Bits, Bytes, and Integers

15-213: Introduction to Computer Systems
2\textsuperscript{nd} and 3\textsuperscript{rd} Lectures, Jan 19 and Jan 24, 2012

\textbf{Instructors:}
Todd C. Mowry & Anthony Rowe
Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
- Representations in memory, pointers, strings
Binary Representations

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

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

![Diagram showing voltage levels and binary transitions](image-url)
Encoding Byte Values

- **Byte = 8 bits**
  - Binary: 00000000₂ to 11111111₂
  - Decimal: 0₁₀ to 255₁₀
  - Hexadecimal: 00₁₆ to FF₁₆
    - Base 16 number representation
    - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
    - Write FA1D37B₁₆ in C as
      - 0xFA1D37B
      - 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
- A&B = 1 when both A=1 and B=1

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

Or
- A|B = 1 when either A=1 or B=1

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

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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></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)
- \( A^B = 1 \) when either A=1 or B=1, but not both

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

- **Operate on Bit Vectors**
  - Operations applied bitwise

  \[
  \begin{array}{c}
  01101001 \\
  & \text{&} \ 01010101 \\
  \hline
  01000001
  \end{array}
  \quad
  \begin{array}{c}
  01101001 \\
  \text{|} \ 01010101 \\
  \hline
  01111101
  \end{array}
  \quad
  \begin{array}{c}
  01101001 \\
  ^ \ 01010101 \\
  \hline
  00111100
  \end{array}
  \quad
  \begin{array}{c}
  01010101 \\
  \sim \ 01010101 \\
  \hline
  10101010
  \end{array}
  \]

- **All of the Properties of Boolean Algebra Apply**
Example: Representing & Manipulating Sets

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

- $\begin{array}{c}
01101001 & \{0, 3, 5, 6\} \\
76543210 & \end{array}$

- $\begin{array}{c}
01010101 & \{0, 2, 4, 6\} \\
76543210 & \end{array}$

**Operations**
- $\&$ Intersection $01000001 \ {0, 6}$
- $|$ Union $01111101 \ {0, 2, 3, 4, 5, 6}$
- $^\wedge$ Symmetric difference $00111100 \ {2, 3, 4, 5}$
- $\sim$ Complement $10101010 \ {1, 3, 5, 7}$
Bit-Level Operations in C

- Operations &, |, ~, ^ Available in C
  - Apply to any “integral” data type
    - long, int, short, char, unsigned
  - View arguments as bit vectors
  - Arguments applied bit-wise

- Examples (Char data type)
  - ~0x41 → 0xBE
    - ~01000011₂ → 10111100₂
  - ~0x00 → 0xFF
    - ~00000000₂ → 11111111₂
  - 0x69 & 0x55 → 0x41
    - 01101001₂ & 01010101₂ → 01000001₂
  - 0x69 | 0x55 → 0x7D
    - 01101001₂ | 01010101₂ → 01111101₂
Contrast: Logic Operations in C

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

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

- Contrast to Logical Operators
  - `&&`, `||`, `!`
    - View 0 as “False”
    - Anything nonzero as “True”
    - Always return 0 or 1
    - Early termination
  
- Examples (char data type)
  - `!0x41` → 0x00
  - `!0x00` → 0x01
  - `!!0x41` → 0x01
  - `0x69 && 0x55` → 0x01
  - `0x69 || 0x55` → 0x01
  - `p && *p` (avoids null pointer access)

Watch out for `&& vs. &` (and `|| vs. |`)… one of the more common oopsies in C programming!
Shift Operations

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

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

- **Undefined Behavior**
  - Shift amount < 0 or ≥ word size
Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
- Representations in memory, pointers, strings
- Summary
Encoding Integers

Unsigned

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

Two’s Complement

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

- **C** short 2 bytes long

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

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

\[
x = \quad 15213: \quad 00111011 \quad 01101101
\]
\[
y = \quad -15213: \quad 11000100 \quad 10010011
\]

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

Sum \quad 15213 \quad -15213
Numeric Ranges

- **Unsigned Values**
  - $UMin = 0$
    - 000...0
  - $UMax = 2^w - 1$
    - 111...1

- **Two’s Complement Values**
  - $TMin = -2^{w-1}$
    - 100...0
  - $TMax = 2^{w-1} - 1$
    - 011...1

