Integers II, Floating Point I
CSE 351 Summer 2020

Instructor: Porter Jones
Teaching Assistants: Amy Xu, Callum Walker, Sam Wolfson, Tim Mandzyuk

http://xkcd.com/1953/
Administrivia

- Questions doc: [https://tinyurl.com/CSE351-7-1](https://tinyurl.com/CSE351-7-1)
- hw4 and hw5 due Monday 7/6 – 10:30am
- hw6 and hw7 due Friday 7/10 – 10:30am
  - Will post Monday’s slides later today so you can get started
- Lab 1a due Monday (7/6) (try to finish by Friday!)
  - Submit `pointer.c` and `lab1Areflect.txt` to Gradescope
- Lab 1b released tomorrow, due 7/10
  - Bit manipulation problems using custom data type
  - Today’s bonus slides have helpful examples, tomorrow’s section will have helpful examples too
Gradescope Lab Turnin

- Make sure you pass the File and Compilation Check!

- Doesn’t indicate if you passed all tests, just indicates that all the correct files were found and there were no compilation or runtime errors.

- Use the testing programs we provide to check your solution for correctness (on attu or the VM)
Quick Aside: C Macros

- Lab1b will have you use some C macros for bit masks
- Syntax is of the form:
  ```c
  #define NAME expression
  ```
- Can now use "NAME" instead of "expression" in code
- Useful to help with readability/factoring in code
  - Especially useful for defining constants such as bit masks!
- Are NOT exactly the same as a constant in Java
  - Does naïve copy and replace before compilation.
  - Everywhere the characters "NAME" appear in the code, the characters "expression" will now appear instead.
- See Lecture 4 (Integers I) slides for example usages
Integers

- Binary representation of integers
  - Unsigned and signed
- Shifting and arithmetic operations – useful for Lab 1a
- In C: Signed, Unsigned and Casting
- Consequences of finite width representations
  - Overflow, sign extension
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 $\text{modulo } 2^w$

### 4-bit Examples:

<table>
<thead>
<tr>
<th>HW</th>
<th>TC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>+4</td>
</tr>
<tr>
<td>+0011</td>
<td>+3</td>
</tr>
<tr>
<td>= 0111</td>
<td>+7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HW</th>
<th>TC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1100</td>
<td>-4</td>
</tr>
<tr>
<td>+0011</td>
<td>+3</td>
</tr>
<tr>
<td>= 1111</td>
<td>-1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HW</th>
<th>TC</th>
</tr>
</thead>
<tbody>
<tr>
<td>10100</td>
<td>+4</td>
</tr>
<tr>
<td>+1101</td>
<td>-3</td>
</tr>
<tr>
<td>=1000</td>
<td>+1</td>
</tr>
</tbody>
</table>
Why Does Two’s Complement Work?

- For all representable positive integers \( x \), we want:
  \[
  \begin{align*}
  \text{bit representation of } x + \text{bit representation of } -x + 0 &= 00000000 \\
  \text{bit representation of } x + \text{bit representation of } -x + 0 &= 00000000 \\
  \text{bit representation of } x + \text{bit representation of } -x + 0 &= 00000000
  \end{align*}
  \]

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

  \[
  \begin{align*}
  00000001 + ???????? &= 00000000 \\
  00000010 + ???????? &= 00000000 \\
  11000011 + ???????? &= 00000000
  \end{align*}
  \]
Why Does Two’s Complement Work?

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

  $$\begin{align*}
  x + (\sim x) &= \cdots 11111111 \\
  x + (\sim x + 1) &= 0
  \end{align*}$$

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

  \[
  \begin{array}{ccc}
  00000001 & 00000010 & 11000011 \\
  + 11111111 & + 11111110 & + 00111101 \\
  \hline
  100000000 & 100000000 & 100000000
  \end{array}
  \]

  These are the bitwise complement plus 1!

