

# Integer arithmetic and floating point

Computer Systems  
Lecture, Sep 07 2020

Michael Kirkedal Thomsen

**Based on slides by:**

Randal E. Bryant and David R. O'Hallaron

# Today: Integer arithmetic and floating point

## ■ Recap


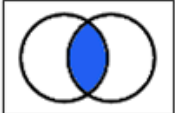

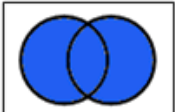

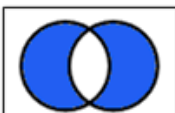



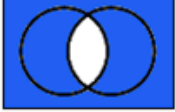





- Representing information as bits
- Bit-level manipulations
- Integers

## ■ Integer arithmetic

## ■ Floating Points

# Everything is bits!

- Why bits? Why no decimals?
- What can we do with bits?
- How do we make integral values?  
Unsigned/signed?
- Student recap  
What can you tell me

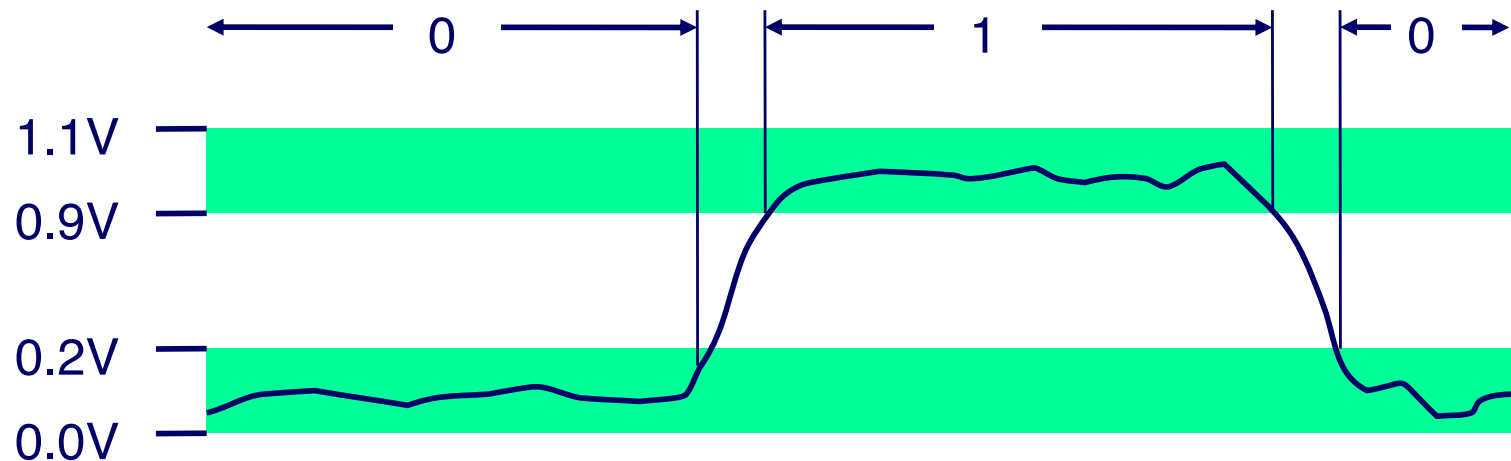
Expression	Symbol	Venn diagram	Boolean algebra	Values		
AND			$A \cdot B$	A	B	Output
				0	0	0
				0	1	0
				1	0	0
				1	1	1
OR			$A + B$	A	B	Output
				0	0	0
				0	1	1
				1	0	1
				1	1	1
XOR			$A \oplus B$	A	B	Output
				0	0	0
				0	1	1
				1	0	1
				1	1	0
NOT			$\bar{A}$	A		Output
				0		1
				1		0
NAND			$\overline{A \cdot B}$	A	B	Output
				0	0	1
				0	1	1
				1	0	1
				1	1	0
NOR			$\overline{A + B}$	A	B	Output
				0	0	1
				0	1	0
				1	0	0
				1	1	0
XNOR			$\overline{A \oplus B}$	A	B	Output
				0	0	1
				0	1	0
				1	0	0
				1	1	1
BUF			$A$	IN		Output
				0		0
				1		1

**Venn Diagram** for logic gates is a schematic representation of A and B overlapping each other inside a rectangle area, the diagram shows the relation of the boolean operators.

# Everything is bits

## ■ Why bits? Electronic Implementation

- Easy to store with bistable elements
- Reliably transmitted on noisy and inaccurate wires



## ■ ... But there exist many models that are not

- E.g. Ternary (3-state) logic, analog computers, quantum computers

# Encoding Byte Values

## ■ Byte = 8 bits

- Binary  $00000000_2$  to  $11111111_2$
- Decimal:  $0_{10}$  to  $255_{10}$
- Hexadecimal  $00_{16}$  to  $FF_{16}$ 
  - Base 16 number representation
  - Use characters '0' to '9' and 'A' to 'F'
  - Write  $FA1D37B_{16}$  in C as
    - `0xFA1D37B`
    - `0xfa1d37b`

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

# Example Data Representations

C Data Type	Typical 32-bit	Typical 64-bit	x86-64
<code>char</code>	1	1	1
<code>short</code>	2	2	2
<code>int</code>	4	4	4
<code>long</code>	4	8	8
<code>int32_t</code>	4	4	4
<code>int64_t</code>	8	8	8
<code>float</code>	4	4	4
<code>double</code>	8	8	8
<code>long double</code>	–	–	10/16
<code>pointer</code>	4	8	8

# Encoding Integers

## Unsigned

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

## Two's Complement

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

```
short int x = 15213;  
short int y = -15213;
```

Sign  
Bit



## ■ C short 2 bytes long

	Decimal	Hex	Binary
<b>x</b>	15213	3B 6D	00111011 01101101
<b>y</b>	-15213	C4 93	11000100 10010011

