BITS, BYTES, AND INTEGERS

SYSTEMS 1
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
- Making ints from bytes
- Summary
Encoding Byte Values

- **Byte = 8 bits**
  - Binary: $00000000_2$ to $11111111_2$
  - Decimal: $0_{10}$ to $255_{10}$
  - 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
      - `0xFA1D37B`
      - `0xfa1d37b`
Boolean Algebra

- Developed by George Boole in 19th Century
- Algebraic representation of logic
  - Encode “True” as 1 and “False” as 0

### And
- \( A \& B = 1 \) when both \( A=1 \) and \( B=1 \)

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

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

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</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
- \( \sim A = 1 \) when \( A=0 \)

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

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

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

- Operate on Bit Vectors
  - Operations applied bitwise
    
    \[
    \begin{array}{ccc}
    01101001 & 01101001 & 01101001 \\
    \& 01010101 & \lor 01010101 & ^ 01010101 & \sim 01010101 \\
    01000001 & 01111101 & 00111100 & 10101010
    \end{array}
    \]

- All of the Properties of Boolean Algebra Apply
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 [1 byte])
  - In gdb, p/t 0xE prints 1110
  - ~0x41 → 0xBE
    - ~01000001 → 10111110
  - ~0x00 → 0xFF
    - ~00000000 → 11111111
  - 0x69 & 0x55 → 0x41
    - 01101001 & 01010101 → 01000001
  - 0x69 | 0x55 → 0x7D
    - 01101001 | 01010101 → 01111101
Representing & Manipulating Sets

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

  - 01101001 \( \{0, 3, 5, 6\} \)
  - 76543210
  - **MSB** Least significant bit (LSB)

  - 01010101 \( \{0, 2, 4, 6\} \)
  - 76543210

- **Operations**
  - & Intersection \( \{0, 6\} \)
  - | Union \( \{0, 2, 3, 4, 5, 6\} \)
  - ^ Symmetric difference \( \{2, 3, 4, 5\} \)
  - ~ Complement \( \{1, 3, 5, 7\} \)
Contrast: Logic Operations in C

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

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

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

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

- **Undefined Behavior**
  - Shift amount $< 0$ or $\geq$ word size

<table>
<thead>
<tr>
<th>Argument $x$</th>
<th>$01100010$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$&lt;&lt;$ 3</td>
<td>$00010000$</td>
</tr>
<tr>
<td>Logical $&gt;&gt;$ 2</td>
<td>$00011000$</td>
</tr>
<tr>
<td>Arithmetic $&gt;&gt;$ 2</td>
<td>$00011000$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument $x$</th>
<th>$10100010$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$&lt;&lt;$ 3</td>
<td>$00010000$</td>
</tr>
<tr>
<td>Logical $&gt;&gt;$ 2</td>
<td>$00101000$</td>
</tr>
<tr>
<td>Arithmetic $&gt;&gt;$ 2</td>
<td>$11101000$</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

- Making ints from bytes

- Summary
## 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>
How to encode unsigned integers?

- Just use exponential notation (4 bit numbers)
  - $0110 = 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 6$
  - $1001 = 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 9$
  - (Just like $13 = 1 \times 10^1 + 3 \times 10^0$)

- No negative numbers, a single zero (0000)
How to encode signed integers?

- Want: Positive and negative values
- Want: Single circuit to add positive and negative values (i.e., no subtractor circuit)
- Solution: Two’s complement
- Positive numbers easy (4 bits)
  - $0110 = 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 6$
- Negative numbers a bit weird
  - $1 + -1 = 0$, so $0001 + X = 0$, so $X = 1111$
  - $-1 = 1111$ in two’s compliment
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>3B 6D 00111011 01101101</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>C4 93 11000100 10010011</td>
</tr>
</tbody>
</table>

**Sign Bit**

- For 2’s complement, most significant bit indicates sign
  - 0 for nonnegative
  - 1 for negative
Encoding Example (Cont.)

\[
\begin{align*}
x &= 15213: 00111011 01101101 \\
y &= -15213: 11000100 10010011
\end{align*}
\]

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

\[
\begin{align*}
\text{Sum} & \quad 15213 & \quad -15213
\end{align*}
\]
UnSigned & Signed Numeric Values

- **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

<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>
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
    - $-1$
    - $111...1$

### Values for $W = 16$

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>65535</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>TMax</td>
<td>32767</td>
<td>7F FF</td>
<td>01111111 11111111</td>
</tr>
<tr>
<td>Tmin</td>
<td>-32768</td>
<td>80 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>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
  - $|T_{\text{Min}}| = T_{\text{Max}} + 1$
    - 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
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
- Making ints from bytes
- Summary
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>

The mapping is as follows:

- Signed values range from -1 to -8.
- Unsigned values range from 0 to 15.
- The binary representation of the signed values is mapped to the corresponding unsigned values.