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

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

\[ \begin{align*}
2^{w-1} - 1 & \geq 0b00\ldots11 \\
2^{w-1} & \geq 0b11\ldots11
\end{align*} \]

\[ \begin{align*}
2^{w-1} & \leq 0b10\ldots00
\end{align*} \]

\[ \begin{align*}
-2^{w-1} & \geq 0b10\ldots00
\end{align*} \]

\[ \begin{align*}
-2^{w-1} & \geq 0b10\ldots00
\end{align*} \]

\[ \begin{align*}
0/\text{UMin} & \quad \text{0/UMin}
\end{align*} \]

\[ \begin{align*}
\text{UMax} & = 0b11\ldots11 = 2^w - 1 \\
\text{UMax} - 1 & \quad \text{UMax - 1}
\end{align*} \]

\[ \begin{align*}
\text{TMax} & \quad \text{TMax}
\end{align*} \]

\[ \begin{align*}
\text{TMin} & \quad \text{TMin}
\end{align*} \]

\[ \begin{align*}
\text{Unsigned Range} & \quad 2^{w-1}
\end{align*} \]
Values To Remember

- **Unsigned Values**
  - UMin = 0b00...0 = 0
  - UMax = 0b11...1 = 2^w - 1

- **Two’s Complement Values**
  - Tmin = 0b10...0 = \(-2^{w-1}\)
  - Tmax = 0b01...1 = 2^{w-1} - 1
  - -1 = 0b11...1

- **Example: Values for w = 64**

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>18,446,744,073,709,551,615</td>
<td>FF FF FF FF FF FF FF FF FF</td>
</tr>
<tr>
<td>Tmax</td>
<td>9,223,372,036,854,775,807</td>
<td>7F FF FF FF FF FF FF FF FF</td>
</tr>
<tr>
<td>Tmin</td>
<td>-9,223,372,036,854,775,808</td>
<td>80 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>FF FF FF FF FF FF FF FF FF</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00 00 00 00 00 00 00 00</td>
</tr>
</tbody>
</table>
Integers

- Binary representation of integers
  - Unsigned and signed
- Shifting and arithmetic operations – useful for Lab 1a
- In C: Signed, Unsigned and Casting
- Consequences of finite width representations
  - Overflow, sign extension
In C: Signed vs. Unsigned

- **Casting**
  - Bits are unchanged, just interpreted differently!
    - `int` tx, ty;
    - `unsigned int` ux, uy;
  - *Explicit* casting
    - `tx = (int) ux;`
    - `uy = (unsigned int) ty;`
  - *Implicit* casting can occur during assignments or function calls
    - `tx = ux;`
    - `uy = ty;`
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>Order</th>
<th>Right Constant</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&lt;=</td>
<td>0U</td>
<td>unsigned</td>
</tr>
<tr>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>&lt;</td>
<td>0</td>
<td>signed</td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>&gt;</td>
<td>0U</td>
<td>unsigned</td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>0000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td>&gt;</td>
<td>-2147483648</td>
<td>signed</td>
</tr>
<tr>
<td>0111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>1000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>2147483647U</td>
<td>&lt;</td>
<td>-2147483648</td>
<td>unsigned</td>
</tr>
<tr>
<td>0111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>1000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>&gt;</td>
<td>-2</td>
<td>signed</td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>1111 1111 1111 1111 1111 1111 1111 1110</td>
<td></td>
</tr>
<tr>
<td>(unsigned) -1</td>
<td>&gt;</td>
<td>-2</td>
<td>unsigned</td>
</tr>
<tr>
<td>1111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>1111 1111 1111 1111 1111 1111 1111 1110</td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td>&lt;</td>
<td>2147483648U</td>
<td>unsigned</td>
</tr>
<tr>
<td>0111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>1000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td>&gt;</td>
<td>(int) 2147483648U</td>
<td>signed</td>
</tr>
<tr>
<td>0111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td>1000 0000 0000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
</tbody>
</table>
Integers

- Binary representation of integers
  - Unsigned and signed
- Shifting and arithmetic operations – useful for Lab 1a
- In C: Signed, Unsigned and Casting
- Consequences of finite width representations
  - Overflow, sign extension
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

