

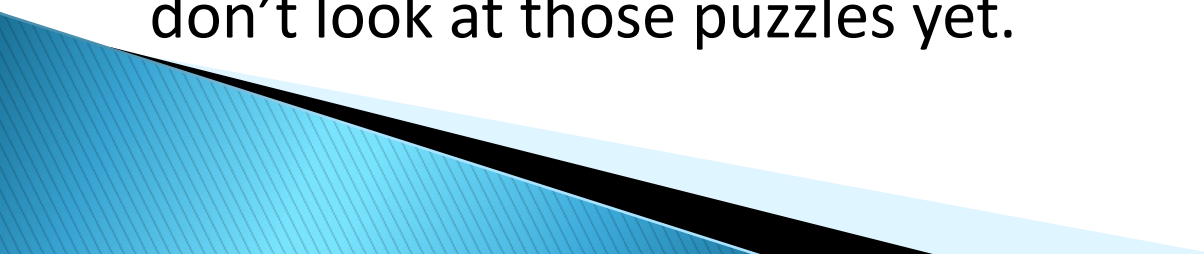
Computer Data Representation (3): Lecture #4 – Part 1

Prof. Soraya Abad-Mota, PhD

Today

- ▶ Part 1: Complete the integral types computer representation:
 - Arithmetic operations on integers
 - Why use unsigned?
- ▶ Part 2: Floating point representation (begin)

NOTE: the datalab project will be posted Friday 8/30/24 after the lab, it contains some puzzles with floating point, but we finish covering FP on 9/4/24 after Labor Day holiday, so please don't look at those puzzles yet.



Arithmetic Operators on Integers

In lecture we cover:

1. Addition (Unsigned and 2's complement): not the derivations of the principles, suggest specific practice exercises to do as you read + example of negative overflow
2. Multiplication (general idea + practice exercises for you to do)
3. Power of 2 (guide to study)
4. Why use unsigned?
5. Slides for everything we don't cover in detail



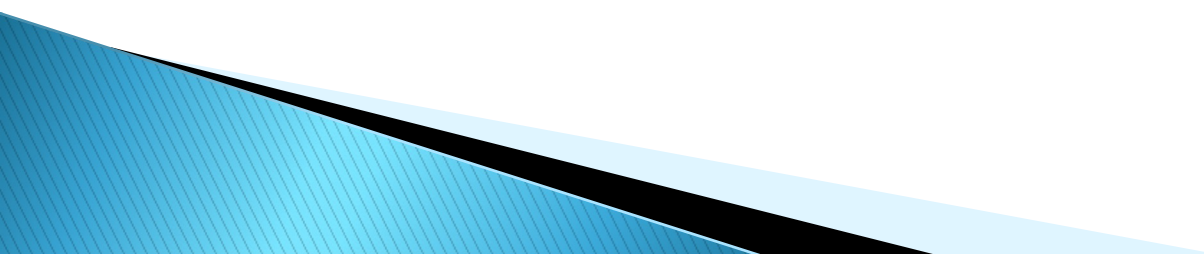
Bits, Bytes, and Integers

▶ Integers

- Representation: unsigned and signed (CS241L)
- Conversion, casting
- Expanding, truncating
- **Addition, negation, multiplication, shifting**

▶ Representations in memory, pointers, strings

▶ Summary



Arithmetic: Basic Rules

- ▶ Most PL support “fixed-size arithmetic” (\sim integers)
- ▶ 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 (on overflow)
 - 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)

