Integer Representation
Bit shifting

\[ x \ll 2 \]

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

logical shift left 2

lose bits on left

fill with zeroes on right

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

fill with zeroes on left

logical shift right 2

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

arithemetic shift right 2

fill with copies of MSB on left

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

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

\[ x \gg 2 \]
Shift gotchas

- Logical or arithmetic shift right: how do we tell?
  - C: compiler chooses
    - Usually based on type: rain check!
  - Java: >> is arithmetic, >>> is logical

- Shift an $n$-bit type by at least 0 and no more than $n-1$.
- C: other shift distances are undefined.
  - *anything* could happen
- Java: shift distance is used modulo number of bits in shifted type
  - Given int $x$: $x << 34 == x << 2$
Shift and Mask: extract a bit field

Write C code:

extract 2\textsuperscript{nd} most significant byte from a 32-bit integer.

given \ x \ = \ 01100001011000100110001101100100

should return: \ 00000000000000000000000001100010

All other bits are zero. Desired bits in least significant byte.
(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

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

minimum =

maximum =
modular arithmetic, overflow

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

\[ \begin{array}{cc}
1011 & 0010 \\
+ 0010 & + 0101 \\
\hline
1101 & 1101
\end{array} \]

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

in math

**unsigned overflow** =

Unsigned addition **overflows** if and only if
Representing Negative Numbers
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 _____

Anything weird here?

**Arithmetic?**

Example:

4 - 3 != 4 + (-3)

\[
\begin{array}{c}
00000100 \\
+10000011 \\
\hline
00000111 \\
\end{array}
\]

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

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

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

4-bit two's complement integers:

minimum =

maximum =
### two's complement vs. unsigned

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

What's the difference?

$n$-bit minimum =

$n$-bit maximum =
8-bit representations

00001001 10000001
11111111 00100111
4-bit unsigned vs. 4-bit two’s complement

1 0 1 1

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

11

-5

4-bit unsigned

4-bit two's complement
two's complement addition

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

Modular Arithmetic
two’s complement overflow

Addition overflows

-1 1111
+2 +0010

6 0110
+3 +0011

Modular Arithmetic

Some CPUs/languages raise exceptions on overflow. C and Java cruise along silently... Feature? Oops?
Reliability

Ariane 5 Rocket,
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
Convert/cast signed number to larger type.

00000010  8-bit 2

_________00000010  16-bit 2

11111100  8-bit -4

_________11111100  16-bit -4

Rule/name?
unsigned shifting and arithmetic

unsigned
\[
x = 27;
\]
\[
y = x \ll 2;
\]
\[
y == 108
\]

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

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

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

\[
\begin{array}{c|c}
2 & 0010 \\
\times 3 & \times 0011 \\
6 & 00000100 \\
\end{array}
\]

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

Modular Arithmetic
### Modular Arithmetic

<table>
<thead>
<tr>
<th>Operation</th>
<th>Binary 1</th>
<th>Binary 2</th>
<th>Result (Binary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$5 \times 4$</td>
<td>0101</td>
<td>x 0100</td>
<td>00010100</td>
</tr>
<tr>
<td>$20$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$4 \times 7$</td>
<td>1101</td>
<td>x 0111</td>
<td>11101011</td>
</tr>
<tr>
<td>$-21$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

-2

Diagram of Modular Arithmetic: Numbers arranged in a circular format with addition and subtraction operations.
**multiplication**

\[
\begin{array}{c}
5 \\
\times 5 \\
25 \\
-7 \\
-2 \\
\times 6 \\
-12 \\
4 \\
\end{array}
\begin{array}{c}
0101 \\
\times 0101 \\
00011001 \\
1110 \\
\times 0110 \\
11110100 \\
\end{array}
\]
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 Result</th>
<th>Op</th>
<th>Argument₂ Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>==</td>
<td>unsigned 1</td>
</tr>
<tr>
<td>-1</td>
<td>&lt;</td>
<td>signed 1</td>
</tr>
<tr>
<td>-1</td>
<td>&lt;</td>
<td>unsigned 0</td>
</tr>
<tr>
<td>2147483647</td>
<td>&lt;</td>
<td>-2147483648</td>
</tr>
<tr>
<td>2147483647U</td>
<td>&lt;</td>
<td>-2147483648</td>
</tr>
<tr>
<td>-1</td>
<td>&lt;</td>
<td>-2</td>
</tr>
<tr>
<td>(unsigned)-1</td>
<td>&lt;</td>
<td>-2</td>
</tr>
<tr>
<td>2147483647</td>
<td>&lt;</td>
<td>2147483648U</td>
</tr>
<tr>
<td>2147483647</td>
<td>&lt;</td>
<td>(int)2147483648U</td>
</tr>
</tbody>
</table>

- Note: $T_{min} = -2,147,483,648$  $T_{max} = 2,147,483,647$