Integers II

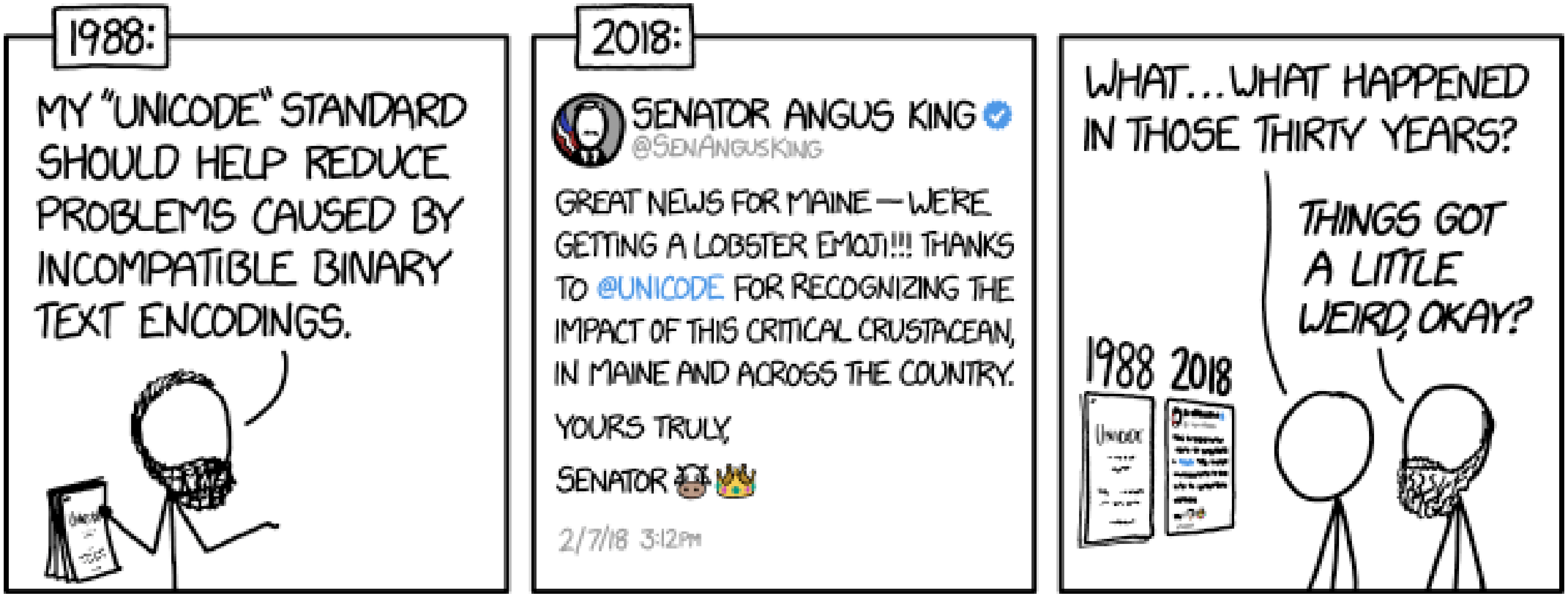
CSE 351 Summer 2018

Instructor:

Justin Hsia

Teaching Assistants:

Josie Lee Natalie Andreeva Teagan Horkan



http://xkcd.com/1953/

Administrivia

- Lab 1a due Friday (6/29)
- Lab 1b due next Thursday (7/5)
  - Bonus slides at the end of today's lecture have relevant examples
- Homework 2 released today, due two Wed from now (7/11)
  - Can start on Integers, will need to wait for Assembly

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

Bits	Unsigned	Signed
`0000`	0	0
`0001`	1	1
`0010`	2	2
`0011`	3	3
`0100`	4	4
`0101`	5	5
`0110`	6	6
`0111`	7	7
`1000`	8	-8
`1001`	9	-7
`1010`	10	-6
`1011`	11	-5
`1100`	12	-4
`1101`	13	-3
`1110`	14	-2
`1111`	15	-1

- 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!