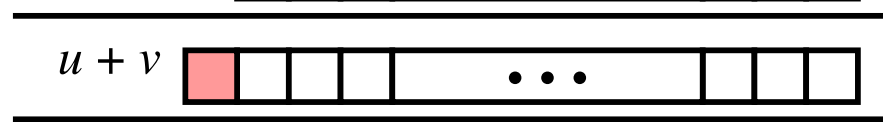


1 A. Unsigned Addition

Operands: w bits



True Sum: $w+1$ bits



Discard Carry: w bits



- ▶ Standard Addition Function

- Ignores carry output

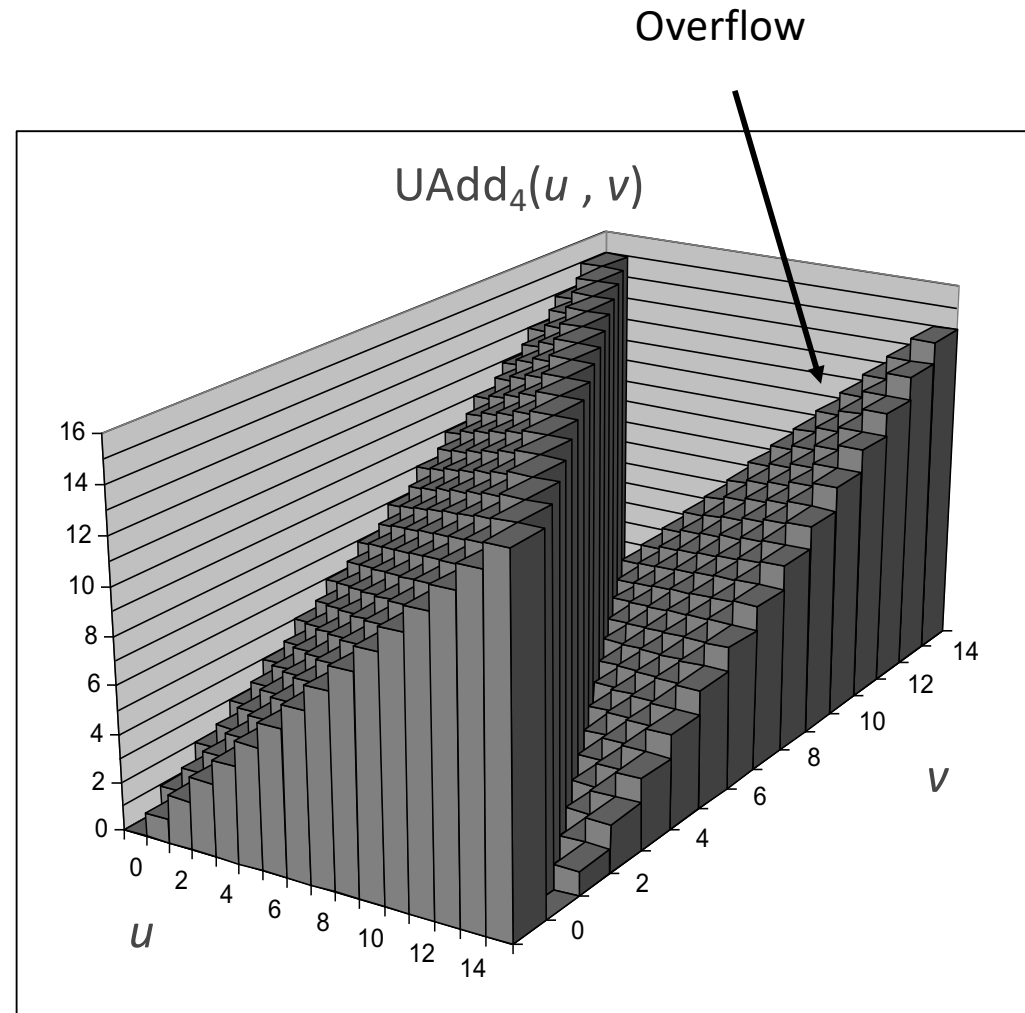
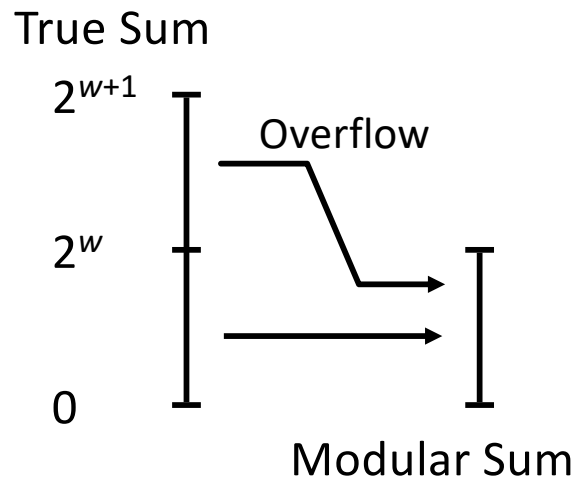
- ▶ Implements Modular Arithmetic

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

- *Detecting overflow occurred: $s < u$ or $s < v$*

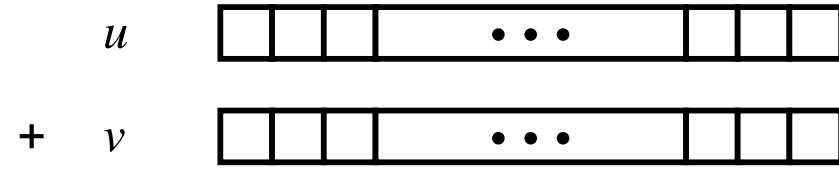
Visualizing Unsigned Addition

- ▶ Wraps Around
 - If true sum $\geq 2^w$
 - At most once
- ▶ If $w=4$, $9 + 12 = 21$
 - $21 \bmod 16 = 5$

