Integer Representation

Representation of integers: unsigned and signed
Modular arithmetic and overflow
Sign extension
Shifting and arithmetic
Multiplication
Casting
Fixed-width integer encodings

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

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

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

Terminology:

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

```
0110010110101001
```

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

MSB

LSB
Unsigned integer representation

Example in 4-bit unsigned 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>

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

n-bit unsigned numbers:

minimum =

maximum =
Unsigned **modular arithmetic, overflow**

Examples in 4-bit unsigned representation.

\[
\begin{align*}
x + y \text{ in N-bit unsigned arithmetic is } (x + y) \mod 2^N \text{ in math} \\
\text{unsigned overflow} = \"wrong\" \text{ answer = wrap-around} \\
= \text{carry 1 out of MSB = math answer too big to fit}
\end{align*}
\]
Unsigned **overflow**

Addition *overflows* if and only if a carry bit is dropped.

\[
\begin{array}{c}
15 \\
+ 2 \\
\hline
1111 \\
+ 0010 \\
\hline
\end{array}
\]
Sign-Magnitude representation?

Most-significant bit (MSB) is *sign bit*

- 0 means non-negative
- 1 means negative

Remaining bits are an unsigned magnitude

8-bit sign-and-magnitude:

- $0x00 = 00000000$ represents _____________
- $0x7F = 01111111$ represents _____________
- $0x85 = 10000101$ represents _____________
- $0x80 = 10000000$ represents _____________

Max and min for $n$ bits? Anything weird here?
Sign-Magnitude Negatives

Cumbersome arithmetic.

Example:

\[ 4 - 3 \neq 4 + (-3) \]

\[
\begin{align*}
0100 \\
+1011
\end{align*}
\]

What about zero?

Sign-magnitude is not such a good idea...
Two’s complement representation

for signed integers

$n$-bit representation

<table>
<thead>
<tr>
<th>Positional representation, <strong>but</strong></th>
<th>negative weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2^{(n-1)}</td>
<td>2^i</td>
</tr>
<tr>
<td>n-1</td>
<td>i</td>
</tr>
</tbody>
</table>
8-bit representations

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

n-bit two's complement numbers:

minimum =

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

1011

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

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

\[ \text{difference} = \_\_ = 2 \_\_ \]

11

\[ \rightarrow -5 \]
Two’s complement addition *Just Works*

2 \hspace{1cm} 0010 
\hline
+ 3 \hspace{1cm} + 0011 
\hline
\underline{3} \hspace{1cm} \underline{+ 0011} 
\hline
\underline{5} \hspace{1cm} \underline{+ 1101} 
\hline
-2 \hspace{1cm} 1110 
\hline
+ 3 \hspace{1cm} + 0011 
\hline
\underline{1} \hspace{1cm} \underline{+ 0011} 
\hline
\underline{10} \hspace{1cm} \underline{+ 1101} 
\hline
Modular Arithmetic
Two’s complement \textit{overflow}

Addition \textit{overflows}

if and only if the \textbf{arguments have the same sign} but the \textbf{result does not}.

if and only if the \textbf{carry in and out} of the \textbf{sign bit} differ.

\begin{align*}
-1 & \quad 1111 \\
+2 & \quad +0010 \\
\hline
+3 & \quad 0110 \\
\hline
6 & \quad 0110 \\
+3 & \quad +0011
\end{align*}

\textbf{Modular Arithmetic}

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

Same exact addition algorithm as for unsigned numbers.

Easy: \[ x + -x = 0 \]
Subtraction is just addition: \[ 4 - 3 = 4 + (-3) \]

MSB is sign: negatives start with 1, non-negatives start with 0

Negative one is 111...11.

Complement rules:
\[ x + \sim x = -1 \]
\[ \sim x + 1 = -x \]
Another derivation

How should we represent 8-bit negatives?

• For all positive integers \(x\), the representations of \(x\) and \(-x\) must sum to zero.
• Use the standard addition algorithm.

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

• Find a rule to represent \(-x\) where that works...
**Convert small** two's complement representation to a **larger** representation?

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

16-bit 2: 0 0 0 0 0 0 0 0 0 0 0 0 1 0

How should these bits be filled?

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

16-bit -4: 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0

ex
Sign extension

Fill new bits with copies of the sign bit.

Casting from smaller to larger signed type does sign extension.
How are **shifting** and **arithmetic** related?

**unsigned**

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

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

**logical shift left:**

**shift in zeros from right**

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

**logical shift right:**

**shift in zeros from left**

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

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

**unsigned**

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

\[
\begin{align*}
1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\
\end{align*}
\]
How are **shifting** and **arithmetic** related?

**signed**

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

**arithmetic** shift right:

shift in copies of MSB from left

\[ x = -101; \]
\[ y = x \gg 2; \]
\[ y == -5 \]

**logical shift left:**

shift in zeros from the right

\[ y = x \ll 2; \]
\[ y == 108 \]
Multiplication
Compute answer in 2x bits. Most languages drop high order half.

\[
\begin{array}{c}
2 \\
\times 3 \\
6
\end{array}
\quad
\begin{array}{c}
0010 \\
\times 0011 \\
00000100
\end{array}
\quad
\begin{array}{c}
-2 \\
\times 2 \\
-4
\end{array}
\quad
\begin{array}{c}
1110 \\
\times 0010 \\
11111100
\end{array}
\]

Modular Arithmetic
Multiplication

Compute answer in \textbf{2x bits}. Most languages drop high order half.

\[
\begin{array}{c}
5 \\
\times 4 \\
\hline
20 \\
\end{array}
\quad
\begin{array}{c}
0101 \\
\times 0100 \\
\hline
000101000 \\
\end{array}
\quad
\begin{array}{c}
4 \\
\times 7 \\
\hline
21 \\
\end{array}
\quad
\begin{array}{c}
1101 \\
\times 0111 \\
\hline
111010111 \\
\end{array}
\quad
\begin{array}{c}
-2 \\
\end{array}

\text{Modular Arithmetic}
Multiplication

Compute answer in **2x bits**. Most languages drop high order half.

\[
\begin{array}{c}
5 \\
\times 5 \\
\hline
25 \\
\hline
-7
\end{array}
\quad
\begin{array}{c}
0101 \\
\times 0101 \\
\hline
00011001
\end{array}
\]

\[
\begin{array}{c}
-2 \\
\times 6 \\
\hline
-12
\end{array}
\quad
\begin{array}{c}
1110 \\
\times 0110 \\
\hline
111101000
\end{array}
\]

Modular Arithmetic
Multiplication by \textit{shift-and-add}

Available operations
\[
\begin{align*}
x \ll k & \quad \text{implements} \quad x \times 2^k \\
x + y & 
\end{align*}
\]

Implement \( y = x \times 24 \) using only \( \ll, +, \text{and integer literals} \)
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;
}
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:

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

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

Includes comparisons (<, >, ==, <=, >=)

<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>-2147483648</td>
<td>unsignd</td>
<td>1</td>
</tr>
<tr>
<td>2147483647U</td>
<td>&lt;</td>
<td>-2147483648</td>
<td>unsignd</td>
<td>1</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: $T_{min} = -2,147,483,648 \quad T_{max} = 2,147,483,647$