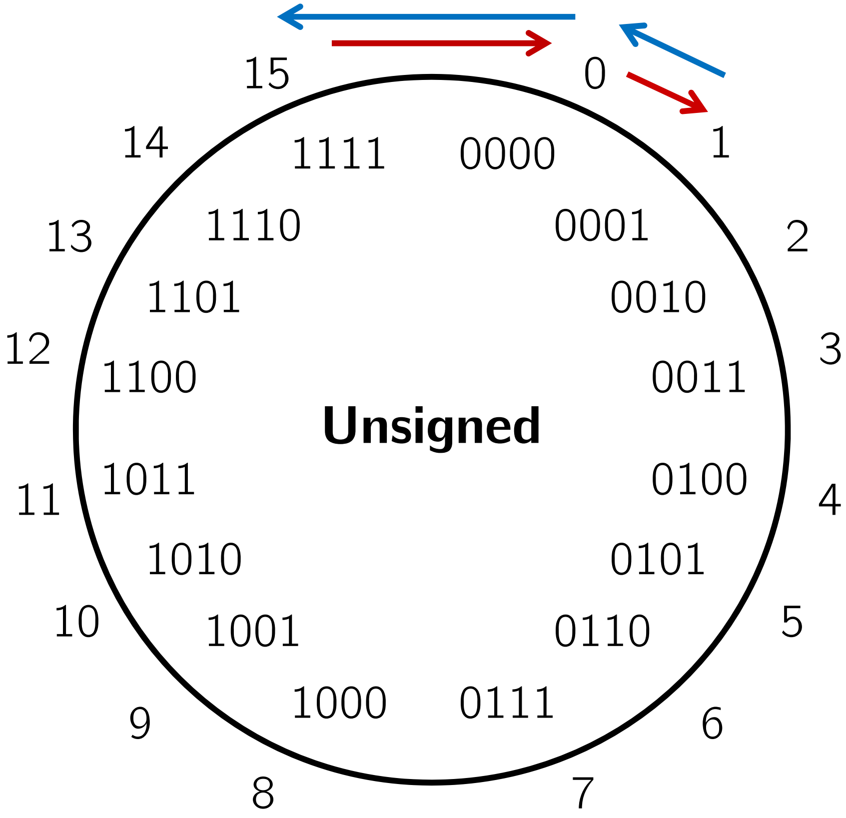
Overflow: Unsigned

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

$$\begin{array}{r} 15 \\ +\ \ 2 \\ \hline \cancel{17} \\ 1 \end{array} \qquad \begin{array}{r} 1111 \\ +\ \ 0010 \\ \hline \cancel{1}0001 \end{array}$$

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

$$\begin{array}{r} 1 \\ -\ \ 2 \\ \hline \cancel{-1} \\ 15 \end{array} \qquad \begin{array}{r} 10001 \\ -\ \ 0010 \\ \hline 1111 \end{array}$$