- **Other Values**
  - Minus 1
    - 111...1

Values for $W = 16$

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>$UMax$</td>
<td>65535</td>
<td>FF FF</td>
<td>11111111 11111111</td>
<td></td>
</tr>
<tr>
<td>$TMax$</td>
<td>32767</td>
<td>7F FF</td>
<td>01111111 11111111</td>
<td></td>
</tr>
<tr>
<td>$TMin$</td>
<td>-32768</td>
<td>80 00</td>
<td>10000000 00000000</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>FF FF</td>
<td>11111111 11111111</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00</td>
<td>00000000 00000000</td>
<td></td>
</tr>
</tbody>
</table>
Values for Different Word Sizes

<table>
<thead>
<tr>
<th></th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
</tr>
<tr>
<td>UMax</td>
<td>255</td>
</tr>
<tr>
<td>Tmax</td>
<td>127</td>
</tr>
<tr>
<td>Tmin</td>
<td>-128</td>
</tr>
</tbody>
</table>

**Observations**
- $|T_{Min}| = T_{Max} + 1$
  - Asymmetric range
- $UMax = 2 * T_{Max} + 1$

**C Programming**
- `#include <limits.h>`
- Declares constants, e.g.,
  - `ULONG_MAX`
  - `LONG_MAX`
  - `LONG_MIN`
- Values platform specific
# 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>$\chi$</th>
<th>$B2U(\chi)$</th>
<th>$B2T(\chi)$</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>
Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations

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

Representations in memory, pointers, strings
Mapping Between Signed & Unsigned

Two’s Complement → Unsigned

Unsigned → Two’s Complement

- 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>

T2U and U2T are used to convert between signed and unsigned representations.
## 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>

- Signed values range from 0 to 7, inclusive.
- Unsigned values range from 0 to 15, inclusive.
- The mapping is indicated by an arrow pointing both ways, signifying a two's complement representation for the signed values.
Relation between Signed & Unsigned

Two’s Complement

Maintain Same Bit Pattern

Unsigned

Large negative weight

Large positive weight
Conversion Visualized

- 2’s Comp. $\rightarrow$ Unsigned
  - Ordering Inversion
  - Negative $\rightarrow$ Big Positive

2’s Complement Range

Unsigned Range

0

$T_{Max}$

$T_{Min}$

$U_{Max}$

$U_{Max} - 1$

$T_{Max} + 1$

$T_{Max}$
Signed vs. Unsigned in C

- **Constants**
  - By default are considered to be signed integers
  - Unsigned if have “U” as suffix
    - \(0\text{U}, 4294967259\text{U}\)

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

- **Expression Evaluation**
  - If there is a mix of unsigned and signed in single expression, *signed values implicitly cast to unsigned*
  - Including comparison operations $<, >, ==, <=, >=$
  - Examples for $W = 32$: $\text{TMIN} = -2,147,483,648$, $\text{TMAX} = 2,147,483,647$

<table>
<thead>
<tr>
<th>Constant$_1$</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>2,147,483,647</td>
<td>-2,147,483,647-1</td>
<td>$&gt;$</td>
<td>signed</td>
</tr>
<tr>
<td>2,147,483,647U</td>
<td>-2,147,483,647-1</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>2,147,483,647</td>
<td>2,147,483,648U</td>
<td>$&lt;$</td>
<td>unsigned</td>
</tr>
<tr>
<td>2,147,483,647</td>
<td>(int) 2,147,483,648U</td>
<td>$&gt;$</td>
<td>signed</td>
</tr>
</tbody>
</table>
Summary
Casting Signed $\leftrightarrow$ Unsigned: Basic Rules

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

- Expression containing signed and unsigned int
  - int is cast to unsigned!!
Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations

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

- Representations in memory, pointers, strings
Sign Extension

- **Task:**
  - Given $w$-bit signed integer $x$
  - Convert it to $w+k$-bit integer with same value

- **Rule:**
  - Make $k$ copies of sign bit:
  - $X' = x_{w-1}, ..., x_{w-1}, x_{w-1}, x_{w-2}, ..., x_0$

---

**Diagram:**

- $X$ to $X'$
- $k$ copies of MSB
- $w$ bits
- $k$ bits
Sign Extension Example

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

<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</td>
</tr>
<tr>
<td>ix</td>
<td>15213</td>
<td>00 00 3B 6D</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>C4 93</td>
</tr>
<tr>
<td>iy</td>
<td>-15213</td>
<td>FF FF C4 93</td>
</tr>
</tbody>
</table>

- Converting from smaller to larger integer data type
- C automatically performs sign extension
Summary: Expanding, Truncating: Basic Rules

- Expanding (e.g., short int to int)
  - Unsigned: zeros added
  - Signed: sign extension
  - Both yield expected result

