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* $\subset \mathbb{N}$ 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) — MSB

```
0110010110101001
```

“Least-significant” bit(s) or “low-order” bit(s) — LSB
(4-bit) unsigned integer representation

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

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

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

\[n\text{-bit unsigned integers:}\]

unsigned minimum = 0

unsigned maximum = \(2^n - 1\)
modular arithmetic, unsigned overflow

\[
\begin{array}{c}
  11_{10} + 2_{10} = 1_{10}
  \\
  1011_{2} + 0010_{2} = 1011_{2}
  \\
  13_{10} + 5_{10} = 13_{10}
  \\
  1101_{2} + 0101_{2} = 1101_{2}
\end{array}
\]

\[x + y \text{ in } n\text{-bit unsigned arithmetic is } (x + y) \mod 2^n \text{ in math}\]

unsigned overflow = "wrong" answer = wrap-around = carry 1 out of MSB = math answer too big to fit

Unsigned addition overflows if and only if a carry bit is dropped.
(4-bit) two's complement signed integer representation

\[
\begin{array}{cccc}
1 & 0 & 1 & 1 \\
-(2^3) & 2^2 & 2^1 & 2^0 \\
\end{array}
\]

\[= 1 \times -(2^3) + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0\]

still only \(2^n\) distinct values, half negative.

4-bit two's complement integers:

- signed minimum = \(- (2^{(n-1)})\)  
  4-bit min: \textbf{1000}
- signed maximum = \(2^{(n-1)} - 1\)  
  4-bit max: \textbf{0111}
alternate signed attempt: 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 _____

---

Arithmetic?

Example:

4 - 3 != 4 + (-3)

\[
\begin{array}{c}
00000100 \\
+10000011 \\
\hline
100000001 \end{array}
\]

Zero?

Note: this is *not* two’s complement
two’s complement vs. unsigned

<table>
<thead>
<tr>
<th></th>
<th>2^{n-1}</th>
<th>2^{n-2}</th>
<th>…</th>
<th>2^1</th>
<th>2^0</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned places</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-2^{n-1}</td>
<td>2^{n-2}</td>
<td>…</td>
<td>2^1</td>
<td>2^0</td>
</tr>
<tr>
<td>two's complement places</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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

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

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

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

\[ 1011 \]

\[ 11 \]

\[ -5 \]
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 = 0 0 1 0 0 1 1 1 1 0 0 1 1 1
maximum =
two's complement (signed) addition

\[\begin{array}{ccc}
2 & 0010 & -2 & 1110 \\
+3 & +0011 & +3 & +1101 \\
5 & 0101 & -5 & 1011 \\
\end{array}\]

\[\begin{array}{ccc}
-2 & 1110 & 2 & 0010 \\
+3 & +0011 & +3 & +1101 \\
1 & 0001 & -1 & 1111 \\
\end{array}\]
two’s complement (signed) overflow

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.

```
  1 1 1
-1  1111
+ 2  + 0010
   0001
```

```
  0 1 1
  6  0110
+ 3  + 0011
   1001
```
Recall: software correctness

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**
The carry algorithm works for everything!

**Sign**
The MSB can be interpreted as a sign bit.

**Negative one**
$-1_{10}$ is encoded as all ones: $0b11...1$

**Complement rules**
$-x == \sim{x} + 1$

5 is $0b0101$

$\sim{0b0101}$ is $0b1010$

$+1$

$0b1011$ is -5

**Even subtraction!**
$x - y == x + \sim{y} == x + \sim{y} + 1$
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.

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

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

<table>
<thead>
<tr>
<th>8-bit</th>
<th>16-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000010</td>
<td>00000010</td>
</tr>
<tr>
<td>11111100</td>
<td>11111100</td>
</tr>
</tbody>
</table>

Rule/name?
Sign extension for two's complement

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

unsigned

\[ x = 27; \]
\[ y = x << 2; \]
\[ y == 108 \]

logical shift left

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

logical shift right

\[ x = 237; \]
\[ y = x >> 2; \]
\[ y == 59 \]
two's complement **shifting** and **arithmetic**