The mapping is symmetric with a range of +/- 16.

![Mapping Diagram](image-url)
Conversion Visualized

- 2’s Comp. → Unsigned
  - Ordering Inversion
  - Negative → Big Positive

2’s Complement Range

TMax

0

-1

-2

TMin

UMax

UMax - 1

TMax + 1

TMax

0

Unsigned Range
Negation: Complement & Increment

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

- **Complement**
  - **Observation:** \[ \sim x + x = 1111...111 = -1 \]

\[
\begin{array}{c}
\times & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\
+ & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\
\hline
& 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1
\end{array}
\]
## Complement & Increment Examples

### $x = 15213$

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>15213</td>
<td>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>$\sim x$</td>
<td>-15214</td>
<td>C4 92</td>
<td>11000100 10010010</td>
</tr>
<tr>
<td>$\sim x+1$</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>$y$</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
</tbody>
</table>

### $x = 0$

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00 00</td>
<td>00000000 00000000</td>
</tr>
<tr>
<td>$\sim 0$</td>
<td>-1</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>$\sim 0+1$</td>
<td>0</td>
<td>00 00</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>
Signed vs. Unsigned in C

- **Constants**
  - By default are considered to be signed integers
  - Unsigned if have “U” as suffix
    - $0U$, $4294967259U$

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

- **Expression Evaluation**
  - If there is a mix of unsigned and signed in single expression, *signed values implicitly cast to unsigned*
  - Including comparison operations `<`, `>`, `==`, `<=`, `>=`

- **Constant 1**
  - **Constant 2**
  - **Relation**
  - **Evaluation**

<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>==</td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>&lt;</td>
<td>signed</td>
</tr>
<tr>
<td>-1</td>
<td>0U</td>
<td>&gt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>-2147483647-1</td>
<td>&gt;</td>
<td>signed</td>
</tr>
<tr>
<td>2147483647U</td>
<td>-2147483647-1</td>
<td>&lt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>-2</td>
<td>&gt;</td>
<td>signed</td>
</tr>
<tr>
<td>(unsigned)-1</td>
<td>-2</td>
<td>&gt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>2147483648U</td>
<td>&lt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>(int) 2147483648U</td>
<td>&gt;</td>
<td>signed</td>
</tr>
</tbody>
</table>
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
/* Kernel memory region holding user-accessible data */
define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
define MSIZE 528

int copy_from_kernel(void *user_dest, int 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);
}
/* 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);
    ...
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

- Making ints from bytes
- Summary
Task:
- Given $w$-bit signed integer $x$
- Convert it to $w+k$-bit integer with same value

Rule:
- Make $k$ copies of sign bit:
- $X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0$
Sign Extension Example

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

```c
short int x = 15213;
int ix = (int) x;
short int y = -15213;
int iy = (int) y;
```

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

- Summary
Unsigned Addition

Operands: \( w \) bits

True Sum: \( w+1 \) bits

Discard Carry: \( w \) bits

- **Standard Addition Function**
  - Ignores carry output
- **Implements Modular Arithmetic**

\[ s = UAdd_w(u, v) = u + v \mod 2^w \]

\[ UAdd_w(u, v) = \begin{cases} 
  u + v & u + v < 2^w \\
  u + v - 2^w & u + v \geq 2^w
\end{cases} \]
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

![Graph](image.png)
Visualizing Unsigned Addition

- Wraps Around
  - If true sum ≥ $2^w$
  - At most once

True Sum
$2^{w+1}$
$2^w$
0

Modular Sum

Overflow

$UAdd_4(u, v)$

Overflow
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 \( \text{UComp}_w(u) = 2^w - u \)
      \[ \text{UAdd}_w(u, \text{UComp}_w(u)) = 0 \]
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

<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}</td>
</tr>
<tr>
<td>0 000...0</td>
<td>0</td>
</tr>
<tr>
<td>1 011...1</td>
<td>-2^{w-1}-1</td>
</tr>
<tr>
<td>1 000...0</td>
<td>-2^w</td>
</tr>
</tbody>
</table>

- \( PosOver \):
  - 011...1
- \( NegOver \):
  - 100...0
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
Characterizing TAdd

