Integer Representation

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

(4-bit) unsigned integer representation

\[
\begin{array}{cccc}
1 & 0 & 1 & 1 \\
8 & 4 & 2 & 1 \\
2^3 & 2^2 & 2^1 & 2^0 \\
3 & 2 & 1 & 0
\end{array}
\]

\[1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 15\]

n-bit unsigned integers:

minimum =

maximum =

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)

<table>
<thead>
<tr>
<th>MSB</th>
<th>011001011010100</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSB</td>
<td>01101110101001</td>
</tr>
</tbody>
</table>

modular arithmetic, overflow

\[
\begin{array}{ccc}
11 & 1011 & 13 \\
+ 2 & + 0010 & + 5 \\
13 & 1101 & 0101
\end{array}
\]

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

in math

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

Remaining bits are an unsigned magnitude

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

**zero?**

---

**two’s complement vs. unsigned**

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

What's the difference?

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

n-bit two's complement numbers:
- minimum =
- maximum =

---

**(4-bit) two's complement signed integer representation**

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-2^3$</td>
<td>$2^2$</td>
<td>$2^1$</td>
<td>$2^0$</td>
</tr>
</tbody>
</table>

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

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

---

**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 |

---

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

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

\[
1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \quad \text{\(1 \times -2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0\)}
\]

\[
11 \quad \text{difference = ___} \quad -5
\]

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

\[
\begin{array}{c}
-1 & 1111 \\
+2 & +0010
\end{array}
\]

\[
\begin{array}{c}
6 & 0110 \\
+3 & +0011
\end{array}
\]

Modular Arithmetic

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.

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
\end{array}
+ \begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
= \begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

- 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} \\
\_\_\_\_\_\_\_\_\_\_\_\_\_\_ & 0 & 0 & 0 & 0 & 0 & 1 & 0 &\quad 16\text{-bit 2} \\
1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 &\quad 8\text{-bit -4} \\
\_\_\_\_\_\_\_\_\_\_\_\_\_\_ & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 &\quad 16\text{-bit -4}
\end{align*}

Rule/name?

unsigned shifting and arithmetic

\begin{align*}
\text{unsigned} \\
x & = 27; \\
y & = x << 2; \\
y & = 108
\end{align*}

\begin{align*}
\text{logical shift left}
\begin{array}{cccccccc}
0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\
0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0
\end{array}
\end{align*}

\begin{align*}
\text{unsigned} \\
x & = 237; \\
y & = x >> 2; \\
y & = 59
\end{align*}

\begin{align*}
\text{logical shift right}
\begin{array}{cccccccc}
1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\
0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1
\end{array}
\end{align*}
two's complement shifting and arithmetic

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

```
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;
}
```

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

```
x = -101;
y = x >> 2;
y == 108
```

shift-and-add

Available operations
- \( x \ll k \) implements \( x \times 2^k \)
- \( x + y \)

Implement \( y = x \times 2^4 \) using only \( \ll, +, \) and integer literals

What does this function compute?

```
x 0010
x 0011
x 0000
x 0110
x 0111
x 1000
x 1001
x 1010
x 1011
x 1100
x 1101
x 1110
x 1111
```

multiplication

```
2 0010
3 0011
6 0000110
-2 1110
-4 11111100
```

Modular Arithmetic
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
```
ux; // tx = (int)ux;
uy; // uy = (unsigned)ty;
```
```
void foo(int z) { ... }
foo(ux); // foo((int)ux);
if (tx < ux) ... // if ((unsigned)tx < ux) ...
```

How are the argument bits interpreted?

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>-1</td>
<td>&lt;</td>
<td>-2</td>
<td>unsigned</td>
<td>-2</td>
</tr>
<tr>
<td>-1</td>
<td>&lt;</td>
<td>-1</td>
<td>unsigned</td>
<td>-1</td>
</tr>
<tr>
<td>2147483647</td>
<td>&lt;</td>
<td>2147483647-1</td>
<td>signed</td>
<td>1</td>
</tr>
<tr>
<td>2147483647U</td>
<td>&lt;</td>
<td>-2147483647-1</td>
<td>unsigned</td>
<td>0</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)