Fixed-width integer encodings

**Unsigned** $\subseteq \mathbb{N}$ non-negative integers only

**Signed** $\subseteq \mathbb{Z}$ both negative and non-negative integers

$n$ bits offer only $2^n$ distinct values.

Terminology:

“Most-significant” bit(s) or “high-order” bit(s)

“Least-significant” bit(s) or “low-order” bit(s)

$$\text{Integer Representation}$$

(4-bit) **unsigned integer representation**

<table>
<thead>
<tr>
<th>$1$</th>
<th>$0$</th>
<th>$1$</th>
<th>$1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$8$</td>
<td>$4$</td>
<td>$2$</td>
<td>$1$</td>
</tr>
</tbody>
</table>

$2^3$ $2^2$ $2^1$ $2^0$

$n$-bit unsigned integers:

minimum =

maximum =

$$\text{modular arithmetic, overflow}$$

$x + y$ in $n$-bit unsigned arithmetic is in math: unsigned overflow =

$$11 + 2 + 0010$$

$$13 + 1101 + 0101$$

11001011010100
sign-magnitude

Most-significant bit (MSB) is sign bit
0 means non-negative
1 means negative
Remaining bits are an unsigned magnitude

8-bit sign-magnitude:
00000000 represents ____
01111111 represents ____
10000101 represents ____
10000000 represents ____

Arithmetic?

Example:
4 - 3 = 4 + (-3)

00000100 + 10000011

4-bit two's complement integers:
minimum =
maximum =

(4-bit) two's complement signed integer representation

\[
\begin{array}{cccc}
1 & 0 & 1 & 1 \\
-2^3 & 2^2 & 2^1 & 2^0 \\
\end{array}
\]

\[= 1 \times -2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0\]

4-bit two's complement integers:
minimum =
maximum =

Two's complement vs. unsigned

What's the difference?

n-bit unsigned numbers:
minimum =
maximum =

n-bit two's complement numbers:
minimum =
maximum =

8-bit representations

0 0 0 0 1 0 0 1 1 1 1 1 1 1 1 1
0 0 1 0 0 1 1 1

Compare to unsigned
4-bit unsigned vs. 4-bit two’s complement

\[ 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \]
\[ 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \]

\[ 11 \leftarrow \text{difference} = \_\_ = 2 \rightarrow -5 \]

Modular Arithmetic

two’s complement addition

\[ \begin{array}{cccc}
2 & 0010 & -2 & 1110 \\
+ 3 & + 0011 & + 3 & + 1101 \\
\end{array} \]

-2 & 1110 & 2 & 0010
+ 3 & + 0011 & -3 & + 1101

Reliability

Ariane 5 Rocket, 1996
Exploded due to cast of 64-bit floating-point number to 16-bit signed number. Overflow.

Boeing 787, 2015
"... a Model 787 airplane ... can lose all alternating current (AC) electrical power ... caused by a software counter internal to the GCUs that will overflow after 248 days of continuous power. We are issuing this AD to prevent loss of all AC electrical power, which could result in loss of control of the airplane."
--FAA, April 2015
A few reasons two’s complement is awesome

Addition, subtraction, hardware

Sign

Negative one

Complement rules

Another derivation

How should we represent 8-bit negatives?

- For all positive integers $x$, we want the representations of $x$ and $-x$ to sum to zero.
- We want to use the standard addition algorithm.

\[
\begin{array}{c}
\text{00000001} & \text{00000010} & \text{00000011} \\
+ & + & + \\
\text{00000000} & \text{00000000} & \text{00000000}
\end{array}
\]

- Find a rule to represent $-x$ where that works...