C and Java ignore overflow exceptions
- You end up with a bad value in your program and no warning/indication... oops!
Overflow: Unsigned

- **Addition**: drop carry bit \((-2^N)\)

  \[
  \begin{array}{c}
  15 \\
  + 2 \\
  \hline
  17 \\
  \end{array}
  \begin{array}{c}
  1111 \\
  + 0010 \\
  \hline
  10001 \\
  \end{array}
  \]

- **Subtraction**: borrow \((+2^N)\)

  \[
  \begin{array}{c}
  1 \\
  - 2 \\
  \hline
  -1 \\
  \end{array}
  \begin{array}{c}
  10001 \\
  - 0010 \\
  \hline
  1111 \\
  \end{array}
  \]

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

- **Addition:** $(+) + (+) = (-)$ result?
  
  \[
  \begin{array}{c}
  6 \\
  + 3 \\
  \hline
  \text{9} \\
  \end{array}
  \quad +
  \begin{array}{c}
  0110 \\
  + 0011 \\
  \hline
  1001 \\
  \end{array}
  
  -7

- **Subtraction:** $(-) + (-) = (+)$?
  
  \[
  \begin{array}{c}
  -7 \\
  - 3 \\
  \hline
  -10 \\
  \end{array}
  \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
    - 4-bit: 0010 = +2
    - 8-bit: 00000010 = +2
  - Negative Case?
Polling Question [Int II - a]

- Which of the following 8-bit numbers has the same signed value as the 4-bit number 0b1100?
  - Underlined digit = MSB
  - Vote at http://pollev.com/pbjones

A. 0b 0000 1100
B. 0b 1000 1100
C. 0b 1111 1100
D. 0b 1100 1100
E. We’re lost…
Sign Extension

- **Task:** Given a \( w \)-bit signed integer \( X \), convert it to \( w+k \)-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
  - \( X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_1, x_0 \)

\begin{align*}
\text{original } X & \quad \downarrow \quad k \text{ copies of MSB} \\
X & \quad \downarrow \quad \cdots \quad \downarrow \quad \cdots \\
X' & \quad \downarrow \quad \cdots \quad \downarrow \quad \cdots
\end{align*}
Sign Extension Example

