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)
- “Least-significant” bit(s) or “low-order” bit(s)

MSB 0110010110101001 LSB
(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>

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

$n$-bit **unsigned** integers:

**unsigned** minimum = 0

**unsigned** maximum = $2^n - 1$
**modular arithmetic, unsigned overflow**

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

\[ \begin{array}{c}
13 & 1101 \\
+ 5 & + 0101 \\
\hline
18 & 1000 \\
\end{array} \]

\[ \begin{array}{c}
18 & 1000 \\
+ \text{carry} & + 1 \\
\hline
19 & 1001 \\
\end{array} \]

\[ (x + y) \mod 2^N \]

**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: 1000

signed maximum = $2^{(n-1)} - 1$  
4-bit max: 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)

\[
\begin{align*}
00000100 \\
+10000011
\end{align*}
\]

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

unsigned range
$(2^n \text{ values})$

- $(2^{(n-1)})$
0
$2^{(n-1)} - 1$
$2^n - 1$

two's complement range
$(2^n \text{ values})$
4-bit unsigned vs. 4-bit two’s complement

\[ 1011 \]

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

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

difference = ____ = 2____

-5
8-bit representations

\[00001001\quad 10000001\]

\[11111111\quad 00100111\]

n-bit two's complement numbers:

minimum =

maximum =
Consider a single byte: `unsigned char x = 10101100;`.
What is the result of `x << 2`?
Consider a single byte: `unsigned char x = 10101100;`.
What is the result of `x >> 2`?
two's complement (signed) addition

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

\[
\begin{array}{cccc}
1 & 1110 & \text{+3} & 0011 \\
-2 & 1110 & \text{+3} & 1101 \\
-5 & 0001 & -1 & 1111 \\
\end{array}
\]

Modular Arithmetic
two’s complement (signed) \textit{overflow}

\textbf{Addition overflows}

if and only if the \textbf{arguments} have the \textbf{same sign} but the \textbf{result does not}.

if and only if the \textbf{carry in} and \textbf{carry out} of the \textbf{sign bit} differ.

\begin{align*}
-1 & \quad 111 \\
+2 & \quad +0010 \\
\hline
0 & \quad 11 \\
6 & \quad 0110 \\
+3 & \quad +0011 \\
\hline
& \quad 1001
\end{align*}

Modular Arithmetic

Some CPUs/languages raise exceptions on overflow.
C and Java cruise along silently... Feature? Oops?
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: $0b11\ldots1$

**Complement rules**

$-x \equiv \sim x + 1$

5 is $0b0101$

$\sim 0b0101$ is $0b1010$

$+ \quad 1$

$0b1011$ is -5
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.

```
  11111111  111111  11111111
  00000001  00000010  00000011
+11111111  +11111110  +11111101
  00000000  00000000  00000000
```

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

\[\begin{array}{c}
00000010 & \text{8-bit 2} \\
\underline{00000010} & \text{16-bit 2} \\
11111100 & \text{8-bit -4} \\
\underline{11111100} & \text{16-bit -4}
\end{array}\]

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

0 0 0 1 1 0 1 1

logical shift left

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

y = x >> 2;
y == 59

0 0 1 1 1 0 1 1 0 1

logical shift right

unsigned

x = 237;
y = x >> 2;
y == 59

0 0 1 1 1 0 1 1 0 1

two's complement **shifting** and **arithmetic**

**signed**

\[
x = -101;
\]

\[
y = x << 2;
\]

\[
y == 108
\]

**signed**

\[
x = -19;
\]

\[
y = x >> 2;
\]

\[
y == -5
\]
Consider a single signed byte: `signed char x = 10101100;`. What is the result of `x >> 2`?

- 10110011
- 00101011
- 11101100
- 11101011

Start the presentation to see live content. For screen share software, share the entire screen. Get help at pollev.com/app
"shift-and-add"

Available operations

\[ x << k \] implements \( x \times 2^k \)
\[ x + y \]

Implement \( y = x \times 24 \) using only \( <<, +, \) and integer literals

\[ y = x \times (16 + 8); \]
\[ y = (x \times 16) + (x \times 8); \]
\[ y = (x << 4) + (x << 3) \]

Parenthesize shifts to be clear about precedence, which may not always be what you expect.
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></td>
</tr>
<tr>
<td>-1</td>
<td>&lt;</td>
<td>0</td>
<td>signed</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>&lt;</td>
<td>0U</td>
<td>unsigned</td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td>&lt;</td>
<td>-2147483647-1</td>
<td>unsigned</td>
<td></td>
</tr>
<tr>
<td>2147483647U</td>
<td>&lt;</td>
<td>-2147483647-1</td>
<td>unsigned</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: \( T_{min} = -2,147,483,648 \quad T_{max} = 2,147,483,647 \)

\( T_{min} \) must be written as \(-2,147,483,647-1\) (see pg. 77 of CSAPP for details)
Let $x$ be an `int` (that is, a 32-bit signed two's complement integer). Write an expression in terms of `$x$` without using constants greater than `0xFF`.

Write an expression that sets all bytes other than the most significant byte to 0.

Nobody has responded yet.

Hang tight! Responses are coming in.
Aside: real-world connection to Alexa’s research

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

Package | Affected versions | Patched versions |
--------|------------------|-----------------|
@ cranefl-codegen (Rust) | <= 0.93.0, >= 0.84.0 | 0.93.1, 0.92.1, 0.91.1 |
@ wasmtime (Rust) | <= 6.0.0, >= 0.37.0 | 6.0.1, 5.0.1, 4.0.1 |

**Severity**

Critical 9.9 / 10

**CVSS base metrics**

- Attack vector: Network
- Attack complexity: Low
- Privileges required: Low
- User Interaction: None
- Scope: Changed
- Confidentiality: High
- Integrity: High
- Availability: High


**CVE ID**

CVE-2023-26489

**Description**

**Impact**

Wasmtime's code generator, Cranefl, 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 = 3650722004 = ~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!