Integer Representation

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

https://cs.wellesley.edu/~cs240/s20/
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)
(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>
<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>

$n$-bit unsigned integers:

minimum = 

maximum = 

$$= 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$
modular arithmetic, overflow

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

\[
\begin{array}{cc}
13 & 1101 \\
+ 5 & + 0101 \\
\hline
18 & 11101 \\
\end{array}
\]

\[\begin{align*}
x + y \text{ in } n\text{-bit unsigned arithmetic is} & \quad \text{in math}
\end{align*}\]

\[\begin{align*}
\text{unsigned overflow} &= \text{carry 1 out of MSB} = \text{math answer too big to fit}
\end{align*}\]

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

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

### What's the difference?

- **n-bit unsigned numbers:**
  - minimum =
  - maximum =

<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**
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**

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

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

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

\[ \text{difference} = \underline{__} = 2 \underline{__} \]
two's complement addition

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

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

Addition *overflows* if and only if

if and only if

![Addition example](image)

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.

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

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

\[
\begin{array}{c c}
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
\_ & \_ & \_ & \_ & \_ & \_ & \_ & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
\_ & \_ & \_ & \_ & \_ & \_ & \_ & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
\_ & \_ & \_ & \_ & \_ & \_ & \_ & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
\end{array}
\]

\begin{align*}
\text{8-bit} & \quad 2 \\
\text{16-bit} & \quad 2 \\
\text{8-bit} & \quad -4 \\
\text{16-bit} & \quad -4 \\
\end{align*}

Rule/name?
unsigned **shifting** and **arithmetic**

**unsigned**

\[
x = 27;
\]

\[
y = x \ll 2;
\]

\[
y == 108
\]

**logical shift left**

**unsigned**

\[
x = 237;
\]

\[
y = x \gg 2;
\]

\[
y == 59
\]

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

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

**arithmetic** shift right
\[ x = -101; \]
\[ y = x \gg 2; \]
\[ y == -5 \]

**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?

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

2
\times 3
= 6
0010
\times 0011
= 00000110

-2
\times 2
= -4
1110
\times 0010
= 11111100

Modular Arithmetic
multiplication

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

Modular Arithmetic
multiplication

\[
\begin{array}{ccc}
5 & 0101 \\
\times 5 & x 0101 \\
25 & 00011001 \\
-7 & \\
-2 & 1110 \\
\times 6 & x 0110 \\
-12 & 11110100 \\
4 & \\
\end{array}
\]

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

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