**signed**
- \( x = -101; \)
- \( y = x << 2; \)
- \( y == 108 \)

\[
\begin{align*}
x &= -101; \\
y &= x << 2; \\
y &= 108
\end{align*}
\]

\[
\begin{align*}
x &= -19; \\
y &= x >> 2; \\
y &= -5
\end{align*}
\]

\( n = \text{shift distance in bits, } w = \text{width of encoding in bits} \)
**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

\[ y = x \times (16 + 8); \]

\[ y = (x \times 16) + (x \times 8); \]

\[ y = (x \ll 4) + (x \ll 3) \]

Parenthesize shifts to be clear about precedence, which may not always be what you expect.
What does this function compute?

```c
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...

```cpp
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;
}
```

```
<table>
<thead>
<tr>
<th>i</th>
<th>y&amp;(1&lt;&lt;i)</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
multiplication

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

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

Modular Arithmetic
multiplication

\[
\begin{array}{c}
5 \\
\times 4 \\
\hline
20 \\
4
\end{array}
\quad \begin{array}{c}
0101 \\
\times 0100 \\
\hline
00010100 \\
4
\end{array}
\quad \begin{array}{c}
0101 \\
\times 0111 \\
\hline
111010111 \\
-21
\end{array}
\quad \begin{array}{c}
-3 \\
\times 7 \\
\hline
-21 \\
-5
\end{array}
\]

Modular Arithmetic
multiplication

\[
\begin{array}{c}
\times 5 \\
5 \\
\times 5 \\
25 \\
-7
\end{array}
\begin{array}{c}
0101 \\
\times 0101 \\
00011001 \\
1110
\end{array}
\begin{array}{c}
\times 6 \\
-2 \\
\times 6 \\
-12
\end{array}
\begin{array}{c}
0110 \\
11110100
\end{array}

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

```
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></td>
<td></td>
</tr>
<tr>
<td>2147483647U</td>
<td>&lt;</td>
<td>-2147483647-1</td>
<td></td>
<td></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)
Aside: real-world connection to Alexa’s research

Guest-controlled out-of-bounds read/write on x86_64

Critical alexcrichton published GHSA-ff4p-7xrq-q5r8 on Mar 8

<table>
<thead>
<tr>
<th>Package</th>
<th>Affected versions</th>
<th>Patched versions</th>
</tr>
</thead>
<tbody>
<tr>
<td>@ cranelift-codegen (Rust)</td>
<td>&lt;= 0.93.0, &gt;= 0.84.0</td>
<td>0.93.1, 0.92.1, 0.91.1</td>
</tr>
<tr>
<td>@ wasmtime (Rust)</td>
<td>&lt;= 6.0.0, &gt;= 0.37.0</td>
<td>6.0.1, 5.0.1, 4.0.1</td>
</tr>
</tbody>
</table>

**Description**

**Impact**

Wasmtime's code generator, Cranelift, has a bug on x86_64 targets where address-mode computation mistakenly would calculate a 35-bit effective address instead of WebAssembly's defined 33-bit effective address. This bug means that, with default codegen settings, a wasm-controlled load/store operation could read/write addresses up to 35 bits away from the base of linear memory. Wasmtime's default sandbox settings provide up to 6G of protection from the base of linear memory to guarantee that any memory access in that range will be semantically correct. Due to this bug, however, addresses up to 0xffffffff * 8 + 0xffffffffc = 36507222004 = -346 bytes away from the base of linear memory are possible from guest code. This means that the virtual memory 6G away from the base of linear memory up to ~34G away can be read/written by a malicious module.
Security-critical bug in shift-and-extend code

Conceptually, the compiler tried to convert this with a 32-bit $x$:

$$\text{address} + \text{zero} \text{\_extend}_{64}(x \ll 2)$$

To this:

$$\text{address} + (\text{zero} \text{\_extend}_{64}(x) \ll 2)$$

Incorrect address calculated!