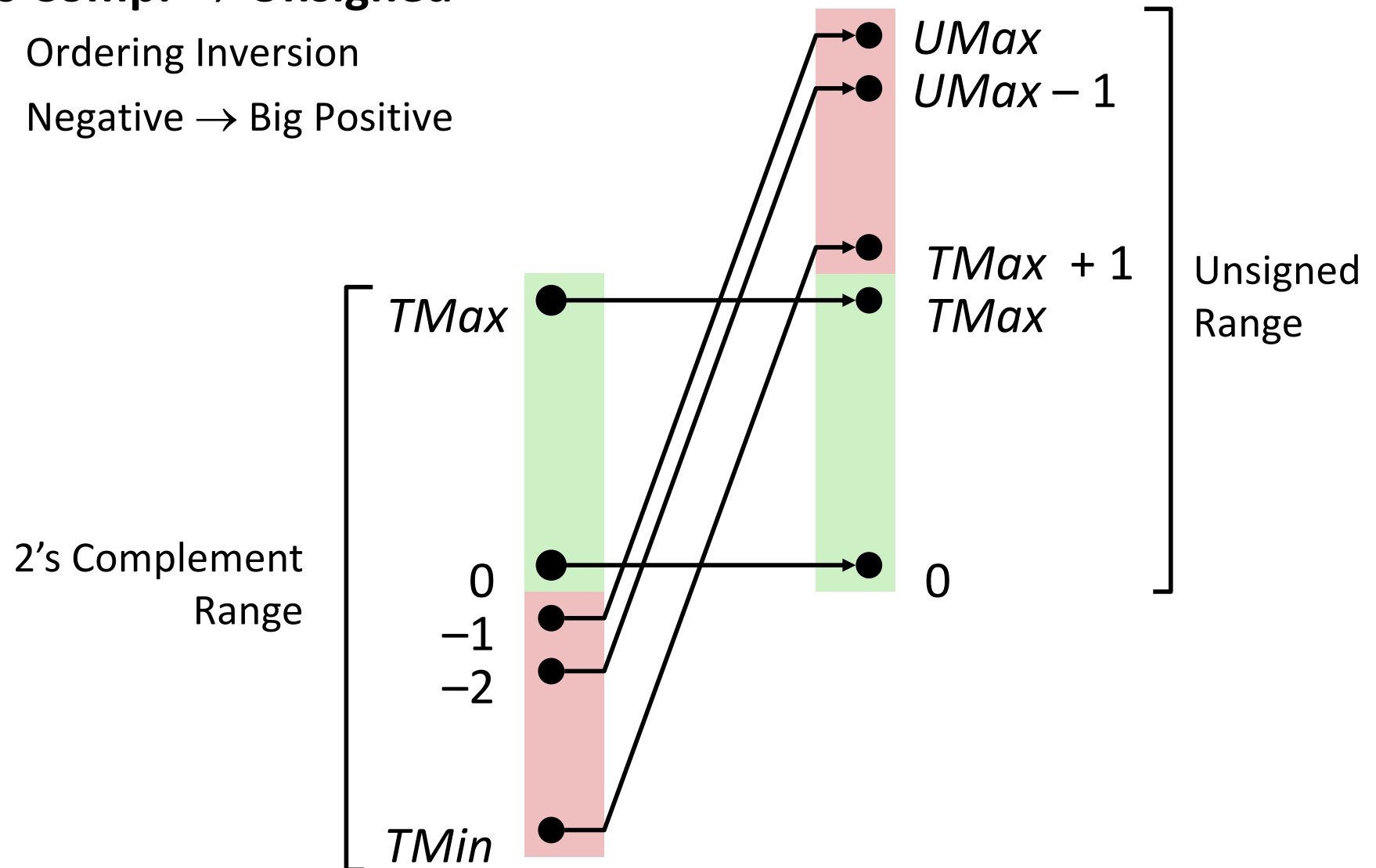
## ■ Sign Bit

- For 2's complement, most significant bit indicates sign
  - 0 for nonnegative
  - 1 for negative

# Conversion Visualized

## ■ 2's Comp. → Unsigned

- Ordering Inversion
- Negative → Big Positive





# Shift Operations

## ■ Left Shift: $x \ll y$

- Shift bit-vector  $x$  left  $y$  positions
  - Throw away extra bits on left
  - Fill with 0's on right

## ■ Right Shift: $x \gg y$

- Shift bit-vector  $x$  right  $y$  positions
  - Throw away extra bits on right
- Logical shift
  - Fill with 0's on left
- Arithmetic shift
  - Replicate most significant bit on left

## ■ Undefined Behavior

- Shift amount  $< 0$  or  $\geq$  word size

Argument $x$	01100010
$\ll 3$	00010000
Log. $\gg 2$	00011000
Arith. $\gg 2$	00011000

Argument $x$	10100010
$\ll 3$	00010000
Log. $\gg 2$	00101000
Arith. $\gg 2$	11101000

# Today: Integer arithmetic and floating point

- **Recap**
- **Integer arithmetic**
  - Addition, negation, multiplication, shifting
  - Summary
- **Floating Points**

# Example: Decimal addition

$$\begin{array}{r} 464 \\ + 875 \\ \hline \end{array}$$

# Example: Binary addition

$$\begin{array}{r} 1011 \\ + 1110 \\ \hline \end{array}$$

# Unsigned Addition

Operands:  $w$  bits



True Sum:  $w+1$  bits



Discard Carry:  $w$  bits



## ■ Standard Addition Function

- Ignores carry output

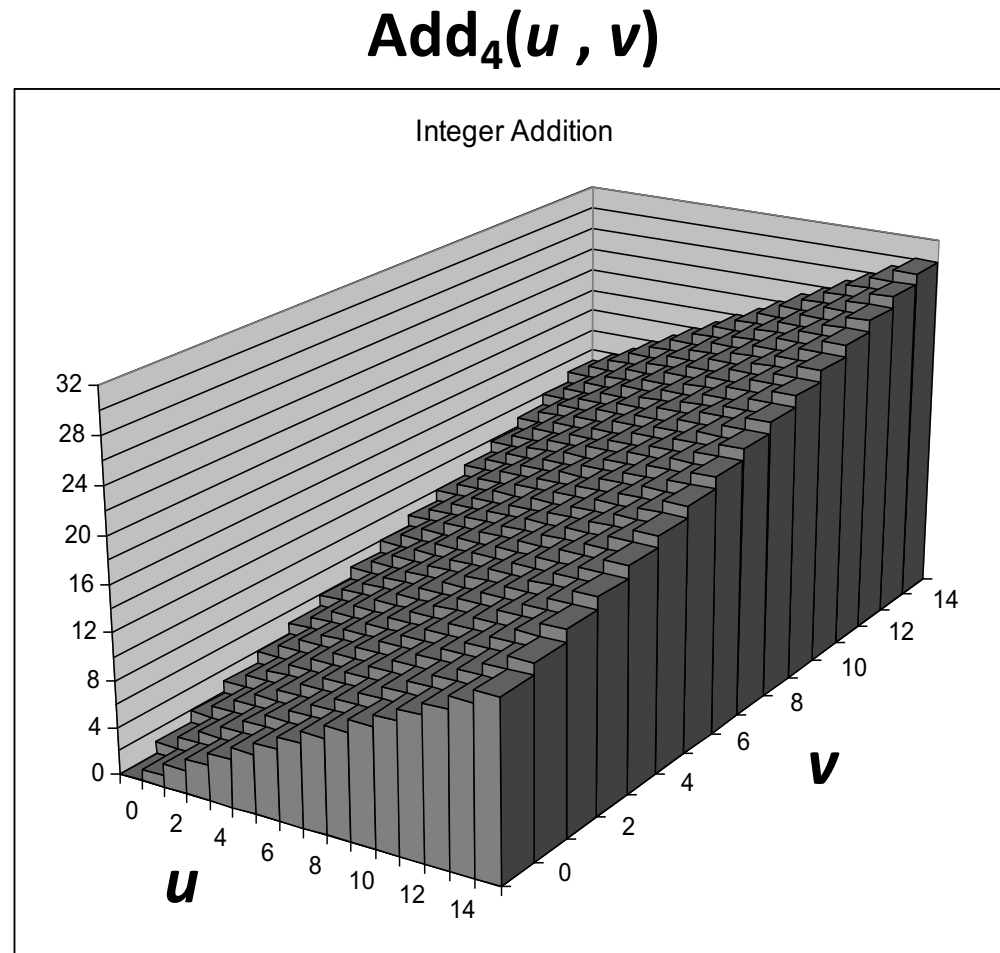
## ■ Implements Modular Arithmetic

$$s = \text{UAdd}_w(u, v) = u + v \bmod 2^w$$

# Visualizing (Mathematical) Integer Addition

## ■ Integer Addition

- 4-bit integers  $u, v$
- Compute true sum  $\text{Add}_4(u, v)$
- Values increase linearly with  $u$  and  $v$
- Forms planar surface

