0’s on right
- **Right Shift:** $x >> y$
  - Shift bit-vector $x$ right $y$ positions
  - Throw away extra bits on right
  - Logical shift
    - Fill with 0’s on left
  - Arithmetic shift
    - Replicate most significant bit on left
- **Undefined Behavior**
  - Shift amount < 0 or ≥ word size

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

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>-15213</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>04 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.)

<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>0</td>
<td>0</td>
</tr>
<tr>
<td>4096</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8192</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>16384</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>32768</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Sum</td>
<td>15213</td>
<td>-15213</td>
</tr>
</tbody>
</table>
Numeric Ranges

- **Unsigned Values**
  - $U_{\text{Min}} = 0$
  - $U_{\text{Max}} = 2^w - 1$

- **Two's Complement Values**
  - $T_{\text{Min}} = -2^{w-1}$
  - $T_{\text{Max}} = 2^{w-1} - 1$

- **Other Values**
  - Minus 1

Values for $W = 16$

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>$U_{\text{Max}}$</td>
<td>65535</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>$T_{\text{Max}}$</td>
<td>32767</td>
<td>FF FF</td>
<td>01111111 11111111</td>
</tr>
<tr>
<td>$T_{\text{Min}}$</td>
<td>-32768</td>
<td>00 00</td>
<td>10000000 00000000</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>

Values for Different Word Sizes

<table>
<thead>
<tr>
<th>W</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>$U_{\text{Max}}$</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>$T_{\text{Max}}$</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>$T_{\text{Min}}$</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**
  - $|T_{\text{Min}}| = T_{\text{Max}} + 1$
  - 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

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

\[ \begin{array}{ccc}
\text{T2U} & \xrightarrow{X} & \text{U2T} \\
\text{T2B} & \xrightarrow{X} & \text{B2T} \\
\text{Maintain Same Bit Pattern} & & \\
\end{array} \]

Unsigned

\[ \begin{array}{ccc}
\text{U2T} & \xrightarrow{X} & \text{B2T} \\
\text{Maintain Same Bit Pattern} & & \\
\end{array} \]

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>-8</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>-7</td>
<td>8</td>
</tr>
<tr>
<td>1001</td>
<td>-6</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>-5</td>
<td>10</td>
</tr>
<tr>
<td>1011</td>
<td>-4</td>
<td>11</td>
</tr>
<tr>
<td>1100</td>
<td>-3</td>
<td>12</td>
</tr>
<tr>
<td>1101</td>
<td>-2</td>
<td>13</td>
</tr>
<tr>
<td>1110</td>
<td>-1</td>
<td>14</td>
</tr>
<tr>
<td>1111</td>
<td></td>
<td>15</td>
</tr>
</tbody>
</table>

Relation between Signed & Unsigned

Two’s Complement

\[ \begin{array}{ccc}
\text{T2U} & \xrightarrow{X} & \text{B2U} \\
\text{T2B} & \xrightarrow{X} & \text{B2U} \\
\text{Maintain Same Bit Pattern} & & \\
\end{array} \]

Unsigned

\[ \begin{array}{ccc}
\text{U2T} & \xrightarrow{X} & \text{B2T} \\
\text{Maintain Same Bit Pattern} & & \\
\end{array} \]

- Large negative weight becomes large positive weight

\[ +/- 16 \]

Carnegie Mellon
**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: TMIN = -2,147,483,648, TMAX = 2,147,483,647

**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:
    \( X' = x_{w-1}, \ldots, x_{w-1}, x_{w-2}, \ldots, x_0 \)

\[ \begin{array}{c|c|c|c|c|c}
  x & \text{Hex} & \text{Binary} \\
  \hline
  15213 & 3B 6D & 00111011 01101101 \\
  \text{ix} & 15213 & 00 00 3B 6D & 00000000 00000000 00111011 01101101 \\
  \text{iy} & -15213 & C4 93 & 11000100 100100011 \\
  \text{iy} & -15213 & FF FF C4 93 & 11111111 11111111 111000100 10010011 \\
\end{array} \]

- 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: $u$ bits $+ v$ bits
- True Sum: $w+1$ bits
- Discard Carry: $w$ bits

\[ UAdd_u(u, v) \]

| True Sum: $u + v$ mod $2^w$ |

- Standard Addition Function
  - Ignores carry output
- Implements Modular Arithmetic
  \[ s = UAdd_u(u, v) = u + v \mod 2^w \]

Visualizing (Mathematical) Integer Addition

- Integer Addition
  - 4-bit integers $u, v$
  - Compute true sum $Add_4(u, v)$
  - Values increase linearly with $u$ and $v$
  - Forms planar surface

Visualizing Unsigned Addition

- Wraps Around
  - If true sum $\geq 2^w$
  - At most once
Two’s Complement Addition