$\pm 2^N$ because of modular arithmetic

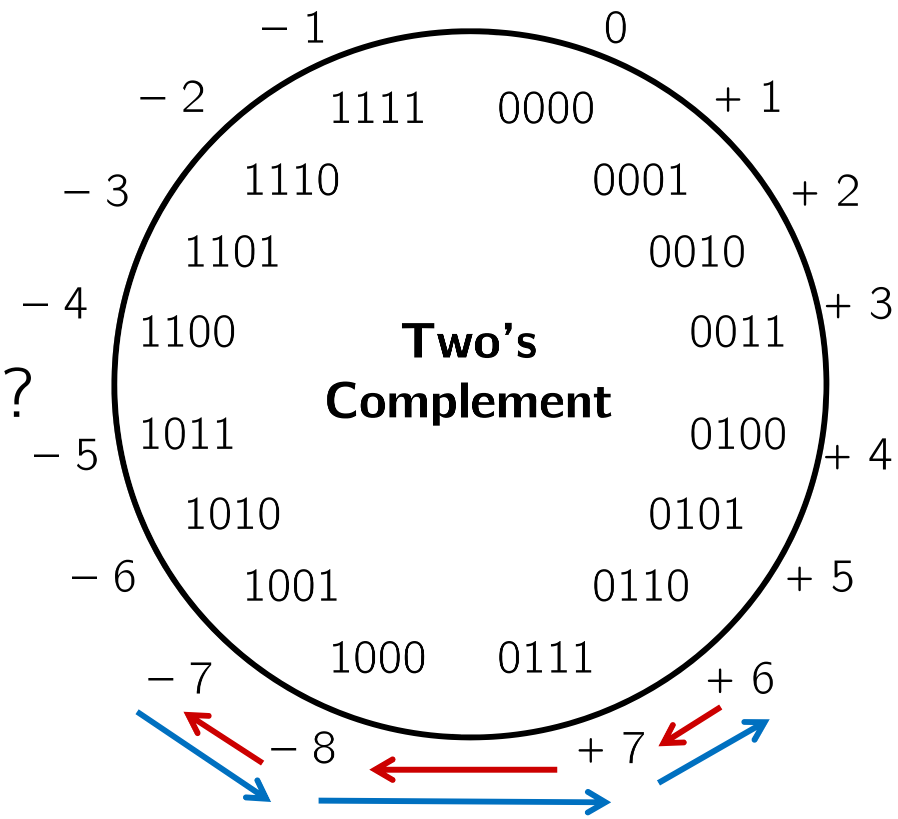
Overflow: Two's Complement

- **Addition:** (+) + (+) = (−) result?

6	0110
+ 3	+ 0011
~~9~~	1001
-7	

- **Subtraction:** (−) + (−) = (+)?

-7	1001
- 3	- 0011
~~-10~~	0110
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?

4-bit: `0010` = +2

8-bit: `00000010` = +2

  - Negative Case?

Peer Instruction Question

- 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/justinh

A. **0b 0000 1100**

B. **0b 1000 1100**

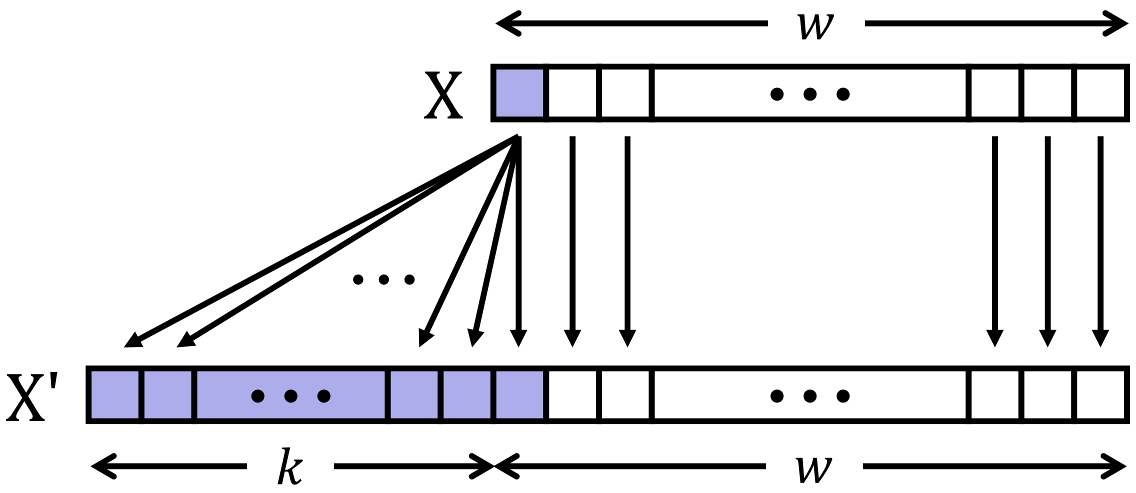
C. **0b 1111 1100**

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
  - $\mathrm{X}' = \underbrace{x_{w-1}, \dots, x_{w-1}}_{k \text{ copies of MSB}}, \underbrace{x_{w-1}, x_{w-2}, \dots, x_1, x_0}_{\text{original } \mathrm{X}}$