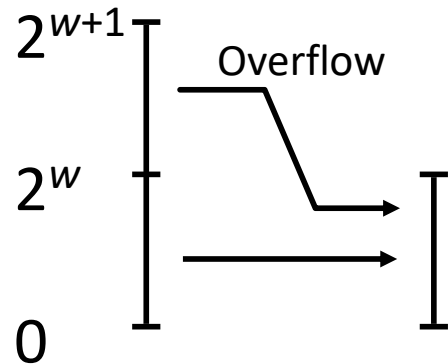


# Visualizing Unsigned Addition

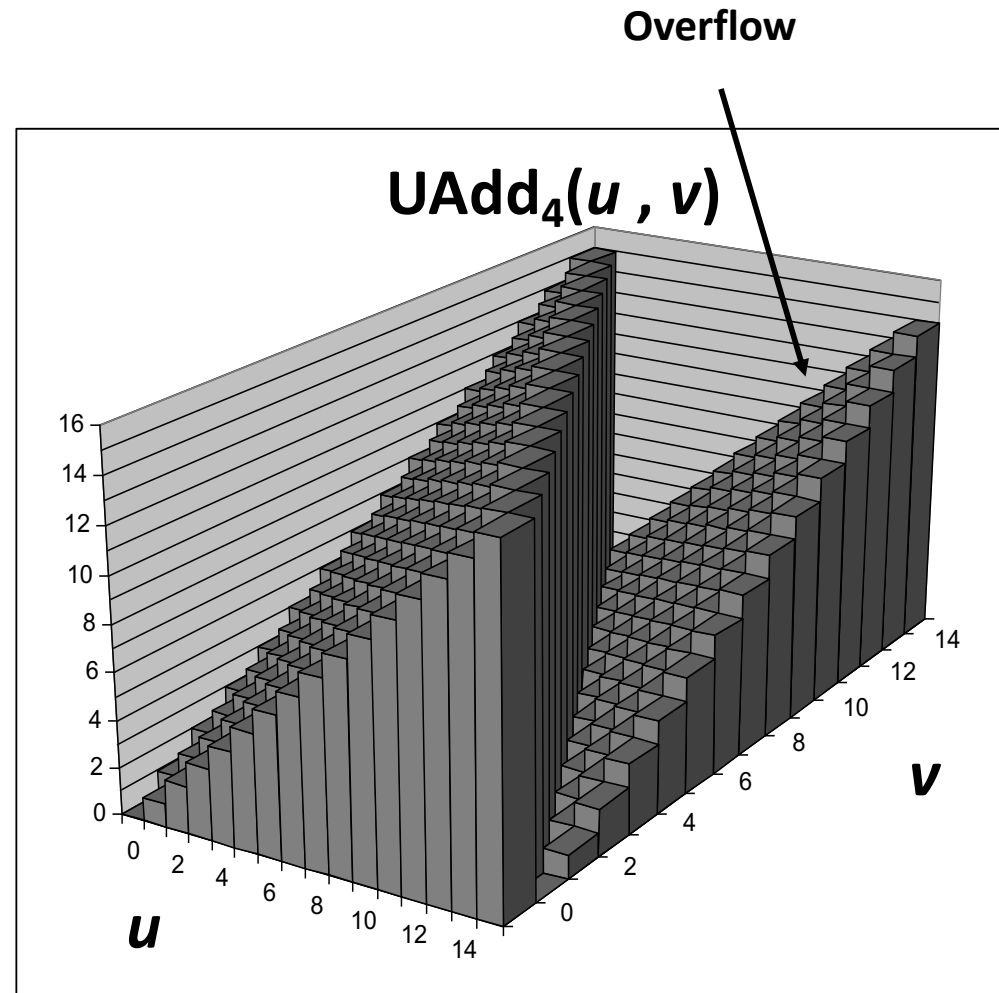
## ■ Wraps Around

- If true sum  $\geq 2^w$
- At most once

True Sum



Modular Sum

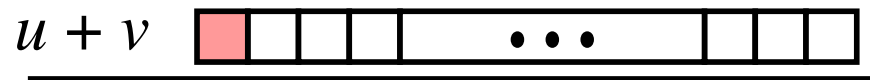


# Two's Complement Addition

Operands:  $w$  bits



True Sum:  $w+1$  bits



Discard Carry:  $w$  bits



## ■ TAdd and UAdd have Identical Bit-Level Behavior

- Signed vs. unsigned addition in C:

```
int s, t, u, v;
```

```
s = (int) ((unsigned) u + (unsigned) v);
```

```
t = u + v
```

- Will give `s == t`

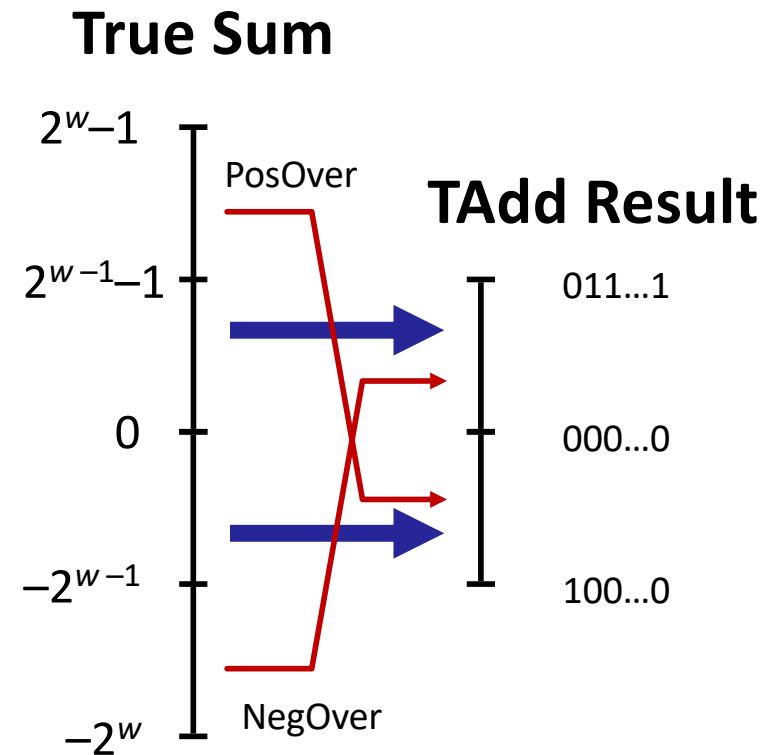


# TAdd Overflow

## ■ Functionality

- True sum requires  $w+1$  bits
- Drop off MSB
- Treat remaining bits as 2's comp. integer

0 111...1  
0 100...0  
0 000...0  
1 011...1  
1 000...0



# Visualizing 2's Complement Addition

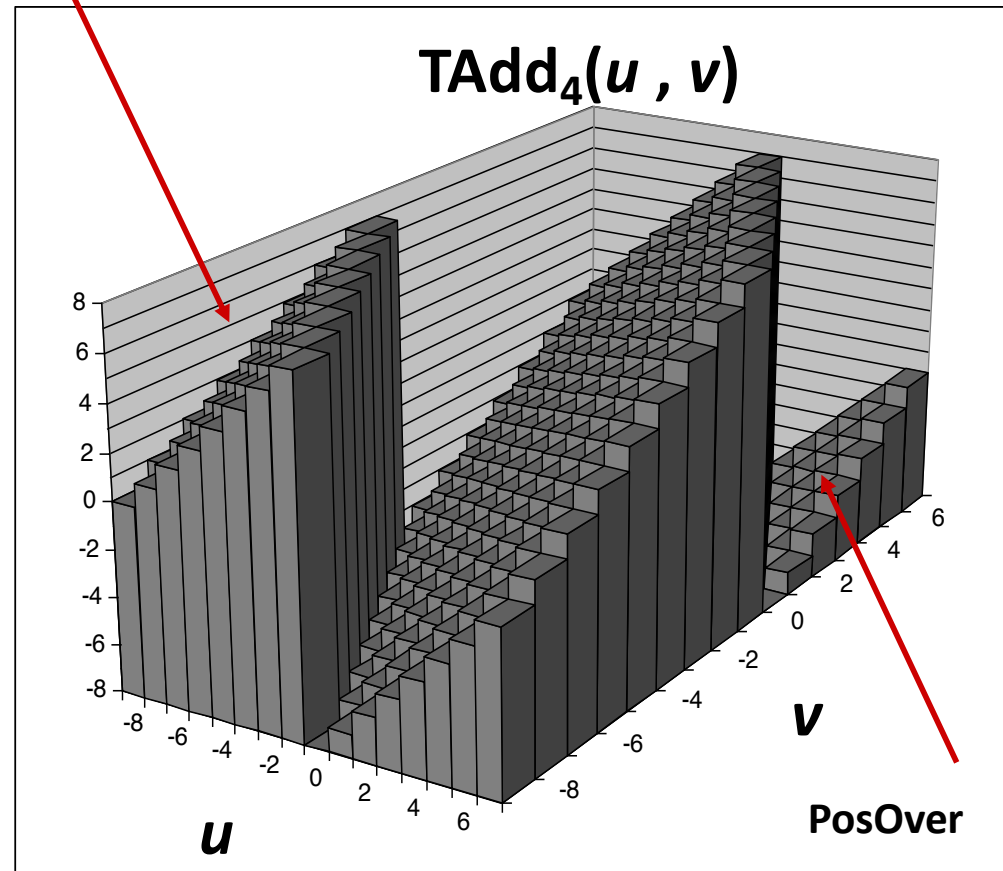
## ■ Values

- 4-bit two's comp.
- Range from -8 to +7

## ■ Wraps Around

- If  $\text{sum} \geq 2^{w-1}$ 
  - Becomes negative
  - At most once
- If  $\text{sum} < -2^{w-1}$ 
  - Becomes positive
  - At most once

