Integer Arithmetic
Signed vs. unsigned in C

• int x; // defines x as a **signed** integer
• unsigned int x; // defines x as an unsigned integer

• Constants are signed by default
  • 42 // would be stored as a two’s complement number
  • 42u // would be stored as an unsigned binary number

• C allows conversion between unsigned and signed
  • Most systems follow the rule that the underlying bit representation does not change

```c
int main(void) {
    unsigned char ux = 0xff;
    char x = ux;
    printf("%d %d\n", ux, x);
}
```

*What will this code print?*  
255 and -1 !!!
### 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 shows a one-to-one correspondence between signed and unsigned integers, with the addition of a range for negative values in the signed representation.

The diagram illustrates the mapping with arrows indicating the equivalence and the range of values for each representation.
Relation between Signed & Unsigned

Two’s Complement

\[ \xrightarrow{T2U} \]

Unsigned

Maintain Same Bit Pattern

\[ \xrightarrow{T2B} \]

\[ \xrightarrow{B2U} \]

Large negative weight \textit{becomes} Large positive weight

Large negative weight

Large positive weight

\[ \begin{array}{c}
  \text{ux} \\
  \hline
  + + + \cdots + + + \\
  \hline
  x \\
  - + + \cdots + + + \\
\end{array} \]
Expanding the bit representation of a number

- How do you convert from a smaller integer data type to a larger integer data type (e.g., a `short` to an `int`) while retaining the same value?
  - Should always be possible because the larger type will have a wider range of numbers than the smaller type

- To convert an unsigned number to a larger data type:
  - Add leading zeros to the representation
  - Known as zero extension

- To convert a two’s complement number to a larger data type:
  - Perform sign extension
  - Add copies of the msb (the sign bit) of the smaller representation to the extra bits of the larger representation
  - C automatically performs sign extension

Example:
- Extend a 5-bit number to an 8-bit

  **UNSIGNED**
  
  \[
  \begin{array}{c}
  \text{10110} \rightarrow 00010110 \\
  \text{01100} \rightarrow 000001100
  \end{array}
  \]

  **SIGNED**
  
  \[
  \begin{array}{c}
  \text{10110} \rightarrow 11110110 \\
  \text{01100} \rightarrow 00000100
  \end{array}
  \]
Truncating Numbers

• When reducing the number of bits in a number, the smaller number may not be able to correctly represent the larger number

• We reduce the number of bits representing a number by truncating the uppermost (high-order) bits
  • Reinterpret this truncated number

• For unsigned numbers, it is equivalent to performing a mod $2^k$ on the original value where $k$ is the smaller bit width

• A similar property holds for signed numbers except that it then converts the msb into a sign bit
  • Which means it is possible that truncating a signed number changes its sign

• Rule of thumb: if the larger number is within the expressible range of the smaller number, the conversion will yield a correct result
Expanding and Truncating Rules

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

- **Truncating (e.g., int to short)**
  - Unsigned/signed: high order bits are truncated (drop)
  - Result reinterpreted
  - For small numbers this yields expected behavior
    - $1111010 \rightarrow -6$ 8-bit two’s complement
    - $1010 \rightarrow -6$ 4-bit two’s complement
  - Overflow can result in a sign change

9/7/2022

CMPU 224 -- Computer Organization
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>9,223,720,036,854,775,807</td>
</tr>
<tr>
<td>Tmin</td>
<td>-128</td>
<td>-32,768</td>
<td>-2,147,483,648</td>
<td>-9,223,720,036,854,775,808</td>
</tr>
</tbody>
</table>

- Observations
  - $|TMin| = Tmax + 1$
  - Asymmetric range
  - $U_{max} = 2 \times Tmax + 1$

- C Programming
  - `#include <limits.h>`
  - Declares constants, e.g.,
    - `ULONG_MAX`
    - `LONG_MAX`
    - `LONG_MIN`
  - Values are platform specific
Unsigned Addition

Operands: $w$ bits

True Sum: $w+1$ bits

Discard Carry: $w$ bits

- C and many other languages only support fixed-size arithmetic
- Standard Addition Function
  - Ignores carry output
- Implements Modular Arithmetic
  $$s = \text{UAdd}_w(u, v) = u + v \mod 2^w$$

9/7/2022

CMPU 224 -- Computer Organization
Unsigned integer overflow

• With n bits, we can represent values 0, 1, 2, ..., 2^n-1
• Overflow occurs when we have a result that doesn’t fit in the n bits
  • E.g., using 4 bits: 0xF + 0x1

\[
\begin{align*}
0xF &= 1111 \\
0x1 &= 0001 \\
\hline
0xF + 0x1 &= 0x0 \\
\text{Overflow!!}
\end{align*}
\]
Visualizing Unsigned Addition