Convert/cast signed number to larger type.

\[
\begin{array}{c}
\text{0 0 0 0 0 0 1 0} & \text{8-bit 2} \\
\text{__________0 0 0 0 0 0 1 0} & \text{16-bit 2} \\
\text{1 1 1 1 1 1 0 0} & \text{8-bit -4} \\
\text{__________1 1 1 1 1 1 0 0} & \text{16-bit -4}
\end{array}
\]

Rule/name?

unsigned shifting and arithmetic

\[
\begin{array}{c}
\text{unsigned} \\
x = 27; \\
y = x \ll 2; \\
y == 108
\end{array}
\]

\[
\begin{array}{c}
\text{unsigned} \\
x = 237; \\
y = x \gg 2; \\
y == 59
\end{array}
\]
two's complement shifting and arithmetic

\[ x = -101; \]
\[ y = x << 2; \]
\[ y == 108 \]

\[ x = -101; \]
\[ y = x >> 2; \]
\[ y == 108 \]

shift-and-add

Available operations
\[ x << k \] implements \( x \cdot 2^k \)
\[ x + y \]

Implement \( y = x \cdot 2^4 \) using only \( <<, +, \) and integer literals

What does this function compute?

unsigned puzzle(unsigned x, unsigned y) {
    unsigned result = 0;
    for (unsigned i = 0; i < 32; i++){
        if (y & (1 << i)) {
            result = result + (x << i);
        }
    }
    return result;
}
### Modular Arithmetic

#### Multiplication

\[
\begin{array}{c|c}
5 & 0101 \\
\times 4 & \times 0100 \\
--- & --- \\
20 & 00010100 \\
-3 & 1101 \\
-7 & 111011011 \\
\end{array}
\]

#### Modular Arithmetic

\[
\begin{array}{c|c}
5 & 0101 \\
\times 5 & \times 0101 \\
--- & --- \\
25 & 00011001 \\
-2 & 1110 \\
-6 & 11110100 \\
\end{array}
\]

### Casting Integers in C

**Number literals:** 37 is signed, 37U is unsigned

**Integer Casting:** *bits unchanged, just reinterpreted.*

**Explicit casting:**

\[
\begin{align*}
\text{int } tx &= (\text{int}) 73U; \quad \text{// still 73} \\
\text{unsigned uy} &= (\text{unsigned}) -4; \quad \text{// big positive #}
\end{align*}
\]

**Implicit casting:** *Actually does*

\[
\begin{align*}
\text{tx} &= \text{ux}; \quad \text{// tx} = (\text{int}) \text{ux}; \\
\text{uy} &= \text{ty}; \quad \text{// uy} = (\text{unsigned}) \text{ty}; \\
\text{void foo(int z) \{ ... \}} \\
\text{foo(ux); \quad // foo((int)ux);} \\
\text{if (tx < ux) ... \quad // if ((unsigned)tx < ux) ...}
\end{align*}
\]

### More Implicit Casting in C

If you mix unsigned and signed in a single expression, then *signed values are implicitly cast to unsigned.*

<table>
<thead>
<tr>
<th>Argument\textsubscript{1}</th>
<th>Op</th>
<th>Argument\textsubscript{2}</th>
<th>Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>==</td>
<td>0U</td>
<td>unsigned</td>
<td>1</td>
</tr>
<tr>
<td>-1</td>
<td>&lt;</td>
<td>0</td>
<td>signed</td>
<td>1</td>
</tr>
<tr>
<td>-1</td>
<td>&lt;</td>
<td>0U</td>
<td>unsigned</td>
<td>0</td>
</tr>
<tr>
<td>2147483647</td>
<td>&lt;</td>
<td>-2147483647</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2147483647U</td>
<td>&lt;</td>
<td>-2147483647U</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>&lt;</td>
<td>-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(unsigned)-1</td>
<td>&lt;</td>
<td>-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td>&lt;</td>
<td>2147483647</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2147483647U</td>
<td>&lt;</td>
<td>(int)2147483648U</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Note:** $T_{\text{min}} = -2,147,483,648 \quad T_{\text{max}} = 2,147,483,647$

$T_{\text{min}}$ must be written as $-2147483647-1$ (see pg. 77 of CSAPP for details)