Integers II
CSE 351 Winter 2017

http://xkcd.com/571/
Administrivia

- No class Monday:

- After today, Lab 1 should be “easy” 😊
Integers

- Binary representation of integers
  - Unsigned and signed
  - Casting in C

- Consequences of finite width representations
  - Overflow, sign extension

- Shifting and arithmetic operations
Encoding Integers

- The hardware (and C) supports two flavors of integers
  - *unsigned* – only the non-negatives
  - *signed* – both negatives and non-negatives

- Cannot represent all integers with \( w \) bits
  - Only \( 2^w \) distinct bit patterns
  - Unsigned values: \( 0 \ldots 2^w - 1 \)
  - Signed values: \( -2^{w-1} \ldots 2^{w-1} - 1 \)

- **Example**: 8-bit integers (i.e., `char`)
Sign and Magnitude

- Designate the high-order bit (MSB) as the “sign bit”
  - sign=0: positive numbers; sign=1: negative numbers

- Positives:
  - Using MSB as sign bit matches positive numbers with unsigned
  - All zeros encoding is still = 0

- Examples (8 bits):
  - 0x00 = 00000000₂ is non-negative, because the sign bit is 0
  - 0x7F = 01111111₂ is non-negative (+127₁₀)
  - 0x85 = 10000101₂ is negative (-5₁₀)
  - 0x80 = 10000000₂ is negative... zero???
Sign and Magnitude

- MSB is the sign bit, rest of the bits are magnitude
- Drawbacks?
Sign and Magnitude

- MSB is the sign bit, rest of the bits are magnitude
- Drawbacks:
  - Two representations of 0 (bad for checking equality)
Sign and Magnitude

- MSB is the sign bit, rest of the bits are magnitude
- Drawbacks:
  - Two representations of 0 (bad for checking equality)
  - Arithmetic is cumbersome
    - Example: $4 - 3 \neq 4 + (-3)$

$$\begin{array}{c|c}
4 & 0100 \\
-3 & 0011 \\
\hline
1 & 0001
\end{array}$$

- Negatives “increment” in wrong direction!

![Sign and Magnitude Diagram](image)
Two’s Complement

- Let’s fix these problems:
  1) “Flip” negative encodings so incrementing works
Two’s Complement

- Let’s fix these problems:
  1) “Flip” negative encodings so incrementing works
  2) “Shift” negative numbers to eliminate \(-0\)

- MSB *still* indicates sign!
Two’s Complement Negatives

- Accomplished with one neat mathematical trick!
  - $b_{w-1}$ has weight $-2^{w-1}$, other bits have usual weights $+2^i$

### 4-bit Examples:
- $1010_2$ unsigned:
  \[ 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 10 \]
- $1010_2$ two’s complement:
  \[ -1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = -6 \]

-1 represented as:
- $1111_2 = -2^3 + (2^3 - 1)$
- MSB makes it super negative, add up all the other bits to get back up to -1
Why Two’s Complement is So Great

- Roughly same number of (+) and (−) numbers
- Positive number encodings match unsigned
- Single zero
- All zeros encoding = 0

Simple negation procedure:
- Get negative representation of any integer by taking bitwise complement and then adding one!
  \[ \sim x + 1 = -x \]
Unsigned vs. Two’s Complement

- 4-bit Example:
  - **Unsigned**: $1x^3 + 0x^2 + 1x^1 + 1x^0$
  - **Two’s Complement**: $1x(-2^3) + 0x^2 + 1x^1 + 1x^0$

 Unsigned: $1011$
 Two’s Complement: $11$

 (math) difference = $16 = 2^4$

 Unsigned: $15$
 Two’s Complement: $-5$
Unsigned vs. Two’s Complement

- 4-bit Example:

Unsigned:

\[1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0\]

Two’s Complement:

\[1\times(-2^3) + 0\times2^2 + 1\times2^1 + 1\times2^0\]