- Truncating (e.g., unsigned to unsigned short)
  - Unsigned/signed: bits are truncated
  - Result reinterpreted
  - Unsigned: mod operation
  - Signed: similar to mod
  - For small numbers yields expected behaviour
Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - *Addition, negation, multiplication, shifting*
- Representations in memory, pointers, strings
- Summary
Unsigned Addition

Operands: $w$ bits

True Sum: $w+1$ bits

Discard Carry: $w$ bits

- Standard Addition Function
  - Ignores carry output

- Implements Modular Arithmetic
  \[ s = \text{UAdd}_w(u, v) = u + v \mod 2^w \]
Visualizing (Mathematical) Integer Addition

- Integer Addition
  - 4-bit integers $u, v$
  - Compute true sum $\text{Add}_4(u, v)$
  - Values increase linearly with $u$ and $v$
  - Forms planar surface
Visualizing Unsigned Addition

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

True Sum

$\begin{array}{c}
2^{w+1} \\
2^w \\
\hline \\
0
\end{array}$

Modular Sum

Overflow

Overflow

$UAdd_4(u, v)$
Two’s Complement Addition

Operands: w bits

True Sum: w+1 bits

Discard Carry: w bits

**TAdd and UAdd have Identical Bit-Level Behavior**

- Signed vs. unsigned addition in C:
  ```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>011...1</td>
</tr>
<tr>
<td>0 100...0</td>
<td>000...0</td>
</tr>
<tr>
<td>0 000...0</td>
<td></td>
</tr>
<tr>
<td>1 011...1</td>
<td>100...0</td>
</tr>
<tr>
<td>1 000...0</td>
<td></td>
</tr>
</tbody>
</table>
Visualizing 2’s Complement Addition

- **Values**
  - 4-bit two’s comp.
  - Range from -8 to +7

- **Wraps Around**
  - If sum $\geq 2^{w-1}$
    - Becomes negative
    - At most once
  - If sum $< -2^{w-1}$
    - Becomes positive
    - At most once
Multiplication

- **Goal:** Computing Product of \( w \)-bit numbers \( x, y \)
  - Either signed or unsigned

- **But, exact results can be bigger than \( w \) bits**
  - Unsigned: up to \( 2w \) bits
    - Result range: \( 0 \leq x \times y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1 \)
  - Two’s complement min (negative): Up to \( 2w-1 \) bits
    - Result range: \( x \times y \geq (-2^{w-1}) \times (2^{w-1}-1) = -2^{2w-2} + 2^{w-1} \)
  - Two’s complement max (positive): Up to \( 2w \) bits, but only for \((TMin_w)^2\)
    - Result range: \( x \times y \leq (-2^{w-1})^2 = 2^{2w-2} \)

- **So, maintaining exact results...**
  - would need to keep expanding word size with each product computed
  - is done in software, if needed
    - e.g., by “arbitrary precision” arithmetic packages
Unsigned Multiplication in C

Operands: $w$ bits

True Product: $2 \cdot w$ bits

Discard $w$ bits: $w$ bits

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

- **Implements Modular Arithmetic**
  
  $\text{UMult}_w(u, v) = u \cdot v \mod 2^w$
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

- $u << 3 == u \cdot 8$
- $u << 5 - u << 3 == u \cdot 24$
- 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 \( \lfloor u / 2^k \rfloor \)
  - Uses logical shift

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>15213</td>
<td>15213</td>
<td>3B 6D 00111011 01101101</td>
</tr>
<tr>
<td>x &gt;&gt; 1</td>
<td>7606.5</td>
<td>7606</td>
<td>1D B6 00011101 10110110</td>
</tr>
<tr>
<td>x &gt;&gt; 4</td>
<td>950.8125</td>
<td>950</td>
<td>03 B6 00000011 10110110</td>
</tr>
<tr>
<td>x &gt;&gt; 8</td>
<td>59.4257813</td>
<td>59</td>
<td>00 3B 00000000 00111011</td>
</tr>
</tbody>
</table>
Signed Power-of-2 Divide with Shift

- **Quotient of Signed by Power of 2**
  - \( x >> k \) gives \( \lfloor \frac{x}{2^k} \rfloor \)
  - Uses arithmetic shift
  - Rounds wrong direction when \( u < 0 \)

Operands:

\[
\begin{array}{l}
\text{x} \quad \text{\( k \)} \\
\hline
\end{array}
\]

Division:

\[
\text{\( \frac{x}{2^k} \)}
\]

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

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y )</td>
<td>-15213</td>
<td>C4 93</td>
<td>11000100 10010011</td>
</tr>
<tr>
<td>( y &gt;&gt; 1 )</td>
<td>-7606.5</td>
<td>E2 49</td>
<td>11100010 01001001</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 \( \left\lfloor \frac{x}{2^k} \right\rfloor \) (Round Toward 0)
  - Compute as \( \left\lfloor \frac{x+2^{k-1}}{2^k} \right\rfloor \)
    - In C: \( (x + (1\ll k) - 1) >> k \)
    - Biases dividend toward 0

**Case 1: No rounding**

**Dividend:**

\[
\begin{array}{c}
\text{u} \\
\text{+2^k-1}
\end{array}
\]

**Divisor:**

\[
\begin{array}{c}
\text{1} \\
\text{2^k}
\end{array}
\]

**Binary Point**

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

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

Case 2: Rounding

Dividend: \( x \)

\[ +2^k - 1 \]

Divisor: \( \frac{x}{2^k} \)

 Incremented by 1

Biasing adds 1 to final result
Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
- Representations in memory, pointers, strings
**Arithmetic: Basic Rules**

- **Addition:**
  - Unsigned/signed: Normal addition followed by truncate, same operation on bit level
  - Unsigned: addition mod $2^w$
    - Mathematical addition + possible subtraction of $2^w$
  - Signed: modified addition mod $2^w$ (result in proper range)
    - Mathematical addition + possible addition or subtraction of $2^w$

- **Multiplication:**
  - Unsigned/signed: Normal multiplication followed by truncate, same operation on bit level
  - Unsigned: multiplication mod $2^w$
  - Signed: modified multiplication mod $2^w$ (result in proper range)
Why Should I Use Unsigned?

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

- **Do Use When Performing Modular Arithmetic**
  - Multiprecision arithmetic

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

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
- Representations in memory, pointers, strings
Byte-Oriented Memory Organization

- Programs refer to data by address
  - Conceptually, envision it as a very large array of bytes
    - In reality, it’s not, but can think of it that way
  - An address is like an index into that array
    - and, a pointer variable stores an address

- Note: system provides private address spaces to each “process”
  - Think of a process as a program being executed
  - So, a program can clobber its own data, but not that of others
Machine Words

- Any given computer has a “Word Size”
  - Nominal size of integer-valued data
    - and of addresses
  - Most current machines use 32 bits (4 bytes) as word size
    - Limits addresses to 4GB ($2^{32}$ bytes)
    - Becoming too small for memory-intensive applications
      - leading to emergence of computers with 64-bit word size
  - Machines still support multiple data formats
    - Fractions or multiples of word size
    - Always integral number of bytes
Word-Oriented Memory Organization

- **Addresses Specify Byte Locations**
  - Address of first byte in word
  - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)
For other data representations too ...

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Typical 32-bit</th>
<th>Intel IA32</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>long long</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>8</td>
<td>10/12</td>
<td>10/16</td>
</tr>
<tr>
<td>pointer</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>
Byte Ordering

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

- Conventions
  - Big Endian: Sun, PPC Mac, Internet
    - Least significant byte has highest address
  - Little Endian: x86
    - Least significant byte has lowest address
Byte Ordering Example

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

Big Endian