NegOver



# Play the game

- <https://topps.diku.dk/compsys/integer-arithmetic.html>

# Multiplication

- **Goal: Computing Product of  $w$ -bit numbers  $x, y$** 
  - Either signed or unsigned
- **But, exact results can be bigger than  $w$  bits**
  - Unsigned: up to  $2w$  bits
    - Result range:  $0 \leq x * y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
  - Two's complement min (negative): Up to  $2w-1$  bits
    - Result range:  $x * y \geq (-2^{w-1}) * (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$
  - Two's complement max (positive): Up to  $2w$  bits, but only for  $(TMin_w)^2$ 
    - Result range:  $x * y \leq (-2^{w-1})^2 = 2^{2w-2}$
- **So, maintaining exact results...**
  - would need to keep expanding word size with each product computed
  - is done in software, if needed
    - e.g., by “arbitrary precision” arithmetic packages

# Unsigned Multiplication in C

Operands:  $w$  bits

$u$  

$*$   $v$  

True Product:  $2 \cdot w$  bits

$u \cdot v$  

Discard  $w$  bits:  $w$  bits

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

## ■ Standard Multiplication Function

- Ignores high order  $w$  bits

## ■ Implements Modular Arithmetic

$$\text{UMult}_w(u, v) = u \cdot v \bmod 2^w$$

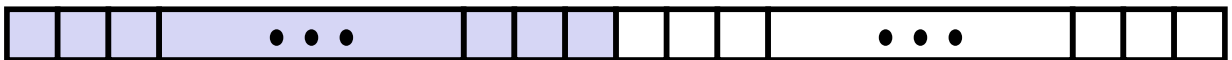
# Signed Multiplication in C

Operands:  $w$  bits


$u$  

$*$   $v$  

True Product:  $2*w$  bits

$u \cdot v$  

Discard  $w$  bits:  $w$  bits

$\text{TMult}_w(u, v)$  

## ■ Standard Multiplication Function

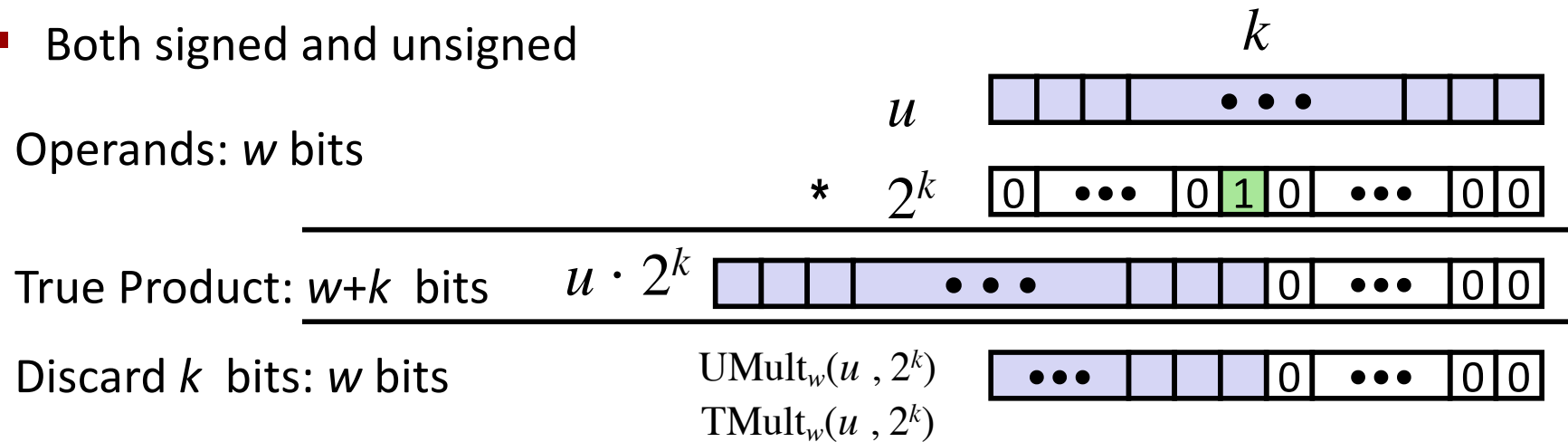
- Ignores high order  $w$  bits
- Some of which are different for signed vs. unsigned multiplication
- Lower bits are the same

# Power-of-2 Multiply with Shift

## ■ Operation

- $u \ll k$  gives  $u * 2^k$
- Both signed and unsigned

Operands:  $w$  bits



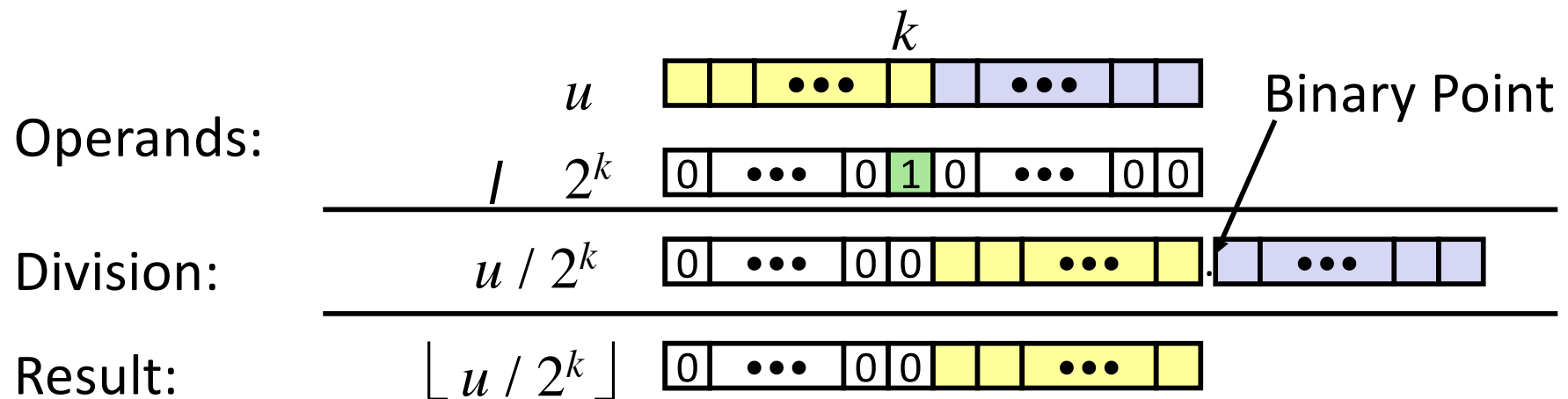
## ■ Examples

- $u \ll 3 == u * 8$
- $(u \ll 5) - (u \ll 3) == u * 24$
- Most machines shift and add faster than multiply
  - Compiler generates this code automatically