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

Operands: $w$ bits

True Sum: $w+1$ bits

Discard Carry: $w$ bits

$TAdd_w(u, v)$

- $TAdd$ and $UAdd$ have Identical Bit-Level Behavior
Visualizing 2’s Complement Addition

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

- Wraps Around
  - If sum $\geq 2^{w-1}$: **Positive Overflow**
    - Adding two positive numbers
    - Answer should be positive
    - Becomes negative
  - If sum $< -2^{w-1}$: **Negative Overflow**
    - Adding two negative numbers
    - Answer should be negative
    - Becomes positive
Multiplication

- Goal: Computing Product of $w$-bit numbers $x$, $y$
  - Either signed or unsigned
- But exact results can be bigger than $w$ bits
  - Unsigned: up to $2^w$ bits
  - Two’s complement min (negative): Up to $2^w-1$ bits
  - Two’s complement max (positive): Up to $2^w$ bits, but only for $(TMin_w)^2$
- So, maintaining exact results...
  - Would need to keep expanding word size with each product computed
  - Can be done in software, if needed
    - e.g., by “arbitrary precision” arithmetic packages
Unsigned Multiplication in C

- Standard Multiplication Function
  - Ignores high order \( w \) bits

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

Operands: \( w \) bits

True Product: \( 2^w \) bits

Discard \( w \) bits: \( w \) bits
Signed Multiplication in C

- Standard Multiplication Function
  - Ignores high order $w$ bits
  - Some of which are different for signed vs. unsigned multiplication
  - Lower bits are the same
Multiplying by Constants

• Historically, the integer multiply instruction was very slow
  • Even on modern hardware it takes 3 cycles
• The compiler tries to replace multiplication by a constant with shifts and adds
• Multiplication by a power of two
  • A single left shift doubles a number
  
  \[(5) \ 00101 \ \ll \ 1 \ \Rightarrow \ 01010 \ \ (10)\]
• In general
  • \[x \ll k = x \cdot 2^k\]
  • Multiplication by a power of two can be performed by left shift
  • For both signed and unsigned numbers
Multiplying by a Constant Examples

- Examples
  - \( u * 8 == u << 3 \)
  - \( u * 5 == (u << 2) + u \)
  - \( u * 24 == (u << 5) - (u << 3) \)
Unsigned Power-of-2 Divide with Shift

- Quotient of Unsigned by Power of 2
  - $u \gg k$ gives $\lfloor u / 2^k \rfloor$ (greatest integer less than)
  - Uses logical right shift
  - Rounds towards zero (truncates decimal part)

<table>
<thead>
<tr>
<th></th>
<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</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>$x \gg 1$</td>
<td>7606.5</td>
<td>7606</td>
<td>1D B6</td>
<td>00011101 10110110</td>
</tr>
<tr>
<td>$x \gg 4$</td>
<td>950.8125</td>
<td>950</td>
<td>03 B6</td>
<td>00000011 10110110</td>
</tr>
<tr>
<td>$x \gg 8$</td>
<td>59.4257813</td>
<td>59</td>
<td>00 3B</td>
<td>00000000 00111011</td>
</tr>
</tbody>
</table>
Signed Power-of-2 Divide with Shift

- Quotient of signed by Power of 2
  - $u \gg k$ gives $\lfloor u / 2^k \rfloor$
- Uses arithmetic right shift
- Rounds down

<table>
<thead>
<tr>
<th></th>
<th>Division</th>
<th>Computed</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>-12.340</td>
<td>-12,340</td>
<td>11001111 11001100</td>
</tr>
<tr>
<td>$x \gg 1$</td>
<td>-6.170</td>
<td>-6,170</td>
<td>11100111 11100110</td>
</tr>
<tr>
<td>$x \gg 4$</td>
<td>-771.25</td>
<td>-772</td>
<td>11111100 11111100</td>
</tr>
<tr>
<td>$x \gg 8$</td>
<td>-48.203125</td>
<td>-49</td>
<td>11111111 11001111</td>
</tr>
<tr>
<td>$X \gg 14$</td>
<td>-0.75317382</td>
<td>-1</td>
<td>11111111 11111111</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)

• Shifting:
  • Multiplying/Dividing by powers of 2
  • Logical right shift: shift in 0
  • Arithmetic right shift: shift in the sign bit
What is going on with this cartoon?

They are counting sheep using a 16-bit two’s complement number!