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

```c
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>
### Practice Question

For the following expressions, find a value of **signed 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)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Example</th>
<th>All solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>x == (unsigned char) x</td>
<td>x = 0</td>
<td>works for all x</td>
</tr>
<tr>
<td>x &gt;= 128U</td>
<td>x = -1</td>
<td>any x &lt; 0</td>
</tr>
<tr>
<td>x != (x&gt;&gt;2)&lt;&lt;2</td>
<td>x = 3</td>
<td>any x where lowest two bits are not 0b00</td>
</tr>
<tr>
<td>x == -x</td>
<td>x = 6</td>
<td>x = 0b0000..0100 ± 128</td>
</tr>
<tr>
<td>(x &lt; 128U) &amp;&amp; (x &gt; 0x3F)</td>
<td>x = 128</td>
<td>any x where upper two bits are exactly 0b01</td>
</tr>
</tbody>
</table>
Aside: Unsigned Multiplication in C

Operands:
- \( w \) bits

\[
\begin{array}{c}
\text{True Product:} \\
2w \text{ bits}
\end{array}
\]

Discard \( w \) bits:
- \( w \) bits

\[
\begin{array}{c}
\text{UMult}_w(u, v)
\end{array}
\]

- **Standard Multiplication Function**
  - Ignores high order \( w \) bits

- **Implements Modular Arithmetic**
  - \( \text{UMult}_w(u, v) = u \cdot v \mod 2^w \)
Aside: Multiplication with shift and add

- **Operation** \( u \ll k \) gives \( u \times 2^k \)
  - Both signed and unsigned

<table>
<thead>
<tr>
<th>Operand: ( w ) bits</th>
<th>True Product: ( w + k ) bits</th>
<th>Discard ( k ) bits: ( w ) bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>( u )</td>
<td>( u \cdot 2^k )</td>
<td>( \text{UMult}_w(u, 2^k) )</td>
</tr>
<tr>
<td>( 0 \cdots 010 \cdots 00 )</td>
<td>( 0 \cdots 00 )</td>
<td>( \text{TMult}_w(u, 2^k) )</td>
</tr>
</tbody>
</table>

- **Examples:**
  - \( u \ll 3 \) \( == u \times 8 \)
  - \( u \ll 5 - u \ll 3 \) \( == u \times 24 \)

- Most machines shift and add faster than multiply
  - *Compiler generates this code automatically*
Number Representation Revisited

- What can we represent so far?
  - Signed and Unsigned Integers
  - Characters (ASCII)
  - Addresses

- How do we encode the following:
  - Real numbers (e.g. 3.14159)
  - Very large numbers (e.g. $6.02 \times 10^{23}$)
  - Very small numbers (e.g. $6.626 \times 10^{-34}$)
  - Special numbers (e.g. $\infty$, NaN)
Floating Point Topics

- Fractional binary numbers
- IEEE floating-point standard
- Floating-point operations and rounding
- Floating-point in C

There are many more details that we won’t cover
- It’s a 58-page standard...
Representation of Fractions

- “Binary Point,” like decimal point, signifies boundary between integer and fractional parts:

Example 6-bit representation:

Example: $10.1010_2 = 1 \times 2^1 + 1 \times 2^{-1} + 1 \times 2^{-3} = 2.625_{10}$
Representation of Fractions

- “Binary Point,” like decimal point, signifies boundary between integer and fractional parts:

Example 6-bit representation: `xx.yyyy`

- In this 6-bit representation:
  - What is the encoding and value of the smallest (most negative) number?
  - What is the encoding and value of the largest (most positive) number?
  - What is the smallest number greater than 2 that we can represent?
Fractional Binary Numbers

- Bits to right of “binary point” represent fractional powers of 2
- Represents rational number: \[ \sum_{k=-j}^{i} b_k \cdot 2^k \]
Fractional Binary Numbers

- **Value**
  - 5 and 3/4: \(101.11_2\)
  - 2 and 7/8: \(10.111_2\)
  - 47/64: \(0.101111_2\)

- **Observations**
  - Shift left = multiply by power of 2
  - Shift right = divide by power of 2
  - Numbers of the form \(0.111111\ldots_2\) are just below 1.0
    - \(1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \rightarrow 1.0\)
    - Use notation \(1.0 - \varepsilon\)
## Limits of Representation

- **Limitations:**
  - Even given an arbitrary number of bits, can only **exactly** represent numbers of the form $x \times 2^y$ ($y$ can be negative).
  - Other rational numbers have repeating bit representations.

### Value: Binary Representation:

<table>
<thead>
<tr>
<th>Value</th>
<th>Binary Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1/3 = 0.33333..._{10}$</td>
<td>$0.01010101[01]..._2$</td>
</tr>
<tr>
<td>$1/5 = _1$</td>
<td>$0.001100110011[0011 _]..._2$</td>
</tr>
<tr>
<td>$1/10 = _1$</td>
<td>$0.0001100110011[0011 _]..._2$</td>
</tr>
</tbody>
</table>
Fixed Point Representation

- Implied binary point. Two example schemes:
  
