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)

"Least-significant" bit(s) or "low-order" bit(s)

MSB  0110010110101001

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

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

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

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

**unsigned overflow** =

**in math**

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

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

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

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<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>...</td>
<td>$2^2$</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>

unsigned places

two's complement places

unsigned range
(2^n values)

two's complement range
(2^{n} values)
4-bit unsigned vs. 4-bit two’s complement

\[ 1011 \]

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

\[ 11 \leftarrow - \rightarrow -5 \]

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

4-bit unsigned

4-bit two's complement
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 =
two’s complement addition

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

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

Modular Arithmetic

Integer Representation 12
two’s complement overflow

Addition overflows
if and only if
if and only if

\[
\begin{array}{ccc}
-1 & + & 2 \\
\hline
\text{1111} & + & \text{0010} \\
\end{array}
\]

\[
\begin{array}{ccc}
6 & + & 3 \\
\hline
\text{0110} & + & \text{0011} \\
\end{array}
\]

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.

\[
\begin{align*}
\text{00000001} & \quad \text{00000010} & \quad \text{00000011} \\
\text{11111111} & \quad \text{11111111} & \quad \text{11111111} \\
\text{+11111111} & \quad \text{+11111110} & \quad \text{+11111101} \\
\text{00000000} & \quad \text{00000000} & \quad \text{00000000}
\end{align*}
\]

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

\[
\begin{align*}
0_0_0_0_0_0_1_0 & \quad 8\text{-bit } 2 \\
\underbrace{0} \underbrace{0} \underbrace{0} \underbrace{0} \underbrace{0} \underbrace{0} \underbrace{0} \underbrace{0} \underbrace{1} \underbrace{0} & \quad 16\text{-bit } 2 \\
1_1_1_1_1_1_0_0 & \quad 8\text{-bit } -4 \\
\underbrace{1} \underbrace{1} \underbrace{1} \underbrace{1} \underbrace{1} \underbrace{1} \underbrace{1} \underbrace{1} \underbrace{0} \underbrace{0} & \quad 16\text{-bit } -4
\end{align*}
\]

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

unsigned

x = 237;
y = x >> 2;
y == 59

n = shift distance in bits, w = width of encoding in bits
two's complement **shifting** and **arithmetic**

**signed**

\[
x = -101;
\]

\[
y = x << 2;
\]

\[
y == 100
\]

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

**arithmetic** shift right

\[
x = -19;
\]

\[
y = x >> 2;
\]

\[
y == -5
\]
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

Parenthesize shifts to be clear about precedence, which may not always be what you expect.
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;
}
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

\[
\begin{array}{c|c}
2 & 0010 \\
\times 3 & \times 0011 \\
6 & 00000110 \\
\end{array}
\]

\[
\begin{array}{c|c}
-2 & 1110 \\
\times 2 & \times 0010 \\
-4 & 11111100 \\
\end{array}
\]

Modular Arithmetic

Integer Representation
multiplication

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

Modular Arithmetic
multiplication

\[
\begin{array}{c}
5 \\
\times 5 \\
25
\end{array}
\begin{array}{c}
0101 \\
\times 0101 \\
00011001
\end{array}
\begin{array}{c}
-7 \\
-2 \\
-12
\end{array}
\begin{array}{c}
1110 \\
11110100
\end{array}
\begin{array}{c}
4
\end{array}
\]
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 unsigned and signed in a single expression, then **signed values are implicitly cast to unsigned**.

---

### How are the argument bits interpreted?

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

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