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 \( \subseteq \mathbb{N} \)  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)

\[ \begin{array}{cccc} 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \end{array} \]

(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

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

\[
\begin{array}{c c c c}
13 & 1101 \\
+ 5 & 0101 \\
\hline
11 & 1011
\end{array}
\]

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

\[
\text{unsigned overflow} =
\]

Unsigned addition overflows if and only if
**sign-magnitude**

Most-significant bit (MSB) is *sign bit*
- 0 means non-negative
- 1 means negative

Remainder bits are an unsigned magnitude

8-bit sign-magnitude: Anything weird here?
- 00000000 represents _____
- 01111111 represents _____
- 10000101 represents _____
- 10000000 represents _____

(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**

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

**unsigned range**

(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**

\[
\begin{array}{cccc}
1 & 0 & 1 & 1 \\
\end{array}
\]

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

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

\[
11 \leftarrow \text{difference} = ____ = 2 \rightarrow -5
\]
8-bit representations

0 0 0 0 1 0 0 1
1 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

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

Modular Arithmetic

Addition overflows

if and only if the arguments have the same sign but the result does not.
if and only if the carry in and carry out of the sign bit differ.

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 a 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 aircraft."

--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.

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

- Find a rule to represent $-x$ where that works...

Convert/cast signed number to larger type.

\[
\begin{array}{cccc}
\text{00000010} & \text{00000010} & \text{00000010} \\
\text{11111100} & \text{11111100} & \text{11111100} \\
\end{array}
\]

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

x = 27;
y = x >> 2;
y == 59
```

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

**shift-and-add**

Available operations

- `x << k` implements `x * 2^k`
- `x + y`

Implement `y = x * 24` using only `<<`, `+`, and integer literals

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

**two's complement shifting and arithmetic**

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

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

**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;
}
```

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.

### Modular Arithmetic

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$x \cdot y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0110</td>
<td>00000110</td>
</tr>
<tr>
<td>6</td>
<td>0011</td>
<td>1111100</td>
</tr>
<tr>
<td>-2</td>
<td>1110</td>
<td>-2</td>
</tr>
<tr>
<td>-4</td>
<td>1110</td>
<td>-4</td>
</tr>
</tbody>
</table>

### Multiplication

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$x \cdot y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0101</td>
<td>0100</td>
</tr>
<tr>
<td>4</td>
<td>001</td>
<td>00010100</td>
</tr>
<tr>
<td>-3</td>
<td>1101</td>
<td>-3</td>
</tr>
<tr>
<td>-5</td>
<td>1111</td>
<td>-5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$x \cdot y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0101</td>
<td>0101</td>
</tr>
<tr>
<td>25</td>
<td>001</td>
<td>00011001</td>
</tr>
<tr>
<td>-7</td>
<td>1110</td>
<td>-7</td>
</tr>
<tr>
<td>-12</td>
<td>1110</td>
<td>-12</td>
</tr>
<tr>
<td>-4</td>
<td>1110</td>
<td>-4</td>
</tr>
</tbody>
</table>

See Bits assignment prep exercise.
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.**

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

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