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;
```

Var	Decimal	Hex	Binary
x	12345	30 39	00110000 00111001
ix	12345	00 00 30 39	00000000 00000000 00110000 00111001
y	-12345	CF C7	11001111 11000111
iy	-12345	FF FF CF C7	11111111 11111111 11001111 11000111

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

Shift Operations

- Left shift (`x<<n`)
  - Fill with `0`s 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≥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 `>>`

	`x`	`0010 0010`
	`x<<3`	`0001 0000`
logical:	`x>>2`	`0000 1000`
arithmetic:	`x>>2`	`0000 1000`

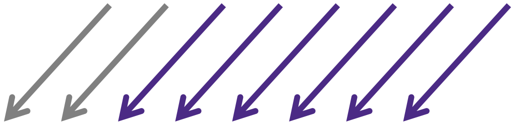
	`x`	`1010 0010`
	`x<<3`	`0001 0000`
logical:	`x>>2`	`0010 1000`
arithmetic:	`x>>2`	`1110 1000`

Shifting Arithmetic?

- What are the following computing?
  - `x>>n`
    - `0b 0100 >> 1 = 0b 0010`
    - `0b 0100 >> 2 = 0b 0001`
    - Divide by 2^n
  - `x<<n`
    - `0b 0001 << 1 = 0b 0010`
    - `0b 0001 << 2 = 0b 0100`
    - Multiply by 2^n
- Shifting is faster than general multiply and divide operations

Left Shift Arithmetic 8-bit Example

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



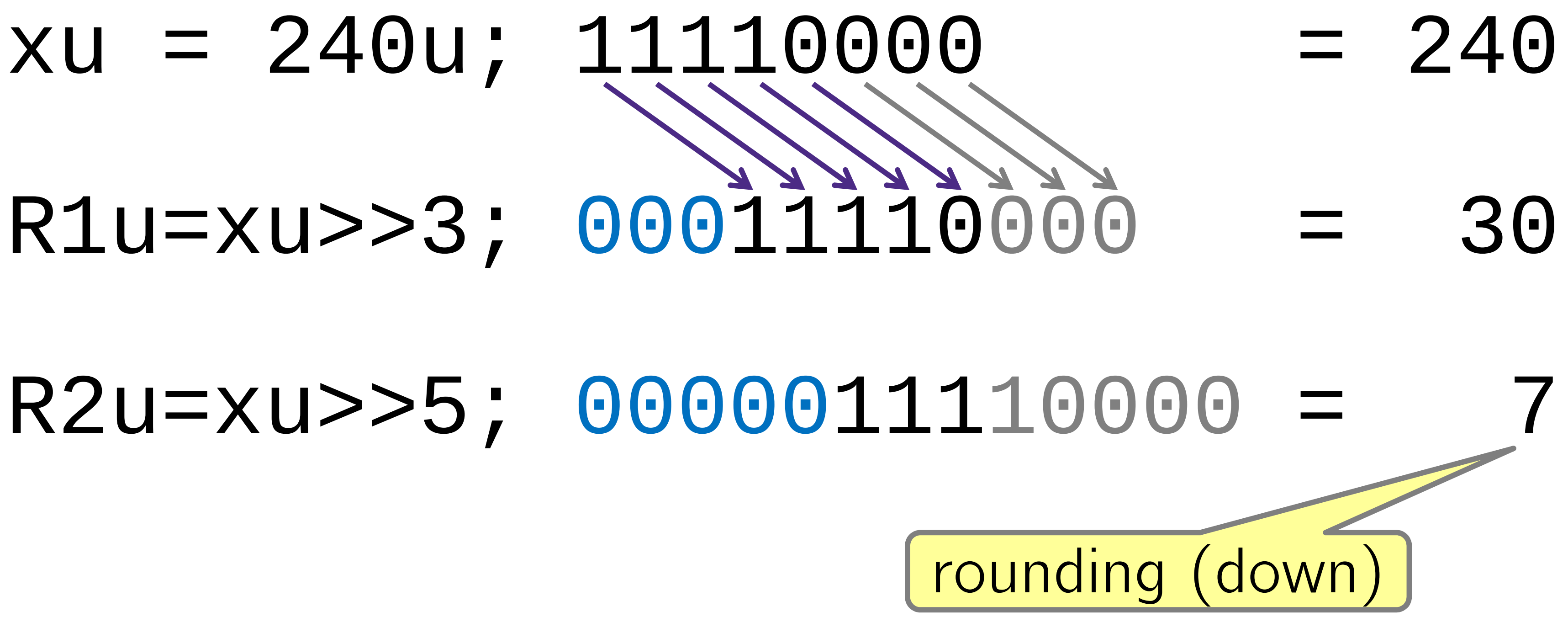
			Signed	Unsigned
`x = 25;`	`00011001`	=	25	25
`L1=x<<2;`	`0001100100`	=	100	100
`L2=x<<3;`	`00011001000`	=	-56	200
`L3=x<<4;`	`000110010000`	=	-112	144

signed overflow

unsigned overflow

Right Shift Arithmetic 8-bit Example

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

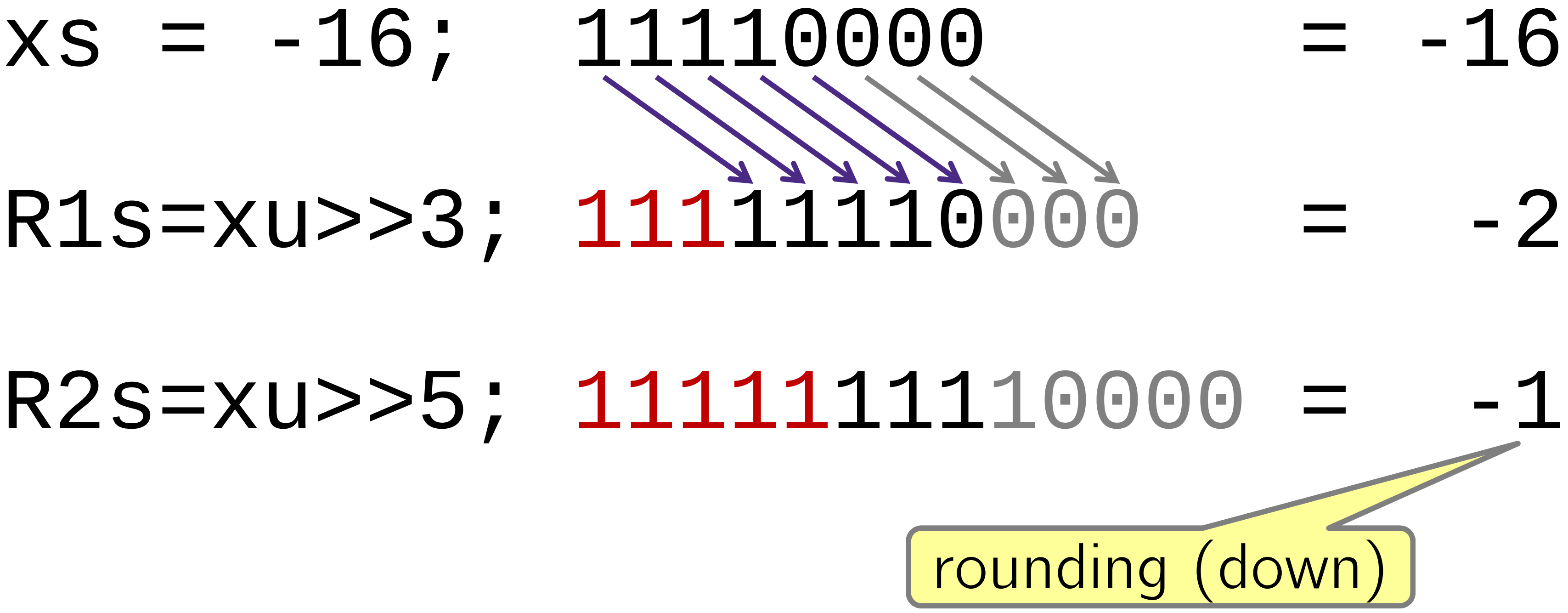