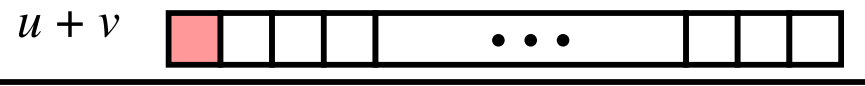


1 B. Two's Complement Addition

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 $s == t$

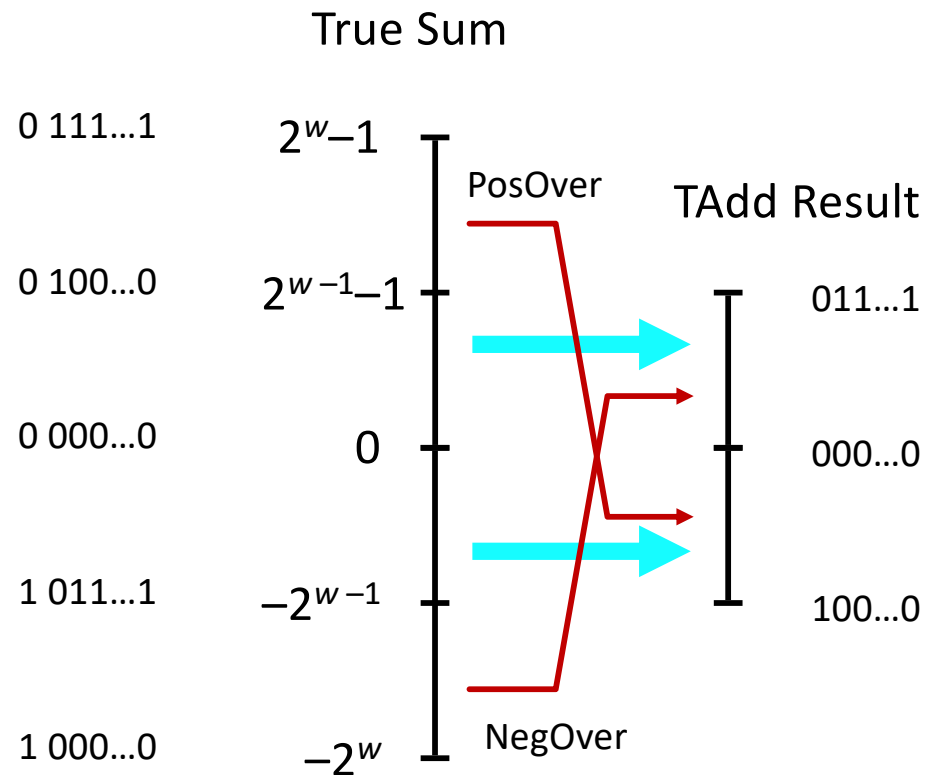
TAdd Overflow

▶ Functionality

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

▶ Detect overflow

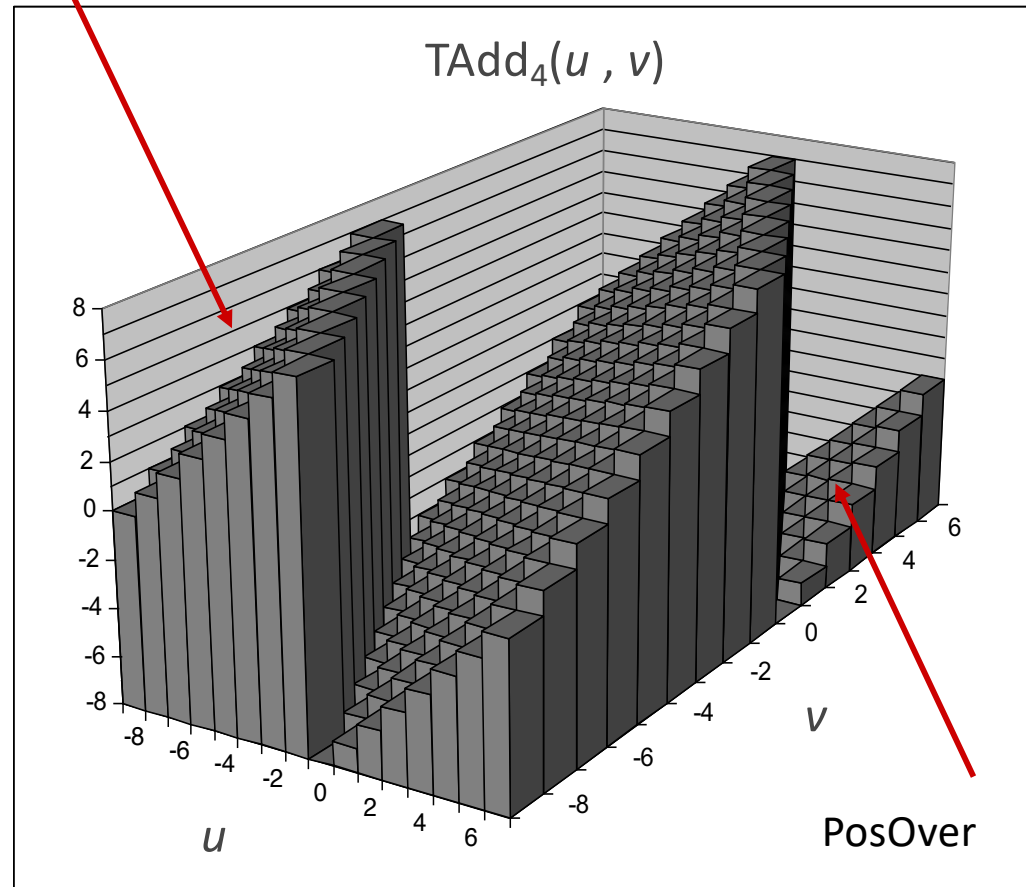
- 2 pos, sum neg
- 2 neg, sum pos



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

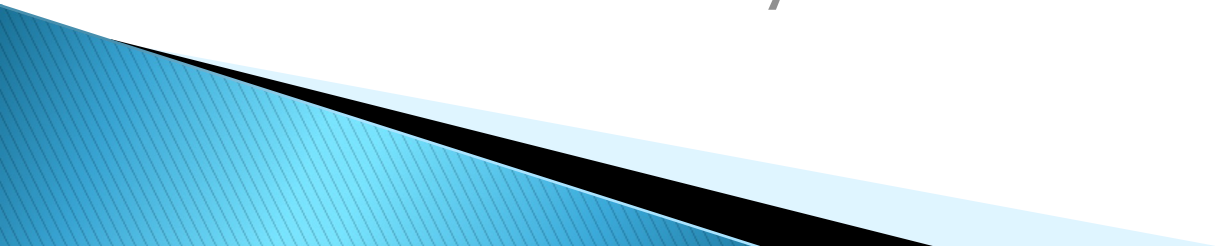
NegOver



Study and Practice (on your own)

- ▶ In all arithmetic operations, for unsigned and signed, there are principles used to compute the