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

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

- Computing Exact Product of $w$-bit numbers $x, y$
  - Either signed or unsigned
- Ranges
  - Unsigned: $0 \leq x \times y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
    - Up to $2w$ bits
  - Two’s complement min: $x \times y \geq (-2^{w-1})(2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$
    - Up to $2w-1$ bits
  - Two’s complement max: $x \times y \leq (-2^{w-1})^2 = 2^{2w-2}$
    - Up to $2w$ bits, but only for $(TMin_w)^2$
- Maintaining Exact Results
  - Would need to keep expanding word size with each product computed
  - Done in software by “arbitrary precision” arithmetic packages
Unsigned Multiplication in C

Operands: $w$ bits

True Product: $2w$ bits

Discard $w$ bits: $w$ bits

- Standard Multiplication Function
  - Ignores high order $w$ bits

- Implements Modular Arithmetic

\[ \text{UMult}_w(u, v) = u \cdot v \mod 2^w \]
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);
```

```
malloc(ele_cnt * ele_size)
```

diagram of data flow and memory allocation
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;
}
```
malloc(ele_cnt * ele_size)

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

- How can I make this function secure?
Signed Multiplication in C

Operands: $w$ bits

True Product: $2w$ bits

Discard $w$ bits: $w$ bits

- Standard Multiplication Function
  - Ignores high order $w$ bits
  - Some of which are different for signed vs. unsigned multiplication
  - Lower bits are the same
Power-of-2 Multiply with Shift

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

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

- **Examples**
  - \( u << 3 \) \(\equiv\) \( u \cdot 8 \)
  - \( u << 5 - u << 3 \) \(\equiv\) \( u \cdot 24 \)

  - Most machines shift and add faster than multiply
    - Compiler generates this code automatically
C Function

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

Compiled Arithmetic Operations

- **leal (%eax,%eax,2), %eax**
- **sall $2, %eax**

**Explanation**

- `t <- x+x*2`
- `return t << 2;`

- C compiler automatically generates shift/add code when multiplying by constant
**Unsigned Power-of-2 Divide with Shift**

- Quotient of Unsigned by Power of 2
  - `u >> k` gives \( \lfloor u / 2^k \rfloor \)
  - Uses logical shift

### Division Example

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>15213</td>
<td>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>( x &gt;&gt; 1)</td>
<td>7606.5</td>
<td>1D B6</td>
<td>00011101 10110110</td>
</tr>
<tr>
<td>( x &gt;&gt; 4)</td>
<td>950.8125</td>
<td>03 B6</td>
<td>00000011 10110110</td>
</tr>
<tr>
<td>( x &gt;&gt; 8)</td>
<td>59.4257813</td>
<td>00 3B</td>
<td>00000000 00111011</td>
</tr>
</tbody>
</table>
Compiled Unsigned Division Code

C Function

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

Compiled Arithmetic Operations

```
shrl $3, %eax
```

Explanation

```
# Logical shift
return x >> 3;
```

- Uses logical shift for unsigned
- For Java Users
  - Logical shift written as >>>>
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:

\[
\begin{array}{c}
\text{Operand:} \\
\end{array}
\]

Division:

\[
\begin{array}{c}
x \div 2^k \\
\end{array}
\]

Result: \( \text{RoundDown}(x / 2^k) \)

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>14</td>
<td>C4</td>
<td>10000010 10010011</td>
</tr>
<tr>
<td>( y \gg 1 )</td>
<td>-7606.5</td>
<td>E2</td>
<td>11100010 01001001</td>
</tr>
<tr>
<td>( y \gg 4 )</td>
<td>-950.8125</td>
<td>FC</td>
<td>11111100 01001001</td>
</tr>
<tr>
<td>( y \gg 8 )</td>
<td>-59.4257813</td>
<td>FF</td>
<td>11111111 11000100</td>
</tr>
</tbody>
</table>
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)
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
Today: Integers

- Representing information as bits
- Bit-level manipulations

Integers
- Representation: unsigned and signed
- Conversion, casting
- Expanding, truncating
- Addition, negation, multiplication, shifting

Summary

- Making ints from bytes
- Summary
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.,
    \[
    \begin{align*}
    u &> 0 \quad \Rightarrow \quad u + v > v \\
    u &> 0, \; v &> 0 \quad \Rightarrow \quad u \cdot v > 0
    \end{align*}
    \]
  - These properties are not obeyed by two’s comp. arithmetic
    \[
    \begin{align*}
    T_{Max} + 1 &= T_{Min} \\
    15213 \times 30426 &= -10030
    \end{align*}
    \] (16-bit words)
Why Should I Use Unsigned?

