Integer Representation

Representation of integers: unsigned and signed
Modular arithmetic and overflow
Sign extension
Shifting and arithmetic
Multiplication
Casting

https://cs.wellesley.edu/~cs240/
Fixed-width integer encodings

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

**Signed** $\subset \mathbb{Z}$ 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)

MSB 0110010110101001 LSB
(4-bit) **unsigned integer representation**

<table>
<thead>
<tr>
<th></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>
<tr>
<td>$2^3$</td>
<td>$2^2$</td>
<td>$2^1$</td>
<td>$2^0$</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

$1011 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$

$n$-bit unsigned integers:

minimum =

maximum =
modular arithmetic, overflow

\[
\begin{array}{c}
11 \\
+ 2 \\
\hline
13
\end{array}
\quad \quad
\begin{array}{c}
1011 \\
+ 0010 \\
\hline
1101
\end{array}
\quad \quad
\begin{array}{c}
1111 \\
+ 0001 \\
\hline
1010
\end{array}
\quad \quad
\begin{array}{c}
0000 \\
0001 \\
0010 \\
0011 \\
0100 \\
0101 \\
0110 \\
0111 \\
1000 \\
1001 \\
1010 \\
1011 \\
1100 \\
1101 \\
1110 \\
1111
\end{array}
\]

\[
x + y \text{ in } n \text{-bit unsigned arithmetic is } \quad \quad \text{in math}
\]

\[
\text{unsigned overflow} = \quad \quad = 
\]

\[
\text{Unsigned addition overflows if and only if}
\]
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 ______

Anything weird here?

Arithmetic?

Example:

4 - 3 != 4 + (-3)

00000100

+10000011

Zero?
(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

### Two’s Complement

<table>
<thead>
<tr>
<th>_ _ _ _ _ _ _ _ _ _ _ _</th>
</tr>
</thead>
<tbody>
<tr>
<td>2(^{n-1}) 2(^{n-2}) ... 2(^2) 2(^1) 2(^0)</td>
</tr>
<tr>
<td>-(2(^{n-1})) 2(^{n-2}) ... 2(^2) 2(^1) 2(^0)</td>
</tr>
</tbody>
</table>

### Unsigned

- `(2\(^n\) values)`

- `- (2\(^{(n-1)}\))`
- `0`
- `2\(^{(n-1)}\) - 1`
- `2^n - 1`

### Two’s Complement Range

- `(2^n values)`
4-bit **unsigned** vs. 4-bit **two’s complement**

\[ \begin{align*}
1 &\times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\
1 &\times -2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0
\end{align*} \]

\[ \text{difference} = \_\_ = 2\_\_ \rightarrow -5 \]
8-bit representations

00001001   10000001

11111111   00100111

n-bit two's complement numbers:

minimum =

maximum =
two’s complement addition

\[
\begin{array}{ccc}
2 & 0010 & -2 & 1110 \\
+3 & 0011 & + -3 & +1101 \\
5 & & -5 \\
\end{array}
\]

\[
\begin{array}{ccc}
-2 & 1110 & 2 & 0010 \\
+3 & 0011 & + -3 & +1101 \\
1 & & -1 \\
\end{array}
\]

Modular Arithmetic
two’s complement overflow

Addition overflows
if and only if
if and only if

\[
\begin{array}{c}
-1 \\
+2 \\
6 \\
+3 \\
\end{array}
\begin{array}{c}
1111 \\
0010 \\
0110 \\
0011 \\
\end{array}
\]

Modular Arithmetic

Some CPUs/languages raise exceptions on overflow. C and Java cruise along silently... Feature? Oops?
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

Arithmetic 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.

\[
\begin{array}{ccc}
11111111 & 1111111 & 11111111 \\
00000001 & 00000010 & 00000011 \\
+11111111 & +11111110 & +11111101 \\
00000000 & 00000000 & 00000000 \\
\end{array}
\]

- Find a rule to represent $-x$ where that works...
Convert/cast signed number to larger type.