# Unsigned Power-of-2 Divide with Shift

## ■ Quotient of Unsigned by Power of 2

- $u \gg k$  gives  $\lfloor u / 2^k \rfloor$
- Uses logical shift



	Division	Computed	Hex	Binary
<b>x</b>	<b>15213</b>	<b>15213</b>	3B 6D	00111011 01101101
<b>x &gt;&gt; 1</b>	<b>7606.5</b>	<b>7606</b>	1D B6	00011101 10110110
<b>x &gt;&gt; 4</b>	<b>950.8125</b>	<b>950</b>	03 B6	00000011 10110110
<b>x &gt;&gt; 8</b>	<b>59.4257813</b>	<b>59</b>	00 3B	00000000 00111011



# Today: Integer arithmetic and floating point

- **Recap**
- **Integer arithmetic**
  - Addition, negation, multiplication, shifting
  - Summary
- **Floating Points**

# Arithmetic: Basic Rules

## ■ Addition:

- Unsigned/signed: Normal addition followed by truncate, same operation on bit level
- Unsigned: addition mod  $2^w$ 
  - Mathematical addition + possible subtraction of  $2^w$
- Signed: modified addition mod  $2^w$  (result in proper range)
  - Mathematical addition + possible addition or subtraction of  $2^w$

## ■ Multiplication:

- Unsigned/signed: Normal multiplication followed by truncate, same operation on bit level
- Unsigned: multiplication mod  $2^w$
- Signed: modified multiplication mod  $2^w$  (result in proper range)

# Why Should I Use Unsigned?

- ***Don't* use without understanding implications**

- Easy to make mistakes

```
unsigned i;  
for (i = cnt-2; i >= 0; i--)  
    a[i] += a[i+1];
```

- Can be very subtle

```
#define DELTA sizeof(int)  
int i;  
for (i = CNT; i-DELTA >= 0; i-= DELTA)  
    . . .
```

# Counting Down with Unsigned

- Proper way to use unsigned as loop index

```
unsigned i;  
for (i = cnt-2; i < cnt; i--)  
    a[i] += a[i+1];
```

- See Robert Seacord, *Secure Coding in C and C++*

- C Standard guarantees that unsigned addition will behave like modular arithmetic
  - $0 - 1 \rightarrow UMax$

- Even better

```
size_t i;  
for (i = cnt-2; i < cnt; i--)  
    a[i] += a[i+1];
```

- Data type `size_t` defined as unsigned value with length = word size
- Code will work even if `cnt = UMax`
- What if `cnt` is signed and `< 0`?

# Why Should I Use Unsigned? (cont.)

- **Do Use When Performing Modular Arithmetic**
  - Multiprecision arithmetic
- **Do Use When Using Bits to Represent Sets**
  - Logical right shift, no sign extension

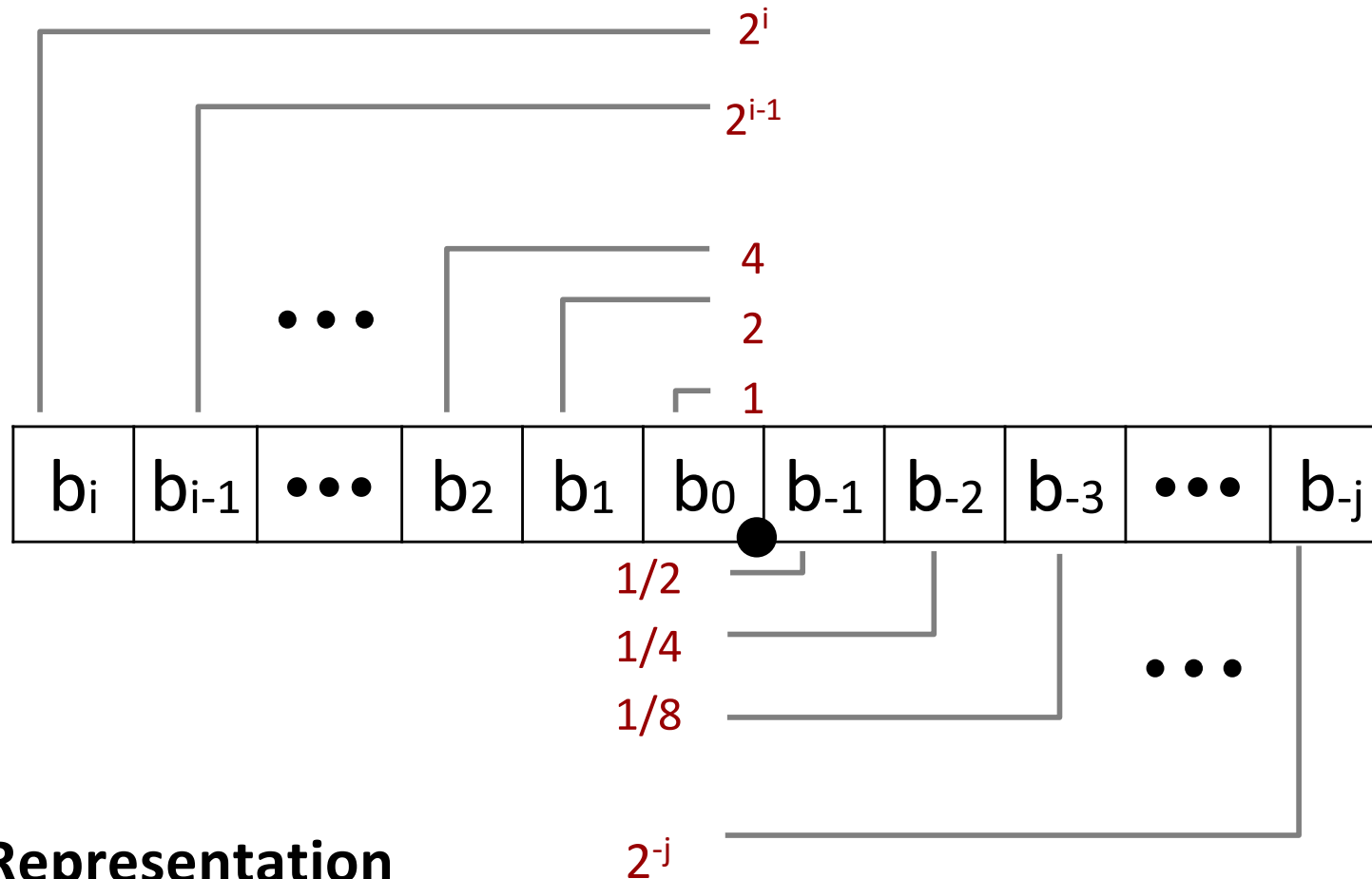
# Today: Integer arithmetic and floating point

- **Recap**
- **Integer arithmetic**
- **Floating Points**
  - Background: Fractional binary numbers
  - IEEE floating point standard: Definition
  - Example and properties
  - Rounding, addition, multiplication
  - Floating point in C
  - Summary

# Fractional binary numbers

- What is  $1011.101_2$ ?

# Fractional Binary Numbers



## ■ Representation

- Bits to right of “binary point” represent fractional powers of 2
- Represents rational number:

$$\sum_{k=-j}^i b_k \times 2^k$$



# Fractional Binary Numbers: Examples

## ■ Value Representation

$5 \frac{3}{4}$	$101.11_2$
$2 \frac{7}{8}$	$10.111_2$
$1 \frac{7}{16}$	$1.0111_2$

## ■ Observations

- Divide by 2 by shifting right (unsigned)
- Multiply by 2 by shifting left
- Numbers of form  $0.111111\dots_2$  are just below 1.0
  - $\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots + \frac{1}{2^i} + \dots \rightarrow 1.0$
  - Use notation  $1.0 - \epsilon$

# Representable Numbers

## ■ Limitation #1

- Can only exactly represent numbers of the form  $x/2^k$ 
  - Other rational numbers have repeating bit representations
- Value      Representation
  - 1/3      0.0101010101 [01] ...<sub>2</sub>
  - 1/5      0.001100110011 [0011] ...<sub>2</sub>
  - 1/10      0.0001100110011 [0011] ...<sub>2</sub>

## ■ Limitation #2

- Just one setting of binary point within the  $w$  bits
  - Limited range of numbers (very small values? very large?)