```
xu = 240u; 11110000        = 240
R1u=xu>>3; 00011110000     =  30
R2u=xu>>5; 0000011110000   =   7
```

rounding (down)

Right Shift Arithmetic 8-bit Example

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



```
xs = -16;   11110000          = -16

R1s=xu>>3; 11111110000       =  -2

R2s=xu>>5; 1111111110000     =  -1
```

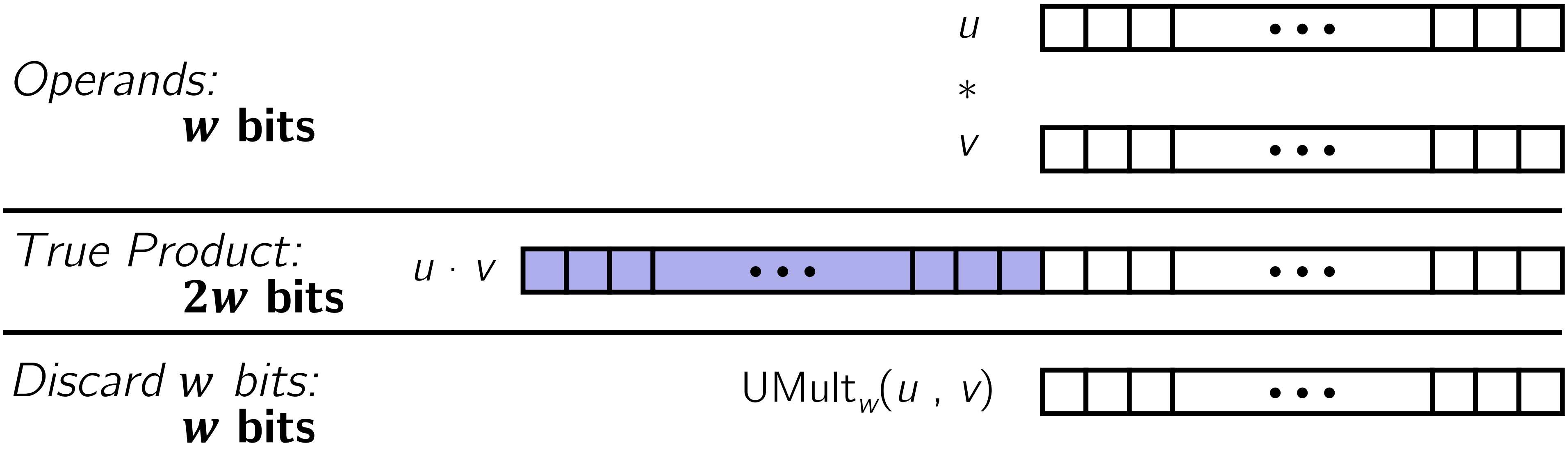
rounding (down)

Peer Instruction 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)`

Unsigned Multiplication in C



Operands:
w bits

u
$*$
v

True Product:
$2w$ bits

$u \cdot v$

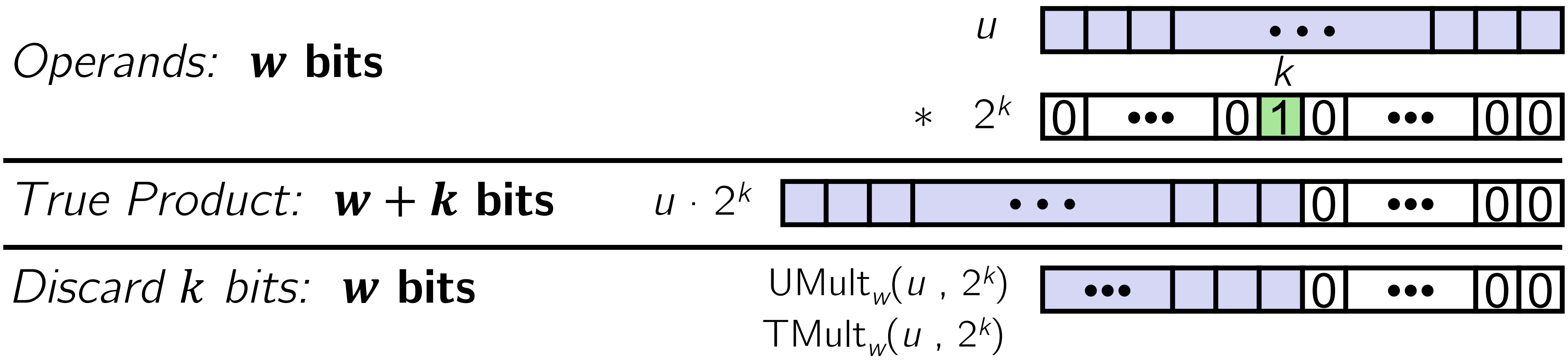
Discard w bits:
w bits

$\mathrm{UMult}_w(u\ ,\ v)$

- Standard Multiplication Function
  - Ignores high order w bits
- Implements Modular Arithmetic
  - $\mathrm{UMult}_w(u\ ,\ v) \qquad = u \cdot v \mod 2^w$

Multiplication with shift and add

- Operation `u<<k` gives `u*`2^k
  - Both signed and unsigned



Operands: **w bits**

u

k

$*\ 2^k$ 0 ••• 0 1 0 ••• 0 0

True Product: **$w + k$ bits** $u \cdot 2^k$ ••• 0 ••• 0 0

Discard k bits: **w bits**

$\text{UMult}_w(u\ ,\ 2^k)$

$\text{TMult}_w(u\ ,\ 2^k)$

••• 0 ••• 0 0

- Examples:
  - `u<<3 == u * 8`
  - `u<<5 - u<<3 == u * 24`
  - Most machines shift and add faster than multiply
    - ***Compiler generates this code automatically***

Number Representation Revisited

- What can we represent in one word?
  - Signed and Unsigned Integers
  - Characters (ASCII)
  - Addresses
- How do we encode the following:
  - Real numbers (*e.g.* 3.14159)