0 0 0 0 0 0 1 0 8-bit 2

_ _ _ _ _ _ _ _ 0 0 0 0 0 0 1 0 16-bit 2

1 1 1 1 1 1 0 0 8-bit -4

_ _ _ _ _ _ _ _ 1 1 1 1 1 1 0 0 16-bit -4

Rule/name?
**Sign extension** for two's complement

Casting from smaller to larger signed type does sign extension.
unsigned shifting and arithmetic

unsigned
x = 27;
y = x << 2;
y == 108

logical shift left

n = shift distance in bits, w = width of encoding in bits

logical shift right

unsigned
x = 237;
y = x >> 2;
y == 59
two's complement **shifting** and **arithmetic**

**signed**

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

\[ x = -19; \]
\[ y = x \gg 2; \]
\[ y == -5 \]

\[ n = \text{shift distance in bits}, \ w = \text{width of encoding in bits} \]
**shift-and-add**

Available operations

- \( x << k \) implements \( x \times 2^k \)
- \( x + y \)

Implement \( y = x \times 24 \) using only \( << \), \( + \), and integer literals

Parenthesize shifts to be clear about precedence, which may not always be what you expect.
What does this function compute?

```c
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;
}
```

See Bits assignment prep exercise.
What does this function compute?

Downsize to fake unsigned nybble type (4 bits) to make this easier to write...

```c
nybble puzzle(nybble x, nybble y) {
    nybble result = 0;
    for (nybble i = 0; i < 4; i++){
        if (y & (1 << i)) {
            result = result + (x << i);
        }
    }
    return result;
}
```

See Bits assignment prep exercise.
multiplication

\[
\begin{array}{c|c}
2 & 0010 \\
\times 3 & \times 0011 \\
6 & 00000110 \\
\hline
-2 & 1110 \\
\times 2 & \times 0010 \\
-4 & 11111100 \\
\end{array}
\]

Modular Arithmetic
multiplication

\[
\begin{array}{c}
5 \\
\times 4 \\
\hline
20 \\
\hline
4
\end{array}
\] 0101

\[
\begin{array}{c}
\frac{-3}{x 7} \\
\hline
-21 \\
\hline
-5
\end{array}
\] 1101

\[
\begin{array}{c}
\frac{0}{x 0111} \\
\hline
\end{array}
\] 11101011

Modular Arithmetic
multiplication

\[
\begin{array}{ccc}
5 & & 0101 \\
x 5 & & \underline{x 0101} \\
25 & & \underline{00011001} \\
\hline
-7 & & \\
-2 & & 1110 \\
x 6 & & \underline{x 0110} \\
-12 & & \underline{11110100} \\
\hline
4 & & \\
\end{array}
\]
Casting Integers in C

Number literals: \texttt{37} is signed, \texttt{37U} is unsigned

Integer Casting: \textit{bits unchanged, just reinterpreted.}

\textbf{Explicit casting:}

\begin{verbatim}
int tx = (int) 73U; // still 73
unsigned uy = (unsigned) -4; // big positive #
\end{verbatim}

\textbf{Implicit casting: Actually does}

\begin{verbatim}
int tx = (int) 73U; // tx = (int)ux;
unsigned uy = (unsigned) -4; // uy = (unsigned)ty;
void foo(int z) { ... }
foo(ux); // foo((int)ux);
if (tx < ux) ... // if ((unsigned)tx < ux) ...
\end{verbatim}
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&lt;sub&gt;1&lt;/sub&gt;</th>
<th>Op</th>
<th>Argument&lt;sub&gt;2&lt;/sub&gt;</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>1</td>
</tr>
<tr>
<td>2147483647</td>
<td>&lt;</td>
<td>-2147483647–1</td>
<td>unsigned</td>
<td>0</td>
</tr>
<tr>
<td>2147483647U</td>
<td>&lt;</td>
<td>-2147483647–1</td>
<td>unsigned</td>
<td>0</td>
</tr>
<tr>
<td>-1</td>
<td>&lt;</td>
<td>-2</td>
<td>unsigned</td>
<td>0</td>
</tr>
<tr>
<td>(unsigned)-1</td>
<td>&lt;</td>
<td>-2</td>
<td>unsigned</td>
<td>0</td>
</tr>
<tr>
<td>2147483647</td>
<td>&lt;</td>
<td>2147483648U</td>
<td>unsigned</td>
<td>0</td>
</tr>
<tr>
<td>2147483647</td>
<td>&lt;</td>
<td>(int)2147483648U</td>
<td>unsigned</td>
<td>0</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)