Integer Representation

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

MSB

0110010110101001

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

LSB
(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>

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

$n$-bit unsigned integers:

**minimum** =

**maximum** =
modular arithmetic, overflow

\[
\begin{array}{cccc}
11 & 1011 & + & 2 & + & 0010 \\
\hline
13 & 1101
\end{array}
\]

\[
\begin{array}{cccc}
13 & 1101 & + & 5 & + & 0101 \\
\hline
14 & 1101
\end{array}
\]

\[x + y\] in \(n\)-bit unsigned arithmetic is

\[\text{unsigned overflow} = \]

in math

\[\text{unsigned overflow} = \]

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

\[
\begin{array}{c}
00000100 \\
+10000011
\end{array}
\]

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

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

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

4-bit two's complement integers:

minimum =

maximum =
## two’s complement vs. unsigned

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>⋯</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(2^{n-1})</td>
<td>(2^{n-2})</td>
<td>(\ldots)</td>
<td>(2^2)</td>
<td>(2^1)</td>
</tr>
<tr>
<td>(-2^{n-1})</td>
<td>(2^{n-2})</td>
<td>(\ldots)</td>
<td>(2^2)</td>
<td>(2^1)</td>
</tr>
</tbody>
</table>

What's the difference?

### n-bit unsigned numbers:

- **minimum** =
- **maximum** =
8-bit representations

\[ \begin{align*}
0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 & 1 & 1 & 1
\end{align*} \]

n-bit two's complement numbers:

minimum = 00000000
maximum = 11111111
4-bit unsigned vs. 4-bit two’s complement

1 0 1 1

1 x 2^3 + 0 x 2^2 + 1 x 2^1 + 1 x 2^0

1 x -2^3 + 0 x 2^2 + 1 x 2^1 + 1 x 2^0

difference = ___ = 2—

\[
\begin{align*}
1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 &= 1 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1 \\
&= 8 + 0 + 2 + 1 \\
&= 11
\end{align*}
\]

\[
\begin{align*}
1 \times -2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 &= 1 \times (-8) + 0 \times 4 + 1 \times 2 + 1 \times 1 \\
&= -8 + 0 + 2 + 1 \\
&= -5
\end{align*}
\]
two’s complement addition

\[
\begin{array}{cccc}
2 & 0010 & -2 & 1110 \\
+3 & +0011 & +(-3) & +1101 \\
\hline
-2 & 1110 & 2 & 0010 \\
+3 & +0011 & +(-3) & +1101 \\
\end{array}
\]
two's complement over flows

Addition over flows
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

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.

\[
\begin{array}{c}
00000001 \\
+ \quad 00000010 \\
\hline
00000000 \\
\end{array}
\quad \begin{array}{c}
00000010 \\
+ \quad 00000000 \\
\hline
00000000 \\
\end{array}
\quad \begin{array}{c}
00000011 \\
+ \quad 00000000 \\
\hline
00000000 \\
\end{array}
\]

• Find a rule to represent \(-x\) where that works...
Convert/cast signed number to larger type.

\[\begin{array}{ll}
00000010 & 8\text{-bit } 2 \\
\underbrace{00000010} & 16\text{-bit } 2 \\
11111100 & 8\text{-bit } -4 \\
\underbrace{11111100} & 16\text{-bit } -4 \\
\end{array}\]

Rule/name?
unsigned shifting and arithmetic

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

unsigned logical shift left

y = x >> 2;
y == 59

unsigned logical shift right
two's complement **shifting** and **arithmetic**

**signed**

x = -101;
y = x << 2;
y == 108

arithemetic shift right

x = -19;
y = x >> 2;
y == -5

logical shift left
shift-and-add

Available operations

\[ x \ll k \quad \text{implements} \quad x \times 2^k \]
\[ x + y \]

Implement \( y = x \times 24 \) using only \( \ll, +, \) and integer literals
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;
}
```
multiplication

\[
\begin{array}{c}
2 \quad 0010 \\
\times 3 \quad x 0011 \\
6 \\
\hline
6 \quad 00000110 \\
\end{array}
\]

\[
\begin{array}{c}
-2 \quad 1110 \\
\times 2 \quad x 0010 \\
-4 \\
\hline
-4 \quad 11111100 \\
\end{array}
\]

Modular Arithmetic
multiplication

\[
\begin{array}{cccc}
5 & 0101 \\
\times 4 & \times 0100 \\
\hline
20 & 00010100 \\
4 & \\
\hline
-3 & 1101 \\
\times 7 & \times 0111 \\
\hline
-21 & 11101011 \\
-5 &
\end{array}
\]

Modular Arithmetic
multiplication

5
\[ \times 5 \]
25

-7

-2

\[ \times 6 \]
-12

4

0101

00011001

1110

111101000

Modular Arithmetic
Casting Integers in C

Number literals: 37 is signed, 37U is unsigned

Integer Casting: *bits unchanged, just reinterpreted.*

Explicit casting:

\[
\text{int } tx = (\text{int}) 73U; \quad // \text{ still 73}
\]

\[
\text{unsigned } uy = (\text{unsigned}) -4; \quad // \text{ big positive #}
\]

Implicit casting: Actually does

\[
\text{tx} = ux; \quad // \text{ tx} = (\text{int}) ux;
\]

\[
\text{uy} = ty; \quad // \text{ uy} = (\text{unsigned}) ty;
\]

void foo(int z) { ... }

foo(ux); // foo((int)ux);

if (tx < ux) ... // if ((unsigned)tx < ux) ...

27
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>Argument₁</th>
<th>Op</th>
<th>Argument₂</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>−2147483647−1</td>
<td>unsigned</td>
<td>1</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></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>2147483648U</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2147483647</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)