# Today: Integer arithmetic and floating point

- Recap
- Integer arithmetic
- **Floating Points**
  - Background: Fractional binary numbers
  - IEEE floating point standard: Definition
  - Example and properties
  - Rounding, addition, multiplication
  - Floating point in C
  - Summary

# IEEE Floating Point

## ■ IEEE Standard 754

- Established in 1985 as uniform standard for floating point arithmetic
  - Before that, many idiosyncratic formats
- Supported by all major CPUs

## ■ Driven by numerical concerns

- Nice standards for rounding, overflow, underflow
- Hard to make fast in hardware
  - Numerical analysts predominated over hardware designers in defining standard

# Floating Point Representation

## ■ Numerical Form:

$$(-1)^s M 2^E$$

- Sign bit  $s$  determines whether number is negative or positive
- Significand  $M$  normally a fractional value in range  $[1.0, 2.0)$ .
- Exponent  $E$  weights value by power of two

## ■ Encoding

- MSB  $S$  is sign bit  $s$
- exp field encodes  $E$  (but is not equal to  $E$ )
- frac field encodes  $M$  (but is not equal to  $M$ )



# Precision options

- **Single precision: 32 bits**



- **Double precision: 64 bits**



- **Extended precision: 80 bits (Intel only)**



# “Normalized” Values

$$v = (-1)^s M 2^E$$

- **When:  $\text{exp} \neq 000\dots 0$  and  $\text{exp} \neq 111\dots 1$**
- **Exponent coded as a biased value:  $E = \text{Exp} - \text{Bias}$** 
  - Exp: unsigned value of exp field
  - Bias =  $2^{k-1} - 1$ , where k is number of exponent bits
    - Single precision: 127 (Exp: 1...254, E: -126...127)
    - Double precision: 1023 (Exp: 1...2046, E: -1022...1023)
- **Significand coded with implied leading 1:  $M = 1.\text{xxx}\dots\text{x}_2$** 
  - xxx...x: bits of frac field
  - Minimum when frac=000...0 ( $M = 1.0$ )
  - Maximum when frac=111...1 ( $M = 2.0 - \epsilon$ )
  - Get extra leading bit for “free”

# Normalized Encoding Example

$$v = (-1)^s M 2^E$$
$$E = \text{Exp} - \text{Bias}$$

■ Value: float  $F = 15213.0$ ;

$$15213_{10} = 11101101101101_2$$
$$= 1.1101101101101_2 \times 2^{13}$$

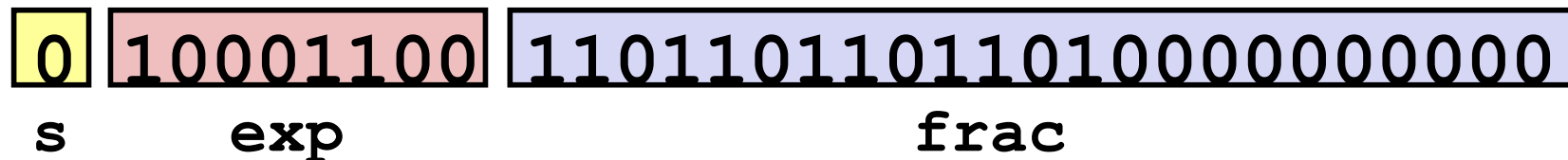
■ Significand

$$M = 1.\underline{1101101101101}_2$$
$$\text{frac} = \underline{1101101101101}0000000000_2$$

■ Exponent

$$E = 13$$
$$\text{Bias} = 127$$
$$\text{Exp} = 140 = 10001100_2$$

■ Result:





# Denormalized Values

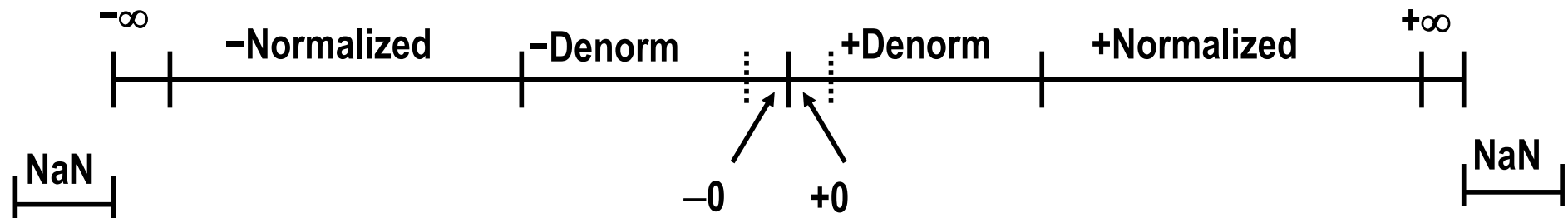
$$v = (-1)^s M 2^E$$
$$E = 1 - \text{Bias}$$

- Condition:  $\text{exp} = 000\dots 0$
- Exponent value:  $E = 1 - \text{Bias}$  (instead of  $E = 0 - \text{Bias}$ )
- Significand coded with implied leading 0:  $M = 0.\text{xxx}\dots\text{x}_2$ 
  - $\text{xxx}\dots\text{x}$ : bits of **frac**
- Cases
  - $\text{exp} = 000\dots 0, \text{frac} = 000\dots 0$ 
    - Represents zero value
    - Note distinct values: +0 and -0 (why?)
  - $\text{exp} = 000\dots 0, \text{frac} \neq 000\dots 0$ 
    - Numbers closest to 0.0
    - Equispaced

# Special Values

- **Condition:  $\text{exp} = 111\dots 1$**
- **Case:  $\text{exp} = 111\dots 1, \text{frac} = 000\dots 0$** 
  - Represents value  $\infty$  (infinity)
  - Operation that overflows
  - Both positive and negative
  - E.g.,  $1.0/0.0 = -1.0/-0.0 = +\infty$ ,  $1.0/-0.0 = -\infty$
- **Case:  $\text{exp} = 111\dots 1, \text{frac} \neq 000\dots 0$** 
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g.,  $\text{sqrt}(-1)$ ,  $\infty - \infty$ ,  $\infty \times 0$

# Visualization: Floating Point Encodings



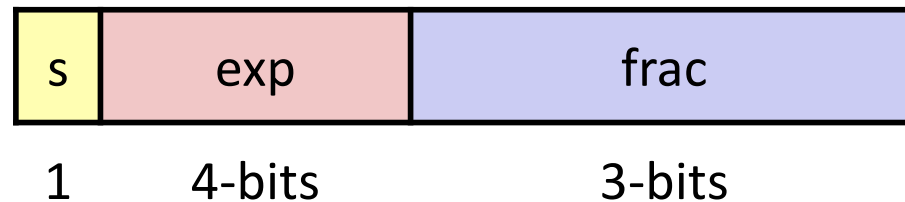
# Today: Integer arithmetic and floating point

- Recap
- Integer arithmetic
- **Floating Points**
  - Background: Fractional binary numbers
  - IEEE floating point standard: Definition
  - Example and properties
  - Rounding, addition, multiplication
  - Floating point in C
  - Summary

# Play the game

- <https://topps.diku.dk/compsys/floating-point.html>

# Tiny Floating Point Example



## ■ 8-bit Floating Point Representation

- the sign bit is in the most significant bit
- the next four bits are the exponent, with a bias of 7
- the last three bits are the **frac**

