Integer Representation

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

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)

MSB

LSB

(4-bit) unsigned integer representation

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

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

weight

position

n-bit unsigned integers:

unsigned minimum = 0

unsigned maximum = \( 2^n - 1 \)

modular arithmetic, unsigned overflow

\[
\begin{array}{c}
11 + 2 = 0111 + 0010 = 1001 \\
13 + 5 = 1101 + 0101 = 0010 \\
\end{array}
\]

\( x+y \) in n-bit unsigned arithmetic is \( (x+y) \mod 2^n \) 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

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

= \(-2^3 + 2^1 + 2^0\)

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

4-bit two's complement integers:
- signed minimum = \(-2^{(n-1)}\)
- signed maximum = \(2^{(n-1)} - 1\)

4-bit min: 1000
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)
- 00000100
- + 10000011

Zero?

two’s complement vs. unsigned

unsigned places
two’s complement places

unsigned range (\(2^n\) values)
two’s complement range (\(2^n\) 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

11 \rightarrow difference = ____ = 2

1011

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

0 \rightarrow +1

1111

1010

-4

-5
8-bit representations

\[
\begin{align*}
0 & 0 0 1 0 0 1 & 1 & 0 0 0 0 0 0 1 \\
1 & 1 1 1 1 1 1 & 0 & 0 1 0 0 1 1 1 \\
\end{align*}
\]

n-bit two's complement numbers:

minimum =
maximum =

two’s complement (signed) addition

\[
\begin{align*}
2 & 0 0 1 0 & -2 & 1 1 1 0 \\
+ 3 & + 0 0 1 1 & + 3 & + 1 1 0 1 \\
5 & 0 1 0 1 & -5 & 1 0 1 1 \\
\end{align*}
\]

\[
\begin{align*}
1 & 1 1 \\
-2 & 1 1 1 0 & 2 & 0 0 1 0 \\
+ 3 & + 0 0 1 1 & + 3 & + 1 1 0 1 \\
1 & 0 0 0 1 & -1 & 1 1 1 1 \\
\end{align*}
\]


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.

\[
\begin{align*}
-1 & 1 1 1 \\
+ 2 & + 0 0 1 0 & 0 0 0 1 \\
6 & 0 1 1 0 \\
+ 3 & + 0 0 1 1 & 1 0 0 1 \\
\end{align*}
\]

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: $0b\ldots1$
Complement rules
$-x = \overline{x} + 1$
5 is $0b0101$
$\overline{0b0101}$ is $0b1010$
$+1$
$0b1011$ is $-5$

---

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.

$$
\begin{align*}
+1 &\ldots &+1 &\ldots &+1 \\
+0 &\ldots &+0 &\ldots &+0 \\
\end{align*}
$$

- 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>0000010</td>
<td>00000010</td>
</tr>
<tr>
<td>-2</td>
<td>8-bit -4</td>
</tr>
<tr>
<td>11111100</td>
<td>111111100</td>
</tr>
</tbody>
</table>

Rule/name?

---

Sign extension for two's complement

<table>
<thead>
<tr>
<th>8-bit</th>
<th>16-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000010</td>
<td>00000010</td>
</tr>
<tr>
<td>1111100</td>
<td>111111100</td>
</tr>
</tbody>
</table>

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

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

```plaintext
logical shift left
```

```plaintext
unsigned x = 237;
y = x >> 2;
y == 59
```

```plaintext
logical shift right
```

```plaintext
y = x * 2
```

```plaintext
Signed
```

```plaintext
unsigned
```

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

**two’s complement shifting and arithmetic**

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

```plaintext
logical shift left
```

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

```plaintext
arithmetic shift right
```

```plaintext
Signed
```

**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**

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

Parenthesize shifts to be clear about precedence, which may not always be what you expect.

**What does this function compute?**

```plaintext
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?

```
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;
}
```

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

<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>0 0 0 0</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>

Modular Arithmetic

```
multiplication

<table>
<thead>
<tr>
<th>5</th>
<th>0101</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>20</td>
<td>00010100</td>
</tr>
<tr>
<td>-3</td>
<td>1101</td>
</tr>
<tr>
<td>x 7</td>
<td>0111</td>
</tr>
<tr>
<td>-24</td>
<td>1110111</td>
</tr>
<tr>
<td>-5</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>------</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>20</td>
<td>00010100</td>
</tr>
<tr>
<td>-3</td>
<td>1101</td>
</tr>
<tr>
<td>x 7</td>
<td>0111</td>
</tr>
<tr>
<td>-24</td>
<td>1110111</td>
</tr>
<tr>
<td>-5</td>
<td></td>
</tr>
</tbody>
</table>
```

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 = cy;  // uy = (unsigned)cy;
void foo(int x) {...}
```
```
foo(ux);  // foo((int)ux);
if (tx < ux) ...  // if ((unsigned)tx < ux) ...
```

Aside: real-world connection to Alexa’s research

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

<table>
<thead>
<tr>
<th>severity</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>low</td>
<td>guest-controlled out-of-bounds read/write on x86_64</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>platform</th>
<th>architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86_64</td>
<td></td>
</tr>
</tbody>
</table>

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>Argument,1</th>
<th>Op</th>
<th>Argument,2</th>
<th>Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>==</td>
<td>0U</td>
<td></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>(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)

Security-critical bug in shift-and-extend code

Conceptually, the compiler tried to convert this with a 32-bit \( x \):
```
address = zero_extend_64(x << 2)
```
To this:
```
address = (zero_extend_64(x) << 2)
```
Incorrect address calculated!