operations and their derivation. Follow and study the derivations on your own. (Derivations start p. 72 (2.5) to p. 106)
- ▶ Problem 2.29, p. 93 (understand 2's comp +)
- ▶ Problem 2.33, p. 95

Answers are at the end of chapter, but do them first then check you answers with the book



Example: Problem 2.29, first case

▶ $x = 10100 =_{10} -12$

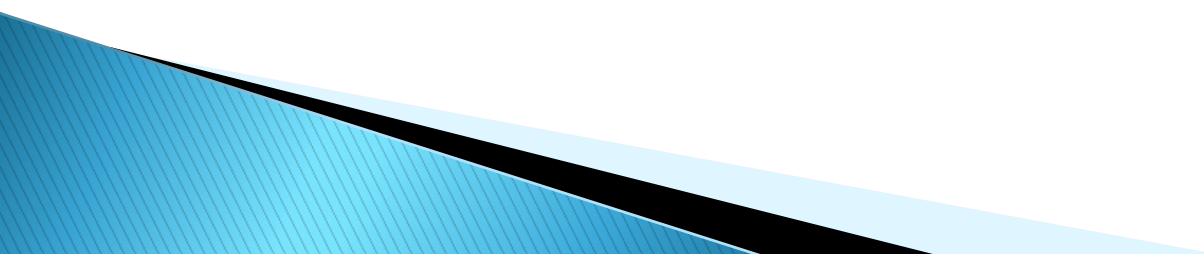
▶ $y = 10001 =_{10} -15$

$x+y = (1) 00101 =_{10} 5$ (negative overflow)

^ this 1 is ignored

p. 90 (answer p. 152)

Principle 2.13, pp.90–91, provides 4 cases
(positive overflow, two normal cases, and negative overflow)