(math) difference = 16 = 2^4

Unsigned and Two’s Complement for 4-bit Example:

<table>
<thead>
<tr>
<th>Unsigned</th>
<th>Two’s Complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>1111</td>
</tr>
<tr>
<td>0001</td>
<td>1110</td>
</tr>
<tr>
<td>0010</td>
<td>1101</td>
</tr>
<tr>
<td>0011</td>
<td>1100</td>
</tr>
<tr>
<td>0100</td>
<td>1011</td>
</tr>
<tr>
<td>0101</td>
<td>1010</td>
</tr>
<tr>
<td>0110</td>
<td>1001</td>
</tr>
<tr>
<td>0111</td>
<td>1000</td>
</tr>
<tr>
<td>1000</td>
<td>0111</td>
</tr>
<tr>
<td>1001</td>
<td>0110</td>
</tr>
<tr>
<td>1010</td>
<td>0101</td>
</tr>
<tr>
<td>1011</td>
<td>0100</td>
</tr>
<tr>
<td>1100</td>
<td>0011</td>
</tr>
<tr>
<td>1101</td>
<td>0010</td>
</tr>
<tr>
<td>1110</td>
<td>0001</td>
</tr>
<tr>
<td>1111</td>
<td>0000</td>
</tr>
</tbody>
</table>

Unsigned (0-15) and Two’s Complement (-7 to +7)
Two’s Complement Arithmetic

- The same addition procedure works for both unsigned and two’s complement integers
  - **Simplifies hardware:** only one algorithm for addition
  - **Algorithm:** simple addition, *discard the highest carry bit*
    - Called modular addition: result is sum *modulo* $2^w$

- **4-bit Examples:**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0100</td>
<td>-4</td>
<td>1100</td>
</tr>
<tr>
<td>+3</td>
<td>+0011</td>
<td>+3</td>
<td>+0011</td>
</tr>
<tr>
<td>=7</td>
<td>=0111</td>
<td>=-1</td>
<td>=1111</td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0100</td>
<td>-3</td>
<td>+1101</td>
</tr>
<tr>
<td>=1</td>
<td>=0001</td>
<td>=10001</td>
<td>=0001</td>
</tr>
</tbody>
</table>
Why Does Two’s Complement Work?

- For all representable positive integers $x$, we want:

  $\begin{align*}
  \text{bit representation of } x \\
  + \text{bit representation of } -x \\
  \hline \\
  0 \quad (\text{ignoring the carry-out bit})
  \end{align*}$

- What are the 8-bit negative encodings for the following?

  $\begin{align*}
  00000001 + ??? &\quad 00000010 + ??? &\quad 11000011 + ??? \\
  \hline \\
  00000000 &\quad 00000000 &\quad 00000000
  \end{align*}$
Why Does Two’s Complement Work?

- For all representable positive integers $x$, we want:
  
  $\text{bit representation of } x + \text{bit representation of } -x = 0$ (ignoring the carry-out bit)

- What are the 8-bit negative encodings for the following?

  $\begin{array}{ccc}
  00000001 & + & 11111111 \\
  100000000 & + & 11111110 \\
  100000000 & + & 00111101
  \end{array}$

  These are the bitwise complement plus 1!

  $-x = \sim x + 1$
Signed/Unsigned Conversion Visualized

- Two’s Complement → Unsigned
  - Ordering Inversion
  - Negative → Big Positive

2’s Complement Range

<table>
<thead>
<tr>
<th>2's Complement Range</th>
<th>Unsigned Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>UMax – 1</td>
</tr>
<tr>
<td>Tmax + 1</td>
<td>Tmax</td>
</tr>
<tr>
<td>Tmax</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>-2</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>TMin</td>
<td>0</td>
</tr>
</tbody>
</table>

- Signed/Unsigned Conversion

- Visualized diagram showing the conversion process from signed to unsigned representation.
Values To Remember