  - Very large numbers (*e.g.* 6.02×10^{23})
  - Very small numbers (*e.g.* 6.626×10^{-34})
  - Special numbers (*e.g.* ∞, NaN)

Floating Point

Floating Point Topics

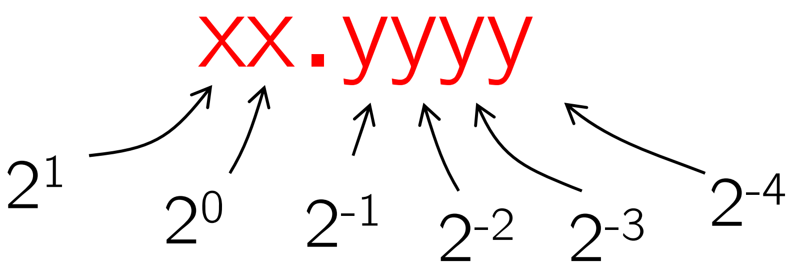
- **Fractional binary numbers**
- IEEE floating-point standard
- Floating-point operations and rounding
- Floating-point in C

- 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:

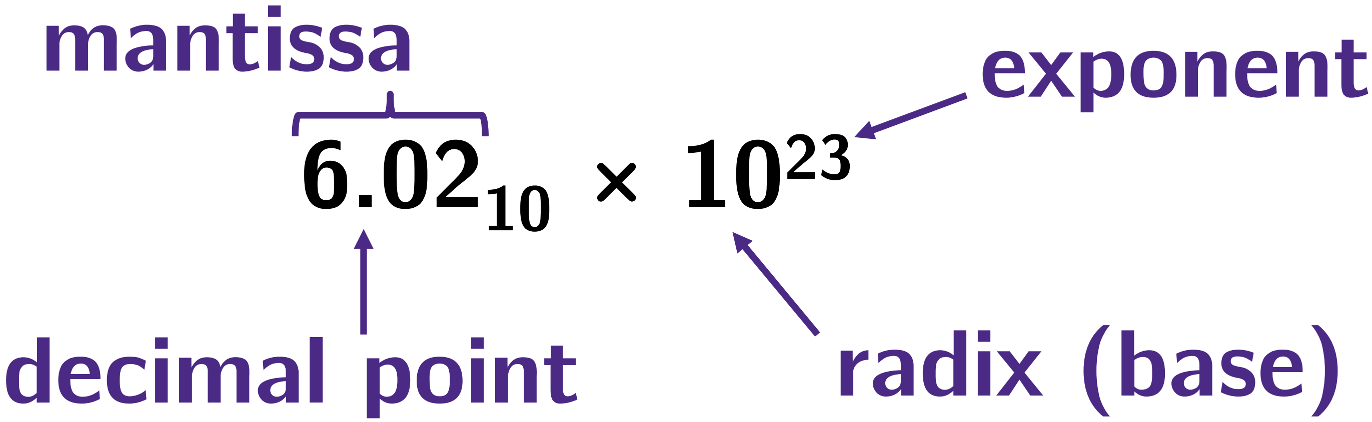
Example 6-bit representation:



- Example: $10.1010_2 = 1\times2^1 + 1\times2^{-1} + 1\times2^{-3} = 2.625_{10}$

- Binary point numbers that match the 6-bit format above range from 0 (00.0000_2) to 3.9375 (11.1111_2)

Scientific Notation (Decimal)



mantissa

exponent

$$\overbrace{6.02}^{\text{mantissa}}{}_{10} \times 10^{23}$$

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: 1.0×10^{-9}
  - Not normalized: 0.1×10^{-8}, 10.0×10^{-10}

Scientific Notation (Binary)

exponent

$$\overbrace{1.01_2}^{\text{mantissa}} \times 2^{-1}$$

binary point

radix (base)

- 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
  - Shift 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 exponent 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
- **Practice:** Convert 1/5 to binary

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

BONUS SLIDES

Some examples of using shift operators in combination with bitmasks, which you may find helpful for Lab 1. We will try to cover these in lecture or section if we have the time.

- Extract the 2nd most significant byte of an `int`