```
0x100 0x101 0x102 0x103
```

```
01 23 45 67
```

Little Endian

```
0x100 0x101 0x102 0x103
```

```
67 45 23 01
```
Representing Integers

\[
\text{Decimal: 15213} \\
\text{Binary: 0011 1011 0110 1101} \\
\text{Hex: 3B 6D}
\]

\[
\begin{align*}
\text{int A} & = 15213; \\
\text{long int C} & = 15213;
\end{align*}
\]

\[
\begin{align*}
\text{int B} & = -15213;
\end{align*}
\]
Examining Data Representations

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

```c
typedef unsigned char *pointer;

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

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

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

Result (Linux):

```c
int a = 15213;
0x11fffffffcb8 0x6d
0x11fffffffcb9 0x3b
0x11fffffffcba 0x00
0x11fffffffcb9 0x00
0x11fffffffcb9 0x00
```
Representing Pointers

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

Different compilers & machines assign different locations to objects
Representing Strings

- **Strings in C**
  - Represented by array of characters
  - Each character encoded in ASCII format
    - Standard 7-bit encoding of character set
    - Character “0” has code 0x30
    - Digit $i$ has code 0x30+$i$
  - String should be null-terminated
    - Final character = 0

- **Compatibility**
  - Byte ordering not an issue

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

Initialization

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

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