- **Unsigned Values**
  - $\text{UMin} = 0b00...0 = 0$
  - $\text{UMax} = 0b11...1 = 2^w - 1$

- **Two’s Complement Values**
  - $\text{TMin} = 0b10...0 = -2^{w-1}$
  - $\text{TMax} = 0b01...1 = 2^{w-1} - 1$
  - $-1 = 0b11...1$

- **Example:** Values for $w = 32$

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>4,294,967,296</td>
<td>FF FF FF FF</td>
<td>11111111 11111111 11111111 11111111</td>
</tr>
<tr>
<td>Tmax</td>
<td>2,147,483,647</td>
<td>7F FF FF FF</td>
<td>01111111 11111111 11111111 11111111</td>
</tr>
<tr>
<td>Tmin</td>
<td>-2,147,483,648</td>
<td>80 00 00 00</td>
<td>10000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>FF FF FF FF</td>
<td>11111111 11111111 11111111 11111111</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00 00 00</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
</tbody>
</table>
In C: Signed vs. Unsigned

- **Casting**
  - Bits are unchanged, just interpreted differently!
    - int tx, ty;
    - unsigned ux, uy;
  - *Explicit* casting
    - tx = (int) ux;
    - uy = (unsigned) ty;
  - *Implicit* casting can occur during assignments or function calls
    - tx = ux;
    - uy = ty;
  - gcc flag `-Wsign-conversion` produces warnings for implicit casts, but `-Wall` does not
Casting Surprises

- Integer literals (constants)
  - By default, integer constants are considered *signed* integers
    - Hex constants already have an explicit binary representation
  - Use “U” (or “u”) suffix to explicitly force *unsigned*
    - Examples: 0U, 4294967259u

- Expression Evaluation
  - When you mixed unsigned and signed in a single expression, then *signed values are implicitly cast to unsigned*
  - Including comparison operators <, >, ==, <=, >=
Casting Surprises

- 32-bit examples:
  - Tmin = -2,147,483,648, Tmax = 2,147,483,647

<table>
<thead>
<tr>
<th>Left Constant</th>
<th>Op</th>
<th>Right Constant</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0000 0000 0000 0000 0000 0000 0000</td>
<td>==</td>
<td>0U 0000 0000 0000 0000 0000 0000 0000</td>
<td>Unsigned</td>
</tr>
<tr>
<td>-1 1111 1111 1111 1111 1111 1111 1111</td>
<td>&lt;</td>
<td>0 0000 0000 0000 0000 0000 0000 0000</td>
<td>Signed</td>
</tr>
<tr>
<td>-1 1111 1111 1111 1111 1111 1111 1111</td>
<td>&gt;</td>
<td>0U 0000 0000 0000 0000 0000 0000 0000</td>
<td>Unsigned</td>
</tr>
<tr>
<td>2147483647 0111 1111 1111 1111 1111 1111 1111</td>
<td>&gt;</td>
<td>-2147483648 1000 0000 0000 0000 0000 0000 0000</td>
<td>Signed</td>
</tr>
<tr>
<td>2147483647U 0111 1111 1111 1111 1111 1111 1111</td>
<td>&lt;</td>
<td>-2147483648 1000 0000 0000 0000 0000 0000 0000</td>
<td>Unsigned</td>
</tr>
<tr>
<td>-1 1111 1111 1111 1111 1111 1111 1111</td>
<td>&gt;</td>
<td>-2 1111 1111 1111 1111 1111 1111 1111</td>
<td>Signed</td>
</tr>
<tr>
<td>(unsigned) -1 1111 1111 1111 1111 1111 1111 1111</td>
<td>&gt;</td>
<td>-2 1111 1111 1111 1111 1111 1111 1111</td>
<td>Unsigned</td>
</tr>
<tr>
<td>2147483647 0111 1111 1111 1111 1111 1111 1111</td>
<td>&lt;</td>
<td>2147483648U 1000 0000 0000 0000 0000 0000 0000</td>
<td>Unsigned</td>
</tr>
<tr>
<td>2147483647 0111 1111 1111 1111 1111 1111 1111</td>
<td>&gt;</td>
<td>(int) 2147483648U 1000 0000 0000 0000 0000 0000 0000</td>
<td>Signed</td>
</tr>
</tbody>
</table>
Integers