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

2A. Unsigned Multiplication in C

Operands: w bits

u 

$*$ v 

True Product: $2*w$ bits

$u \cdot v$ 

Discard w bits: w bits

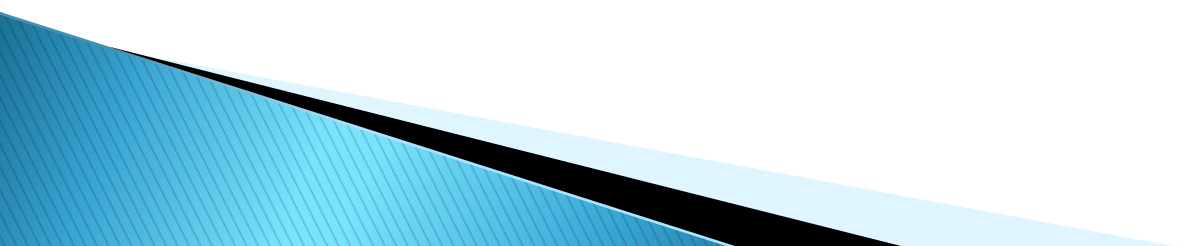
$\text{UMult}_w(u, v)$ 

- ▶ Standard Multiplication Function

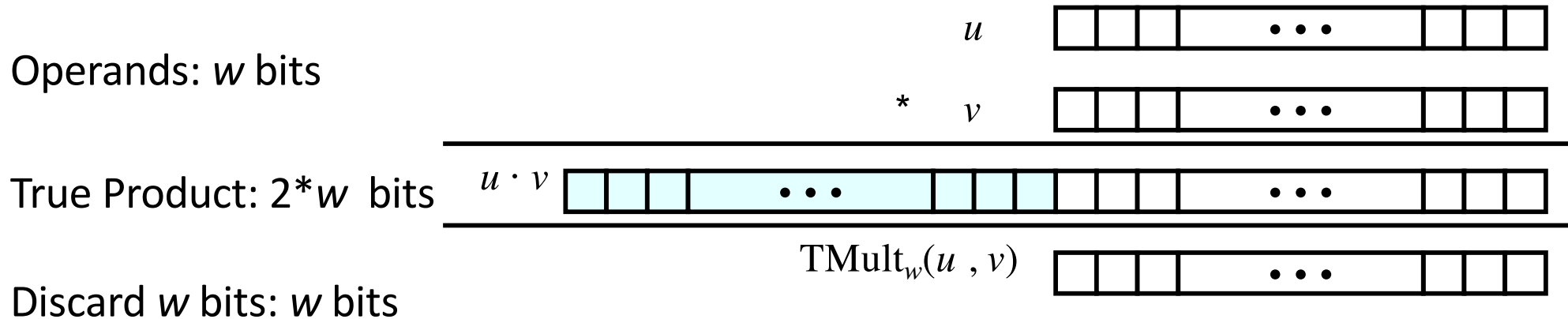
- Ignores high order w bits

- ▶ Implements Modular Arithmetic

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



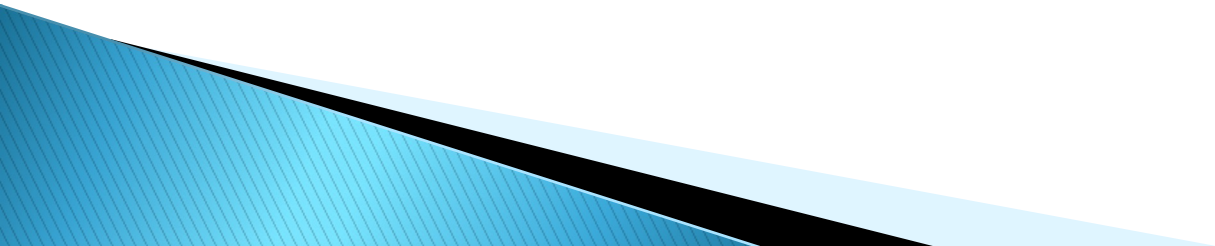
2B. Signed (2's comp.) Multiplication in C



- ▶ Standard Multiplication Function
 - Ignores high order w bits
 - Some of which are different for two's complement vs. unsigned multiplication
 - Lower bits are the same
 - first compute $(x * y) \bmod 2^w$
 - then convert unsigned to two's complement