- Extract the sign bit of a signed `int`
- Conditionals as Boolean expressions

Using Shifts and Masks

- Extract the 2nd most significant *byte* of an `int`:
  - First shift, then mask: `(x>>16) & 0xFF`

x	00000001 00000010 00000011 00000100
x>>16	00000000 00000000 00000001 00000010
0xFF	00000000 00000000 00000000 11111111
(x>>16) & 0xFF	00000000 00000000 00000000 00000010

  - Or first mask, then shift: `(x & 0xFF0000)>>16`

x	00000001 00000010 00000011 00000100
0xFF0000	00000000 11111111 00000000 00000000
x & 0xFF0000	00000000 00000010 00000000 00000000
(x&0xFF0000)>>16	00000000 00000000 00000000 00000010

Using Shifts and Masks

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

`x`	`00000001 00000010 00000011 00000100`
`x>>31`	`00000000 00000000 00000000 00000000`
`0x1`	`00000000 00000000 00000000 00000001`
`(x>>31) & 0x1`	`00000000 00000000 00000000 00000000`

`x`	`10000001 00000010 00000011 00000100`
`x>>31`	`11111111 11111111 11111111 11111111`
`0x1`	`00000000 00000000 00000000 00000001`
`(x>>31) & 0x1`	`00000000 00000000 00000000 00000001`

Using Shifts and Masks

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

`x=!!123`	`00000000 00000000 00000000 00000001`
`x<<31`	`10000000 00000000 00000000 00000000`
`(x<<31)>>31`	`11111111 11111111 11111111 11111111`
`!x`	`00000000 00000000 00000000 00000000`
`!x<<31`	`00000000 00000000 00000000 00000000`
`(!x<<31)>>31`	`00000000 00000000 00000000 00000000`

  - 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);`