- Binary representation of integers
  - Unsigned and signed
  - Casting in C
- Consequences of finite width representations
  - Overflow, sign extension
- Shifting and arithmetic operations
Arithmetic Overflow

- When a calculation produces a result that can’t be represented in the current encoding scheme
  - Integer range limited by fixed width
  - Can occur in both the positive and negative directions

- Computer handling of overflow
  - CPU *may be* capable of “throwing an exception” for overflow on signed values
  - CPU doesn’t throw exception for unsigned
  - C and Java ignore overflow exceptions... oops!
Overflow: Unsigned

- **Addition**: drop carry bit \((-2^N)\)
  
  \[
  \begin{array}{c}
  \hline \\
  15 & 1111 \\
  + & 0010 \\
  \hline \\
  17 & 10001 \\
  \end{array}
  \]

- **Subtraction**: borrow \((+2^N)\)
  
  \[
  \begin{array}{c}
  \hline \\
  1 & 10001 \\
  - & 0010 \\
  \hline \\
  15 & 1111 \\
  \end{array}
  \]

\(\pm 2^N\) because of modular arithmetic
Overflow: Two’s Complement

- **Addition:** (+) + (+) = (−) result?
  
  \[
  \begin{array}{c}
  6 \\
  + \ 3 \\
  \hline
  9
  \end{array}
  \quad \quad \quad
  \begin{array}{c}
  0110 \\
  + \ 0011 \\
  \hline
  1001
  \end{array}
  \]

  -7

- **Subtraction:** (−) + (−) = (+)?
  
  \[
  \begin{array}{c}
  -7 \\
  - \ 3 \\
  \hline
  -10
  \end{array}
  \quad \quad \quad
  \begin{array}{c}
  1001 \\
  - \ 0011 \\
  \hline
  0110
  \end{array}
  \]

  6

For signed: overflow if operands have same sign and result’s sign is different
Sign Extension

- What happens if you convert a *signed* integral data type to a larger one?
  - e.g., char → short → int → long

**4-bit → 8-bit Example:**

- Positive Case
  
- Add 0’s?

<table>
<thead>
<tr>
<th>4-bit:</th>
<th>0010</th>
<th>=</th>
<th>+2</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-bit:</td>
<td>00000010</td>
<td>=</td>
<td>+2</td>
</tr>
</tbody>
</table>
Sign Extension

- What happens if you convert a *signed* integral data type to a larger one?
  - e.g., `char` → `short` → `int` → `long`

4-bit → 8-bit Example:

- **Positive Case**
  - 4-bit: 0010 = +2
  - 8-bit: 00000010 = +2

- **Negative Case**
  - 4-bit: 1100 = -4
  - 8-bit: 00001100 = +12
  - 10001100 = -116
  - 11111100 = -4
Sign Extension

- **Task:** Given a $w$-bit signed integer $X$, convert it to a $w+k$-bit signed integer $X'$ *with the same value*

- **Rule:** Add $k$ copies of sign bit
  - Let $x_i$ be the $i$-th digit of $X$ in binary
  - $X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_1, x_0$

![Diagram of sign extension](image)
Sign Extension Example

- Convert from smaller to larger integral data types
- C automatically performs sign extension
  - Java too

```
short int x = 12345;
int   ix = (int) x;
short int y = -12345;
int   iy = (int) y;
```