- Don’t Use Just Because Number Nonnegative
  - Easy to make mistakes
    ```c
    unsigned i;
    for (i = cnt-2; i >= 0; i--)
        a[i] += a[i+1];
    ```
  - Can be very subtle
    ```c
    #define DELTA sizeof(int)
    int i;
    for (i = CNT; i-DELTA >= 0; i-= DELTA)
        ...
    ```

- Do Use When Performing Modular Arithmetic
  - Multiprecision arithmetic

- Do Use When Using Bits to Represent Sets
  - Logical right shift, no sign extension
Today: Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
- Making ints from bytes
- Summary
Byte-Oriented Memory Organization

- Programs Refer to Virtual Addresses
  - Conceptually very large array of bytes
  - Actually implemented with hierarchy of different memory types
  - System provides address space private to particular “process”
    - Program being executed
    - Program can clobber its own data, but not that of others

- Compiler + Run-Time System Control Allocation
  - Where different program objects should be stored
  - All allocation within single virtual address space
Machine Words

- **Machine Has “Word Size”**
  - Nominal size of integer-valued data
    - Including addresses
  - Most current machines use 32 bits (4 bytes) words
    - Limits addresses to 4GB
    - Becoming too small for memory-intensive applications
  - High-end systems use 64 bits (8 bytes) words
    - Potential address space $\approx 1.8 \times 10^{19}$ bytes
    - x86-64 machines support 48-bit addresses: 256 Terabytes
  - Machines support multiple data formats
    - Fractions or multiples of word size
    - Always integral number of bytes
Word-Oriented Memory Organization

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

<table>
<thead>
<tr>
<th>32-bit Words</th>
<th>64-bit Words</th>
<th>Bytes</th>
<th>Addr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr = 0000</td>
<td>Addr = 0000</td>
<td></td>
<td>0000</td>
</tr>
<tr>
<td>Addr = 0004</td>
<td></td>
<td></td>
<td>0001</td>
</tr>
<tr>
<td>Addr = 0008</td>
<td></td>
<td></td>
<td>0002</td>
</tr>
<tr>
<td>Addr = 0012</td>
<td></td>
<td></td>
<td>0003</td>
</tr>
<tr>
<td></td>
<td>Addr = 0008</td>
<td></td>
<td>0004</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0005</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0006</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0007</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0008</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0009</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0011</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0012</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0013</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0014</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0015</td>
</tr>
</tbody>
</table>
Byte Ordering

- How should bytes within a multi-byte word be 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

- **Big Endian**
  - Least significant byte has highest address

- **Little Endian**
  - Least significant byte has lowest address

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

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

<table>
<thead>
<tr>
<th>Little Endian</th>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>67</td>
<td>45</td>
<td>23</td>
<td>01</td>
</tr>
</tbody>
</table>
Disassembly
- Text representation of binary machine code
- Generated by program that reads the machine code

Example Fragment

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

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

- Code to Print Byte Representation of Data
  - Casting pointer to unsigned char * creates byte array

```c
typedef unsigned char *pointer;

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

**Printf directives:**
- `%p`: Print pointer
- `%x`: Print Hexadecimal
int a = 15213;
printf("int a = 15213;\n");
show_bytes((pointer) &a, sizeof(int));

Result (Linux):

int a = 15213;
0x11ffffffcb8 0x6d
0x11ffffffcb9 0x3b
0x11ffffffcba 0x00
0x11ffffffcbb 0x00
Data alignment

- A memory address $a$, is said to be $n$-byte aligned when $a$ is a multiple of $n$ bytes.
  - $n$ is a power of two in all interesting cases
  - Every byte address is aligned
  - A 4-byte quantity is aligned at addresses 0, 4, 8,…

- Some architectures require alignment (e.g., MIPS)
- Some architectures tolerate misalignment at performance penalty (e.g., x86)
Data alignment in C structs

- Struct members are never reordered in C & C++
- Compiler adds padding so each member is aligned
  - `struct {char a; char b;}` no padding
  - `struct {char a; short b;}` one byte pad after a
- Last member is padded so the total size of the structure is a multiple of the largest alignment of any structure member (so struct can go in array)
  - `struct containing int requires 4-byte alignment`
  - `struct containing long requires 8-byte (on 64-bit arch)`
Data alignment malloc

- `malloc(1)`
  - 16-byte aligned results on 32-bit
  - 32-byte aligned results on 64-bit

- `int posix_memalign(void **memptr, size_t alignment, size_t size);`
  - Allocates size bytes
  - Places the address of the allocated memory in *memptr
  - Address will be a multiple of alignment, which must be a power of two and a multiple of `sizeof(void *)`
Representing Integers

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

int A = 15213;

long int C = 15213;

int B = -15213;

Two’s complement representation (Covered later)
Representing Pointers

Different compilers & machines assign different locations to objects

```c
int B = -15213;
int *P = &B;
```
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
Integer C Puzzles

Initialization

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

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