Practice

- ▶ Problem 2.34, p. 98
- ▶ Problem 2.35, p. 99

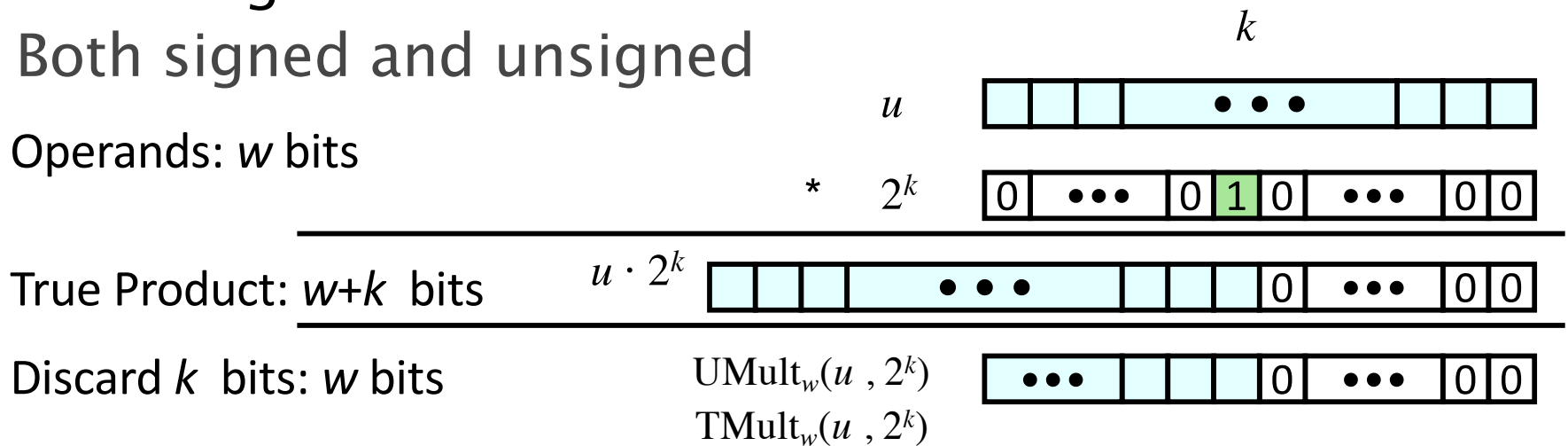


3. Power-of-2 Multiply with Shift

▶ Operation ($k \geq 0$)

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

Operands: w bits



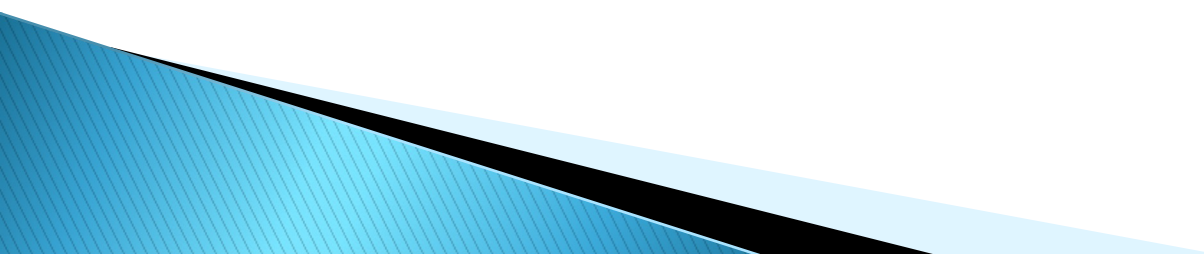
▶ Examples

- $u \ll 3 \quad == \quad u * 8$
- $(u \ll 5) - (u \ll 3) \quad == \quad u * 24$

- ▶ **Motivation:** Most machines shift and add faster than multiply. (Compiler generates this code automatically.)

See examples + Practice

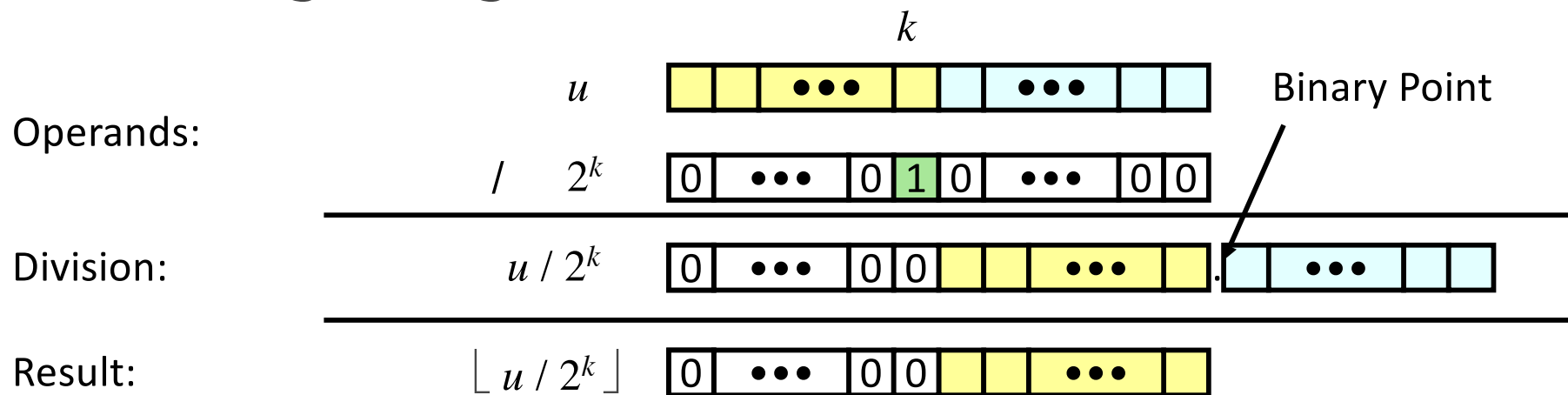
- ▶ Section 2.3.6:
 - explains the principle behind multiplying by a power of 2
 - valid for unsigned and signed
 - procedure to substitute * with shift and additions
 - generalization of procedure of multiplying by constant (form A and B, pp. 102–103)
- ▶ Problems 2.39, 2.40, 2.41, p. 103