<table>
<thead>
<tr>
<th>Var</th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>12345</td>
<td>30 39</td>
<td>00110000 00111001</td>
</tr>
<tr>
<td>ix</td>
<td>12345</td>
<td>00 00 30 39</td>
<td>00000000 00000000 00110000 00111001</td>
</tr>
<tr>
<td>y</td>
<td>-12345</td>
<td>CF C7</td>
<td>11001111 11000111</td>
</tr>
<tr>
<td>iy</td>
<td>-12345</td>
<td>FF FF CF C7</td>
<td>11111111 11111111 11001111 11000111</td>
</tr>
</tbody>
</table>
Integers

- Binary representation of integers
  - Unsigned and signed
  - Casting in C
- Consequences of finite width representations
  - Overflow, sign extension
- Shifting and arithmetic operations
Shift Operations

- Left shift \((x << n)\) bit vector \(x\) by \(n\) positions
  - Throw away (drop) extra bits on left
  - Fill with 0s on right

- Right shift \((x >> n)\) bit-vector \(x\) by \(n\) positions
  - Throw away (drop) extra bits on right
  - Logical shift (for \textit{unsigned} values)
    - Fill with 0s on left
  - Arithmetic shift (for \textit{signed} values)
    - Replicate most significant bit on left
    - Maintains sign of \(x\)
Shift Operations

- **Left shift ($x << n$)**
  - Fill with 0s on right

- **Right shift ($x >> n$)**
  - Logical shift (for **unsigned** values)
    - Fill with 0s on left
  - Arithmetic shift (for **signed** values)
    - Replicate most significant bit on left

**Notes:**

- Shifts by $n < 0$ or $n \geq w$ (bit width of $x$) are **undefined**
- **In C**: behavior of $>>$ is determined by compiler
  - In gcc / C lang, depends on data type of $x$ (signed/unsigned)
- **In Java**: logical shift is $>>>$ and arithmetic shift is $>>$
Shifting Arithmetic?

- What are the following computing?
  - \( x >> n \)
    - \( \text{0b 0100} >> 1 = \text{0b 0010} \)
    - \( \text{0b 0100} >> 2 = \text{0b 0001} \)
    - **Divide** by \( 2^n \)
  - \( x << n \)
    - \( \text{0b 0001} << 1 = \text{0b 0010} \)
    - \( \text{0b 0001} << 2 = \text{0b 0100} \)
    - **Multiply** by \( 2^n \)

- Shifting is faster than general multiply and divide operations
Left Shifting Arithmetic 8-bit Example

- No difference in left shift operation for unsigned and signed numbers (just manipulates bits)
  - Difference comes during interpretation: \( x \times 2^n \)?

<table>
<thead>
<tr>
<th>Signed</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = 25; )</td>
<td>( 00011001 = 25 )</td>
</tr>
<tr>
<td>( L1=x&lt;&lt;2; )</td>
<td>( 0001100100 = 100 )</td>
</tr>
<tr>
<td>( L2=x&lt;&lt;3; )</td>
<td>( 00011001000 = -56 )</td>
</tr>
<tr>
<td>( L3=x&lt;&lt;4; )</td>
<td>( 000110010000 = -112 )</td>
</tr>
</tbody>
</table>

- signed overflow
- unsigned overflow
Right Shifting Arithmetic 8-bit Examples

- **Reminder:** C operator `>>` does *logical* shift on *unsigned* values and *arithmetic* shift on *signed* values
  - Logical Shift: $x/2^n$?

\[
x_u = 240_u; \quad 11110000 = 240
\]
\[
R1_u = x_u >> 3; \quad 00011110000 = 30
\]
\[
R2_u = x_u >> 5; \quad 0000011110000 = 7
\]

(rounding (down))
Right Shifting Arithmetic 8-bit Examples

- **Reminder**: C operator `>>` does *logical* shift on *unsigned* values and *arithmetic* shift on *signed* values
  - **Arithmetic Shift**: \( x / 2^n \)?

\[
\begin{align*}
xs &= -16; \quad 11110000 \quad = \quad -16 \\
R1s &= xu >> 3; \quad 11111110000 \quad = \quad -2 \\
R2s &= xu >> 5; \quad 1111111110000 \quad = \quad -1
\end{align*}
\]

(rounding (down))
Peer Instruction Question

For the following expressions, find a value of `char x`, if there exists one, that makes the expression `TRUE`. Compare with your neighbor(s)!