## ■ Same general form as IEEE Format

- normalized, denormalized
- representation of 0, NaN, infinity

# Dynamic Range (Positive Only)

$$v = (-1)^s M 2^E$$

$$n: E = \text{Exp} - \text{Bias}$$

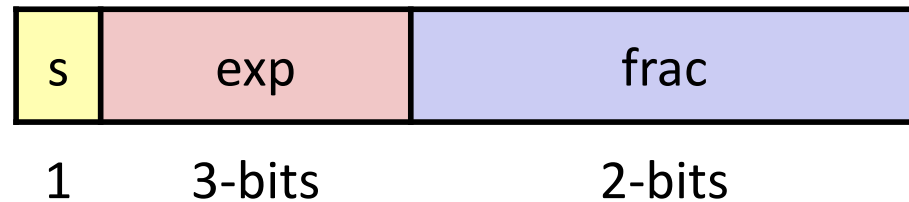
$$d: E = 1 - \text{Bias}$$

	s	exp	frac	E	Value	
Denormalized numbers	0	0000	000	-6	0	closest to zero
	0	0000	001	-6	$1/8 * 1/64 = 1/512$	
	0	0000	010	-6	$2/8 * 1/64 = 2/512$	
	...					
	0	0000	110	-6	$6/8 * 1/64 = 6/512$	
	0	0000	111	-6	$7/8 * 1/64 = 7/512$	
	0	0001	000	-6	$8/8 * 1/64 = 8/512$	largest denorm
Normalized numbers	0	0001	001	-6	$9/8 * 1/64 = 9/512$	smallest norm
	...					
	0	0110	110	-1	$14/8 * 1/2 = 14/16$	
	0	0110	111	-1	$15/8 * 1/2 = 15/16$	closest to 1 below
	0	0111	000	0	$8/8 * 1 = 1$	
	0	0111	001	0	$9/8 * 1 = 9/8$	closest to 1 above
	0	0111	010	0	$10/8 * 1 = 10/8$	
	...					
	0	1110	110	7	$14/8 * 128 = 224$	
	0	1110	111	7	$15/8 * 128 = 240$	largest norm
	0	1111	000	n/a	inf	

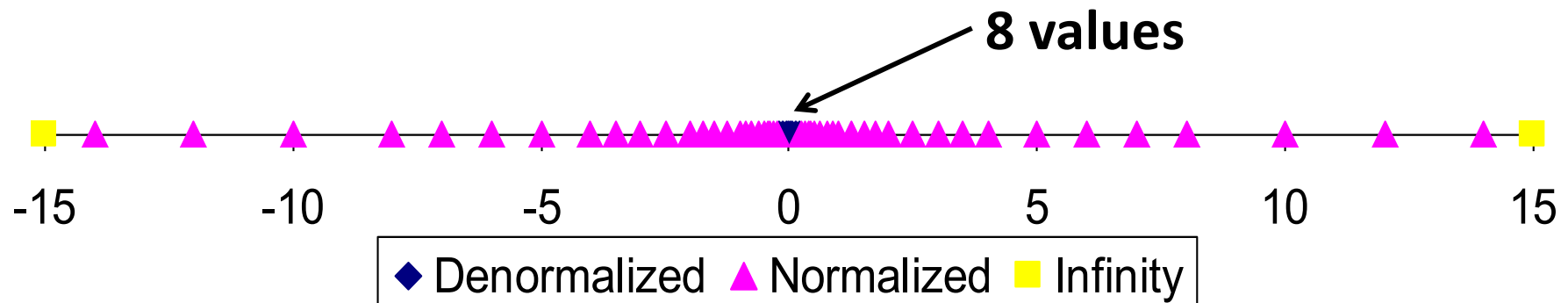
# Distribution of Values

## ■ 6-bit IEEE-like format

- $e = 3$  exponent bits
- $f = 2$  fraction bits
- Bias is  $2^{3-1}-1 = 3$



## ■ Notice how the distribution gets denser toward zero.

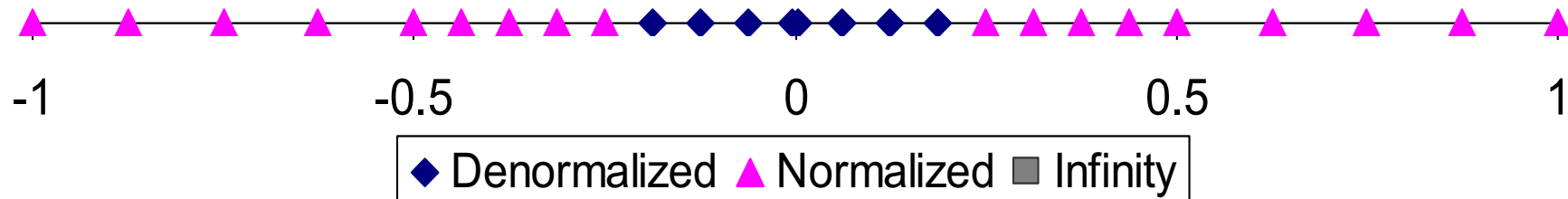
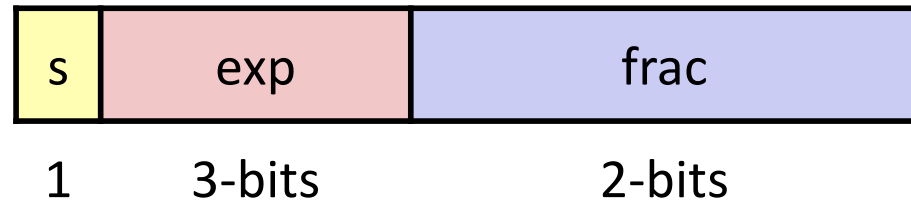




# Distribution of Values (close-up view)

## ■ 6-bit IEEE-like format

- $e = 3$  exponent bits
- $f = 2$  fraction bits
- Bias is 3



# Special Properties of the IEEE Encoding

- **FP Zero Same as Integer Zero**
  - All bits = 0
- **Can (Almost) Use Unsigned Integer Comparison**
  - Must first compare sign bits
  - Must consider  $-0 = 0$
  - NaNs problematic
    - Will be greater than any other values
    - What should comparison yield?
  - Otherwise OK
    - Denorm vs. normalized
    - Normalized vs. infinity

For more details:

[https://www.gnu.org/software/libc/manual/html\\_node/Infinity-and-NaN.html](https://www.gnu.org/software/libc/manual/html_node/Infinity-and-NaN.html)

# Today: Integer arithmetic and floating point

- Recap
- Integer arithmetic
- **Floating Points**
  - Background: Fractional binary numbers
  - IEEE floating point standard: Definition
  - Example and properties
  - Rounding, addition, multiplication
  - Floating point in C
  - Summary

# Floating Point Operations: Basic Idea

- $x +_f y = \text{Round}(x + y)$
- $x \times_f y = \text{Round}(x \times y)$
- **Basic idea**
  - First **compute exact result**
  - Make it fit into desired precision
    - Possibly overflow if exponent too large
    - Possibly **round to fit into frac**

# Rounding

## ■ Rounding Modes (illustrate with \$ rounding)

■	\$1.40	\$1.60	\$1.50	\$2.50	–
<b>\$1.50</b>					
■ Towards zero	\$1	\$1	\$1	\$2	–\$1
■ Round down ( $-\infty$ )	\$1	\$1	\$1	\$2	–\$2
■ Round up ( $+\infty$ )	\$2	\$2	\$2	\$3	–\$1
■ Nearest Even (default)	\$1	\$2	\$2	\$2	–\$2

# Closer Look at Round-To-Even

## ■ Default Rounding Mode

- Hard to get any other kind without dropping into assembly
- All others are statistically biased
  - Sum of set of positive numbers will consistently be over- or under-estimated

## ■ Applying to Other Decimal Places / Bit Positions

- When exactly halfway between two possible values
  - Round so that least significant digit is even
- E.g., round to nearest hundredth

7.8949999	7.89	(Less than half way)
7.8950001	7.90	(Greater than half way)
7.8950000	7.90	(Half way—round up)
7.8850000	7.88	(Half way—round down)

# Rounding Binary Numbers

## ■ Binary Fractional Numbers

- “Even” when least significant bit is 0
- “Half way” when bits to right of rounding position = 100...<sub>2</sub>