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 right shift



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

4. Why Should I Use Unsigned?

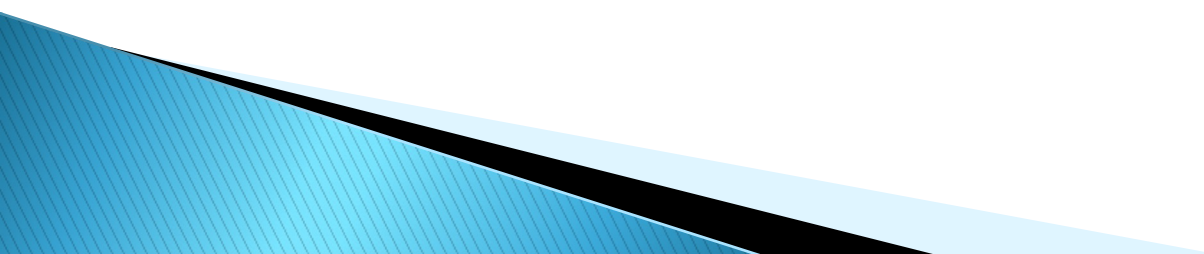
- ▶ *Don't* use without understanding implications

- 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)  
    . . .
```

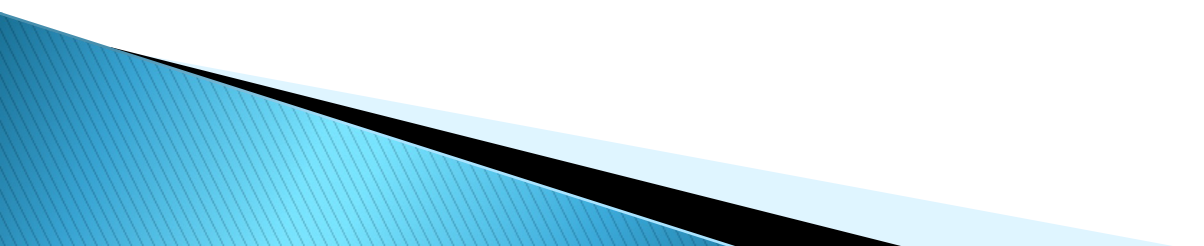


Counting Down with Unsigned (1)

- ▶ 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
 - $0 - 1 \rightarrow UMax$



Counting Down with Unsigned (2)

- ▶ Proper way to use unsigned as loop index

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

- ▶ Even better

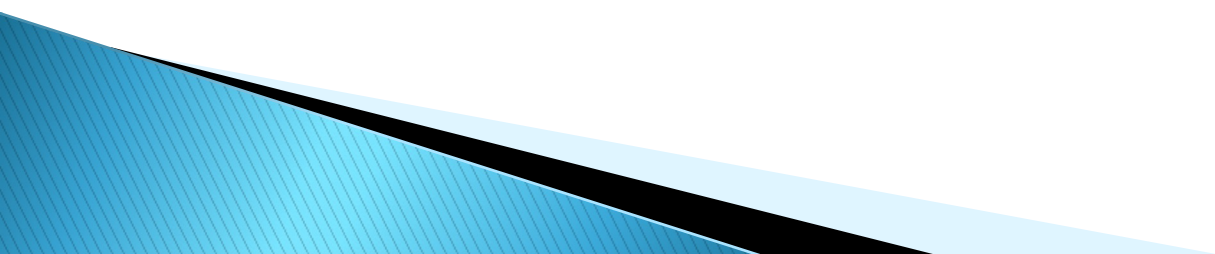
```
size_t i;  
for (i = cnt-2; i < cnt; i--)  
    a[i] += a[i+1];
```

- 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`?



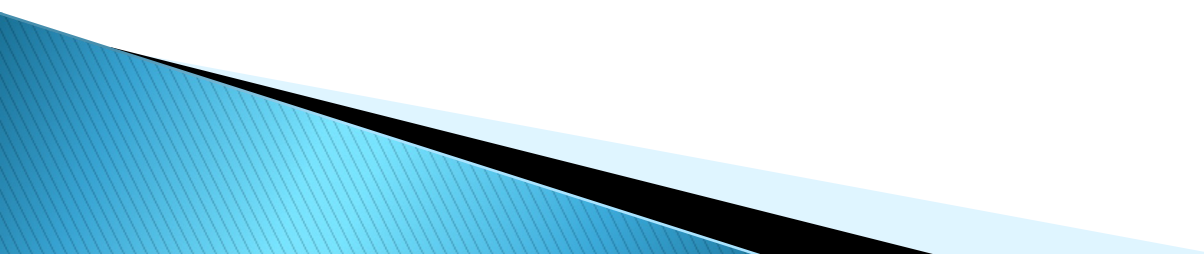
Why Should I Use Unsigned? (cont.)

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



Next three slides (on your own)