  #1: the binary point is between bits 2 and 3
  \[ b_7 b_6 b_5 b_4 b_3 [.] b_2 b_1 b_0 \]
  
  #2: the binary point is between bits 4 and 5
  \[ b_7 b_6 b_5 [.] b_4 b_3 b_2 b_1 b_0 \]

- Wherever we put the binary point, with fixed point representations there is a trade off between the amount of range and precision we have

- Fixed point = fixed range and fixed precision
  
  - range: difference between largest and smallest numbers possible
  - precision: smallest possible difference between any two numbers

- Hard to pick how much you need of each!
Floating Point Representation

- Analogous to scientific notation
  - In Decimal:
    - Not 12000000, but $1.2 \times 10^7$ In C: 1.2e7
    - Not 0.0000012, but $1.2 \times 10^{-6}$ In C: 1.2e-6
  - In Binary:
    - Not 11000.000, but $1.1 \times 2^4$
    - Not 0.000101, but $1.01 \times 2^{-4}$

- We have to divvy up the bits we have (e.g., 32) among:
  - the sign (1 bit)
  - the mantissa (significand)
  - the exponent
**Scientific Notation (Decimal)**

- **mantissa**
  - $6.02_{10} \times 10^{23}$
- **exponent**
- **decimal point**
- **radix (base)**

- **Normalized form**: exactly one digit (non-zero) to left of decimal point

- **Alternatives to representing $1/1,000,000,000$**
  - Normalized: $1.0 \times 10^{-9}$
  - Not normalized: $0.1 \times 10^{-8}, 10.0 \times 10^{-10}$
Scientific Notation (Binary)

- Computer arithmetic that supports this called floating point due to the “floating” of the binary point
  - Declare such variable in C as float (or double)
Scientific Notation Translation

- Convert from scientific notation to binary point
  - Perform the multiplication by shifting the decimal until the exponent disappears
    - Example: $1.011_2 \times 2^4 = 10110_2 = 22_{10}$
    - Example: $1.011_2 \times 2^{-2} = 0.01011_2 = 0.34375_{10}$

- Convert from binary point to *normalized* scientific notation
  - Distribute out exponents until binary point is to the right of a single digit
    - Example: $1101.001_2 = 1.101001_2 \times 2^3$

- **Practice:** Convert $11.375_{10}$ to normalized binary scientific notation
  
  
  $8 + 2 + 1 = 1.011_2$
  
  $0.25 + 0.125 = 0.011_2$
  
  $1.011011 \times 2^3$
Summary

- Sign and unsigned variables in C
  - Bit pattern remains the same, just *interpreted* 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

- Floating point approximates real numbers
  - We will discuss more details on Monday!
Some examples of using shift operators in combination with bitmasks, which you may find helpful for Lab 1.

- Extract the 2\textsuperscript{nd} most significant byte of an int
- Extract the sign bit of a signed int
- Conditionals as Boolean expressions
## Using Shifts and Masks

- **Extract the 2\textsuperscript{nd} most significant byte of an 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 00000001 00000010</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 00000010</td>
</tr>
</tbody>
</table>

- Or first mask, then shift: \((x \& 0xFF0000) \gg\!16\)

<table>
<thead>
<tr>
<th>(x)</th>
<th>00000001 00000010 00000011 00000100</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFF0000</td>
<td>00000000 11111111 00000000 00000000</td>
</tr>
<tr>
<td>(x &amp; 0xFF0000)</td>
<td>00000000 00000010 00000000 00000000</td>
</tr>
<tr>
<td>((x&amp;0xFF0000) \gg!16)</td>
<td>00000000 00000000 00000000 00000010</td>
</tr>
</tbody>
</table>
Using Shifts and Masks

- Extract the *sign bit* of a signed int:
  - First shift, then mask: \((x \gg 31) \& 0x1\)
    - Assuming arithmetic shift here, but this works in either case
    - Need mask to clear 1's 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 \gg 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 \gg 31) &amp; 0x1)</td>
<td>00000000 00000000 00000000 00000000</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>Condition</th>
<th>Binary Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x=!!123</code></td>
<td>00000000 00000000 00000000 00000001</td>
</tr>
<tr>
<td><code>x&lt;&lt;31</code></td>
<td>10000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td><code>(x&lt;&lt;31)&gt;&gt;31</code></td>
<td>11111111 11111111 11111111 11111111</td>
</tr>
<tr>
<td><code>!x</code></td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td><code>!x&lt;&lt;31</code></td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td><code>(!!x&lt;&lt;31)&gt;&gt;31</code></td>
<td>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);`