Integer Representation

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

```
MSB
0110010110101001
```

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

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

\[
\begin{array}{cccc}
1 & 0 & 1 & 1 \\
8 & 4 & 2 & 1 \\
2^3 & 2^2 & 2^1 & 2^0 \\
3 & 2 & 1 & 0 \\
\end{array}
\]

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

**n-bit unsigned integers:**

- Minimum =
- Maximum =
modular arithmetic, overflow

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

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

\[\text{unsigned overflow} = \]

in math

UnSigned addition \textit{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**

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

\[-2^3 \quad 2^2 \quad 2^1 \quad 2^0\]

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

4-bit two's complement integers:
- **Minimum** =
- **Maximum** =

*compare to unsigned*
two's complement vs. unsigned

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$2^{n-1}$</td>
<td>$2^{n-2}$</td>
<td>...</td>
<td>$2^1$</td>
<td>$2^0$</td>
</tr>
<tr>
<td>$-2^{n-1}$</td>
<td>$2^{n-2}$</td>
<td>...</td>
<td>$2^2$</td>
<td>$2^1$</td>
<td>$2^0$</td>
</tr>
</tbody>
</table>

What's the difference?

$n$-bit minimum $=$

$n$-bit maximum $=$
8-bit representations

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

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

n-bit two's complement numbers:

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

1 0 1 1

1 x 2³ + 0 x 2² + 1 x 2¹ + 1 x 2⁰

difference = ___ = 2——

1 x -2³ + 0 x 2² + 1 x 2¹ + 1 x 2⁰

--→ -5
two’s complement addition

\[
\begin{array}{cccccc}
2 & 0010 & -2 & 1110 \\
+ 3 & + 0011 & + -3 & + 1101 \\
-2 & 1110 & 2 & 0010 \\
+ 3 & + 0011 & + -3 & + 1101
\end{array}
\]
two’s complement overflow

Addition overflows
if and only if
if and only if

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

Modular Arithmetic

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

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

• 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?
unsigned shifting and arithmetic

unsigned

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y == 108 \]

\[ x = 27; \rightarrow 00011011 \]
\[ y = x \ll 2; \rightarrow 00011011 \rightarrow 0001101100 \]
\[ y == 108 \]

logical shift left

\[ x = 237; \]
\[ y = x \gg 2; \]
\[ y == 59 \]

\[ x = 237; \rightarrow 11101101 \]
\[ y = x \gg 2; \rightarrow 11101101 \rightarrow 0011101101 \]
\[ y == 59 \]
two's complement **shifting** and **arithmetic**

**signed**

\[
x = -101;
\]

\[
y = x << 2;
\]

\[
y == 108
\]

\[
x = -19;
\]

\[
y = x >> 2;
\]

\[
y == -5
\]
**shift-and-add**

**Available operations**

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

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

\[
\begin{array}{crl}
2 & \times 3 & \quad \times 0011 \\
\hline
6 & \quad 00000100 \\
\end{array}
\]

\[
\begin{array}{crl}
-2 & \times 2 & \quad \times 0010 \\
\hline
-4 & \quad 11111100 \\
\end{array}
\]

Modular Arithmetic
multiplication

\[
\begin{array}{c}
5 \\
\times 4 \\
\underline{20}
\end{array}
\quad 0101

\begin{array}{c}
\times 0100 \\
\underline{00010100}
\end{array}

\begin{array}{c}
-3 \\
\times 7 \\
\underline{-21}
\end{array}
\quad 1101

\begin{array}{c}
\times 0111 \\
\underline{11101011}
\end{array}

Modular Arithmetic
multiplication

\[
\begin{array}{c}
5 \\
x \ 5 \\
\hline
25 \\
-7 \\
\hline
-2 \\
x \ 6 \\
\hline
-12 \\
\hline
4
\end{array}
\]

\[
\begin{array}{c}
0101 \\
\times \ 0101 \\
\hline
00011001 \\
-1110 \\
\hline
11110100 \\
\hline
11110100 \\
-01110100 \\
\hline
11110100
\end{array}
\]

Modular Arithmetic
Casting Integers in C

Number literals: 37 is signed, 37U is unsigned

Integer Casting: *bits unchanged, just reinterpreted.*

Explicit casting:

```c
int tx = (int) 73U;       // still 73
unsigned uy = (unsigned) -4;    // big positive #
```

Implicit casting:

Actually does

```c
tx = ux;                  // tx = (int)ux;
uy = ty;                  // uy = (unsigned)ty;
void foo(int z) { ... }  
foo(ux);                  // foo((int)ux);  
if (tx < ux) ...          // if ((unsigned)tx < ux) ...
```
More Implicit Casting in C

If you mix \texttt{unsigned} and \texttt{signed} in a single expression, then \textit{signed values are implicitly cast to unsigned.}

<table>
<thead>
<tr>
<th>Argument(_1)</th>
<th>Op</th>
<th>Argument(_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>-2147483648</td>
<td>unsigned</td>
<td>1</td>
</tr>
<tr>
<td>2147483647U</td>
<td>&lt;</td>
<td>-2147483648</td>
<td>unsigned</td>
<td>0</td>
</tr>
<tr>
<td>-1</td>
<td>&lt;</td>
<td>-2</td>
<td>(int)</td>
<td></td>
</tr>
<tr>
<td>(unsigned)-1</td>
<td>&lt;</td>
<td>-2</td>
<td>(int)</td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td>&lt;</td>
<td>2147483648U</td>
<td>unsigned</td>
<td>1</td>
</tr>
<tr>
<td>2147483647</td>
<td>&lt;</td>
<td>(int)2147483648U</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: \( T_{min} = -2,147,483,648 \quad T_{max} = 2,147,483,647 \)