Operands: $w$ bits

$$u \begin{array}{c} \vdots \\ \vdots \\ \vdots \end{array}$$

True Sum: $w+1$ bits

$$u + v \begin{array}{c} \vdots \\ \vdots \\ \vdots \end{array}$$

Discard Carry: $w$ bits

$$\text{TAdd}(u, v) \begin{array}{c} \vdots \\ \vdots \\ \vdots \end{array}$$

- TAdd and UAdd have Identical Bit-Level Behavior
  - Signed vs. unsigned addition in C:
    ```c
    int s, t, u, v;
    s = (int) ((unsigned) u + (unsigned) v);
    t = u + v
    ```
  - Will give $s == t$

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

TAdd Overflow

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

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

  - TAdd Result
    - $011...1$ (PosOver)
    - $100...0$ (NegOver)

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

- **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

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

True Product: \( 2^w \) bits

\[ u \cdot v \begin{array}{c}
\times \\
v
\end{array} \]

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

UMult\(_w\)(\( u \), \( v \))

- 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

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

True Product: \( 2^w \) bits

\[ u \cdot v \begin{array}{c}
\times \\
v
\end{array} \]

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

TMult\(_w\)(\( u \), \( v \))

- 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 \( u \cdot 2^k \)
  - Both signed and unsigned
  - Operands: \( w \) bits
  - True Product: \( w+k \) bits
  - Discard \( k \) bits: \( \) bits

\[ \text{UMult}_w( u, 2^k ) \]

- Examples
  - \( u \ll 3 \) \( = \) \( u \cdot 8 \)
  - \( u \ll 5 - u \ll 3 \) \( = \) \( u \cdot 24 \)
  - 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

- Operands:
  - \( u \ll k \) bits
  - True Product: \( w+k \) bits
  - Discard \( k \) bits: \( \) bits

\[ \text{TMult}_w( u, 2^k ) \]

- Division:
  - Result:

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

\[ \begin{array}{|c|c|c|c|}
\hline
\text{Division} & \text{Computed} & \text{Hex} & \text{Binary} \\
\hline
x & 15213 & 15213 & 00111011 01101101 \\
x \gg 1 & 7606.5 & 7606 & 00011011 01101110 \\
x \gg 4 & 950.8125 & 950 & 00000011 01101110 \\
x \gg 8 & 59.42578125 & 59 & 00 3B 00111000 01110101 \\
\hline
\end{array} \]
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 $u < 0$

| Operands: $x$ | Binary Point $2^k$
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$x / 2^k$</td>
<td>$x / 2^k$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y &gt;&gt; 1$</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>$y &gt;&gt; 4$</td>
<td>-950.8125</td>
<td>FC 49</td>
<td>11111100 01001001</td>
</tr>
<tr>
<td>$y &gt;&gt; 8$</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 $\lfloor x / 2^k \rfloor$ (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

Dividend: $u$  
Divisor: $\lfloor u / 2^k \rfloor$

Biasing has no effect

Correct Power-of-2 Divide (Cont.)

Case 2: Rounding

Dividend: $x + 2^k - 1$
Divisor: $\lfloor x / 2^k \rfloor$

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
  ```
  #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

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>

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)

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

```
int A = 15213;
```

IA32, x86-64

<table>
<thead>
<tr>
<th>6D</th>
<th>3B</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

Sun

<table>
<thead>
<tr>
<th>6D</th>
<th>3B</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

```
long int C = 15213;
```

IA32, x86-64

<table>
<thead>
<tr>
<th>6D</th>
<th>3B</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

Sun

<table>
<thead>
<tr>
<th>6D</th>
<th>3B</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

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

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

Printf directives:

- %p: Print pointer
- %x: Print Hexadecimal

show_bytes Execution Example

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

Result (Linux):

```
int a = 15213;
0x11ffffcb8 0x6d
0x11ffffcbb 0x3b
0x11ffffcba 0x00
0x11ffffcbb 0x00
```
**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>D4</td>
<td>0C</td>
</tr>
<tr>
<td>FF</td>
<td>F8</td>
<td>89</td>
</tr>
<tr>
<td>FB</td>
<td>FF</td>
<td>EC</td>
</tr>
<tr>
<td>2C</td>
<td>FF</td>
<td>FF</td>
</tr>
<tr>
<td></td>
<td>7F</td>
<td></td>
</tr>
<tr>
<td></td>
<td>00</td>
<td></td>
</tr>
</tbody>
</table>

Different compilers & machines assign different locations to objects

---

**Representing Strings**

```c
char S[6] = "18243";
```

- **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

---

**Integer C Puzzles**

- x < 0
- ux >= 0
- x & 7 == 7
- ux > -1
- x > y
- x * x >= 0
- x > 0 && y > 0
- x >= 0
- x == 0
- x > y
- (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;
```