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)

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

$0110010110101001$

MSB

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

\[
\begin{array}{cccc}
11 & 1011 \\
+ 2 & + 0010 \\
\hline
13 & 0101 \\
\end{array}
\]

\[
\begin{array}{cccc}
15 & 1111 \\
0000 \\
0010 \\
0100 \\
0110 \\
\hline
1101 & + 5 + 0101 \\
0010 \\
\end{array}
\]

$x+y$ in $n$-bit unsigned arithmetic is

\[
\text{unsigned overflow } = 
\]

in math

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 _____

**This seems weird here?**

**Arithmetic?**

Example:
- $4 - 3 = 4 + (-3)$

**Zero?**

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

**Unsigned places**

**Two's complement places**

**Unsigned range** (2^n values)
- **- (2^{n-1})**
- **0**
- **2^{(n-1)} - 1**
- **2^n - 1**

**Two's complement range** (2^n values)

---

(4-bit) **Two's Complement Signed Integer Representation**

$$
\begin{array}{cccc}
1 & 0 & 1 & 1 \\
\end{array}
= 1 \times -(2^3) + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0
$$

4-bit two's complement integers:
- **minimum** =
- **maximum** =

---

4-bit **unsigned** vs. 4-bit **two's complement**

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

**Difference** = ____ = 2

---
8-bit representations

00001001 10000001

11111111 00100111

n-bit two's complement numbers:

minimum = maximum =

two’s complement addition

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

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

Modular Arithmetic

Addition overflows
if and only if
if and only if

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

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.

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

- Find a rule to represent \( -x \) where that works...

Convert/cast signed number to larger type.

\[
\begin{array}{ccc}
00000010 & 00000010 & 00000010 \\
\_\_\_\_\_\_\_00000010 & \_\_\_\_\_\_\_00000010 & \_\_\_\_\_\_\_00000010 \\
11111100 & 11111100 & 11111100 \\
\_\_\_\_\_\_\_11111100 & \_\_\_\_\_\_\_11111100 & \_\_\_\_\_\_\_11111100 \\
\end{array}
\]

Rule/name?

Sign extension for two's complement

Casting from smaller to larger signed type does sign extension.
unsigned **shifting** and **arithmetic**

```
x = 27;
y = x << 2;
y == 108
```

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

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

---

**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;
}
```

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
```

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

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

---

See Bits assignment prep exercise.
What does this function compute?

Downsize to fake unsigned nybble type (4 bits) to make this easier to write...

def nybble_puzzle(nybble x, nybble y):
    nybble result = 0;
    for (nybble i = 0; i < 4; i+1):
        if (y & (1 << i)):
            result = result + (x << i);
    return result;

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

Implicit casting: Actually does
```
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></td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td>&lt;</td>
<td>(int)2147483648U</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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