- Assume we are using 8-bit arithmetic:
  - `x == (unsigned char) x`
  - `x >= 128U`
  - `x != (x>>2)<<2`
  - `x == -x`
    - Hint: there are two solutions
  - `(x < 128U) && (x > 0x3F)`
Using Shifts and Masks

- Extract the 2\textsuperscript{nd} most significant \textit{byte} of an \texttt{int}:
  - First shift, then mask: \((x\gg\!16) \& \ 0xFF\)

<table>
<thead>
<tr>
<th>x</th>
<th>00000001 00000010 00000011 00000100</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x\gg!16)</td>
<td>00000000 00000000 00000010 00000100</td>
</tr>
<tr>
<td>(0xFF)</td>
<td>00000000 00000000 00000000 11111111</td>
</tr>
<tr>
<td>((x\gg!16) &amp; \ 0xFF)</td>
<td>00000000 00000000 00000000 00000100</td>
</tr>
</tbody>
</table>
Using Shifts and Masks

- **Extract the sign bit of a signed int:**
  - First shift, then mask: \((x>>31) \& 0x1\)
  - Assuming arithmetic shift here, but works in either case
  - Need mask to clear 1s possibly shifted in

<table>
<thead>
<tr>
<th>x</th>
<th>00000001 00000010 00000011 00000100</th>
</tr>
</thead>
<tbody>
<tr>
<td>x&gt;&gt;31</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>0x1</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>(x&gt;&gt;31) &amp; 0x1</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>10000001 00000010 00000011 00000100</th>
</tr>
</thead>
<tbody>
<tr>
<td>x&gt;&gt;31</td>
<td>11111111 11111111 11111111 11111111</td>
</tr>
<tr>
<td>0x1</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>(x&gt;&gt;31) &amp; 0x1</td>
<td>00000000 00000000 00000000 00000001</td>
</tr>
</tbody>
</table>
Using Shifts and Masks

- **Conditionals as Boolean expressions**
  - For `int x`, what does `(x<<31)>>31` do?

<table>
<thead>
<tr>
<th></th>
<th>00000000 00000000 00000000 00000000 0000001</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=!123</td>
<td>00000000 00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>x&lt;&lt;31</td>
<td>10000000 00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>(x&lt;&lt;31)&gt;&gt;31</td>
<td>11111111 11111111 11111111 11111111</td>
</tr>
<tr>
<td>!x</td>
<td>00000000 00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>!x&lt;&lt;31</td>
<td>00000000 00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>(!x&lt;&lt;31)&gt;&gt;31</td>
<td>00000000 00000000 00000000 00000000 00000000</td>
</tr>
</tbody>
</table>

- Can use in place of conditional:
  - In C: `if(x) {a=y;} else {a=z;}` equivalent to `a=x?y:z`;
  - `a=((x<<31)>>31)&y) | (((!x<<31)>>31)&z);`
Summary

- Sign and unsigned variables in C
  - Bit pattern remains the same, just *interpret* differently
  - Strange things can happen with our arithmetic when we convert/cast between sign and unsigned numbers
  - Type of variables affects behavior of operators (shifting, comparison)

- We can only represent so many numbers in $w$ bits
  - When we exceed the limits, *arithmetic overflow* occurs
  - *Sign extension* tries to preserve value when expanding

- Shifting is a useful bitwise operator
  - Can be used in multiplication with constant or bit masking
  - Right shifting can be arithmetic (sign) or logical (0)