## ■ Examples

- Round to nearest 1/4 (2 bits right of binary point)

Value	Binary	Rounded	Action	Rounded
Value				
2 3/32	10.00011 <sub>2</sub>	10.00 <sub>2</sub>	(<1/2—down)	2
2 3/16	10.00110 <sub>2</sub>	10.01 <sub>2</sub>	(>1/2—up)	2 1/4
2 7/8	10.11100 <sub>2</sub>	11.00 <sub>2</sub>	( 1/2—up)	3
2 5/8	10.10100 <sub>2</sub>	10.10 <sub>2</sub>	( 1/2—down)	2 1/2

# FP Multiplication

- $(-1)^{s1} M1 2^{E1} \times (-1)^{s2} M2 2^{E2}$

- **Exact Result:**  $(-1)^s M 2^E$

- Sign  $s$ :  $s1 \wedge s2$
- Significand  $M$ :  $M1 \times M2$
- Exponent  $E$ :  $E1 + E2$

- **Fixing**

- If  $M \geq 2$ , shift  $M$  right, increment  $E$
- If  $E$  out of range, overflow
- Round  $M$  to fit **frac** precision

- **Implementation**

- Biggest chore is multiplying significands



# Floating Point Addition

■  $(-1)^{s1} M1 2^{E1} + (-1)^{s2} M2 2^{E2}$

■ Assume  $E1 > E2$

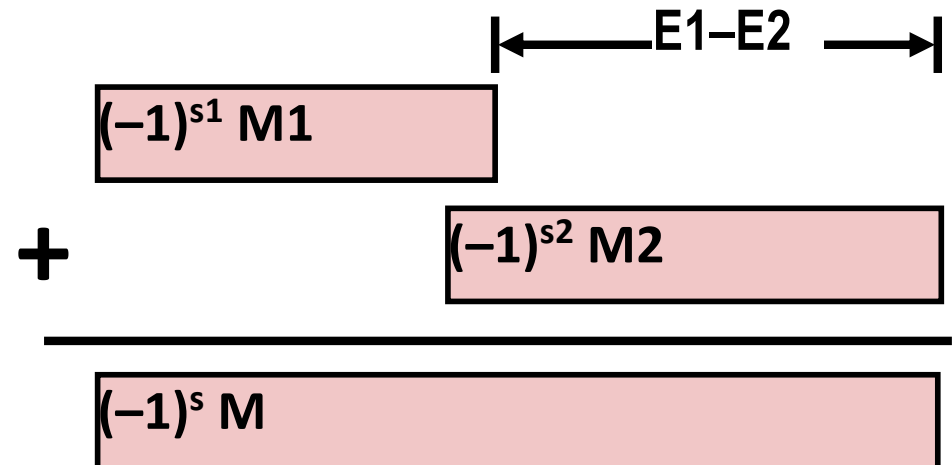
■ **Exact Result:**  $(-1)^s M 2^E$

■ Sign  $s$ , significand  $M$ :

■ Result of signed align & add

■ Exponent  $E$ :  $E1$

Get binary points lined up



## ■ Fixing

■ If  $M \geq 2$ , shift  $M$  right, increment  $E$

■ if  $M < 1$ , shift  $M$  left  $k$  positions, decrement  $E$  by  $k$

■ Overflow if  $E$  out of range

■ Round  $M$  to fit **frac** precision

# Mathematical Properties of FP Add

## ■ Compare to those of Abelian Group

- Closed under addition? **Yes**
  - But may generate infinity or NaN
- Commutative? **Yes**
- Associative? **No**
  - Overflow and inexactness of rounding
  - $(3.14 + 1e10) - 1e10 = 0$ ,  $3.14 + (1e10 - 1e10) = 3.14$
- 0 is additive identity? **Yes**
- Every element has additive inverse? **Almost**
  - Yes, except for infinities & NaNs

## ■ Monotonicity

- $a \geq b \Rightarrow a + c \geq b + c$  **Almost**
  - Except for infinities & NaNs

# Mathematical Properties of FP Mult

## ■ Compare to Commutative Ring

- Closed under multiplication? **Yes**
  - But may generate infinity or NaN
- Multiplication Commutative? **Yes**
- Multiplication is Associative? **No**
  - Possibility of overflow, inexactness of rounding
  - Ex:  $(1e20 * 1e20) * 1e-20 = \text{inf}$ ,  $1e20 * (1e20 * 1e-20) = 1e20$
- 1 is multiplicative identity? **Yes**
- Multiplication distributes over addition? **No**
  - Possibility of overflow, inexactness of rounding
  - $1e20 * (1e20 - 1e20) = 0.0$ ,  $1e20 * 1e20 - 1e20 * 1e20 = \text{NaN}$

## ■ Monotonicity

- $a \geq b \ \& \ c \geq 0 \Rightarrow a * c \geq b * c$ ?

**Almost**

# Today: Integer arithmetic and floating point

- Recap
- Integer arithmetic
- **Floating Points**
  - Background: Fractional binary numbers
  - IEEE floating point standard: Definition
  - Example and properties
  - Rounding, addition, multiplication
  - Floating point in C
  - Summary

# Floating Point in C

## ■ C Guarantees Two Levels

- **float**      single precision
- **double**     double precision

## ■ Conversions/Casting

- Casting between **int**, **float**, and **double** changes bit representation
- **double/float** → **int**
  - Truncates fractional part
  - Like rounding toward zero
  - Not defined when out of range or NaN: Generally sets to TMin
- **int** → **double**
  - Exact conversion, as long as **int** has  $\leq 53$  bit word size
- **int** → **float**
  - Will round according to rounding mode

# Floating Point Puzzles

## ■ For each of the following C expressions, either:

- Argue that it is true for all argument values
- Explain why not true

```
int x = ...;  
float f = ...;  
double d = ...;
```

Assume neither  
d nor f is NaN

- `x == (int)(float) x`
- `x == (int)(double) x`
- `f == (float)(double) f`
- `d == (double)(float) d`
- `f == -(-f);`
- `2/3 == 2/3.0`
- `d < 0.0`  $\Rightarrow$  `((d*2) < 0.0)`
- `d > f`  $\Rightarrow$  `-f > -d`
- `d * d >= 0.0`
- `(d+f)-d == f`

# Today: Integer arithmetic and floating point

- Recap
- Integer arithmetic
- **Floating Points**
  - Background: Fractional binary numbers
  - IEEE floating point standard: Definition
  - Example and properties
  - Rounding, addition, multiplication
  - Floating point in C
  - Summary

# Summary

- IEEE Floating Point has clear mathematical properties
- Represents numbers of form  $M \times 2^E$
- One can reason about operations independent of implementation
  - As if computed with perfect precision and then rounded
- **Not the same as real arithmetic**
  - Violates associativity/distributivity
  - Makes life difficult for compilers & serious numerical applications programmers



# Interesting Numbers

{single, double}

<i>Description</i>	<i>exp</i>	<i>frac</i>	<i>Numeric Value</i>
■ Zero	00...00	00...00	0.0
■ Smallest Pos. Denorm.	00...00	00...01	$2^{-\{23,52\}} \times 2^{-\{126,1022\}}$
■ Single $\approx 1.4 \times 10^{-45}$			
■ Double $\approx 4.9 \times 10^{-324}$			
■ Largest Denormalized	00...00	11...11	$(1.0 - \epsilon) \times 2^{-\{126,1022\}}$
■ Single $\approx 1.18 \times 10^{-38}$			
■ Double $\approx 2.2 \times 10^{-308}$			
■ Smallest Pos. Normalized	00...01	00...00	$1.0 \times 2^{-\{126,1022\}}$
■ Just larger than largest denormalized			
■ One	01...11	00...00	1.0
■ Largest Normalized	11...10	11...11	$(2.0 - \epsilon) \times 2^{\{127,1023\}}$
■ Single $\approx 3.4 \times 10^{38}$			
■ Double $\approx 1.8 \times 10^{308}$			