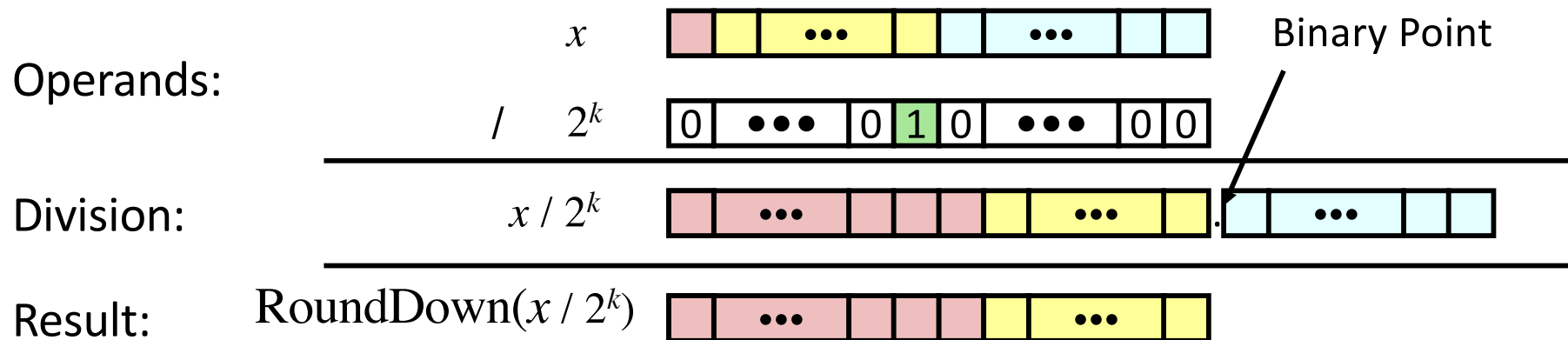
- ▶ Cover power of 2 divide operation with shift
- ▶ More complex than unsigned
- ▶ Have to be careful with the rounding of negative results.



Signed Power-of-2 Divide with Shift

▶ Quotient of Signed by Power of 2

- $x \gg k$ gives $\lfloor x / 2^k \rfloor$
- Uses arithmetic right shift
- ok when $x > 0$, rounds wrong direction when $x < 0$



	Division	Computed	Hex	Binary
x	-15213	-15213	C4 93	11000100 10010011
x >> 1	-7606.5	-7607	E2 49	11100010 01001001
x >> 4	-950.8125	-951	FC 49	11111100 01001001
x >> 8	-59.4257813	-60	FF C4	11111111 11000100

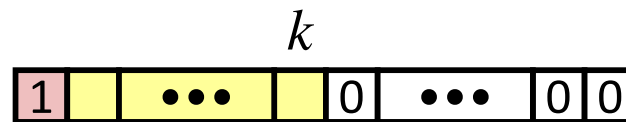
Correct Power-of-2 Divide

- ▶ Quotient of Negative Number by Power of 2
 - Want $\lceil x / 2^k \rceil$ (Round Toward 0)
 - Compute as $\lfloor (x+2^k-1) / 2^k \rfloor$
 - In C: $(x + (1 \ll k) - 1) \gg k$
 - Biases dividend toward 0

Case 1: No rounding

Dividend:

u

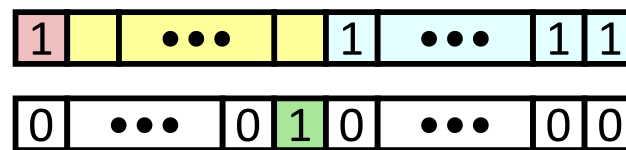


$+2^k - 1$



Divisor:

$/ \quad 2^k$



Binary Point

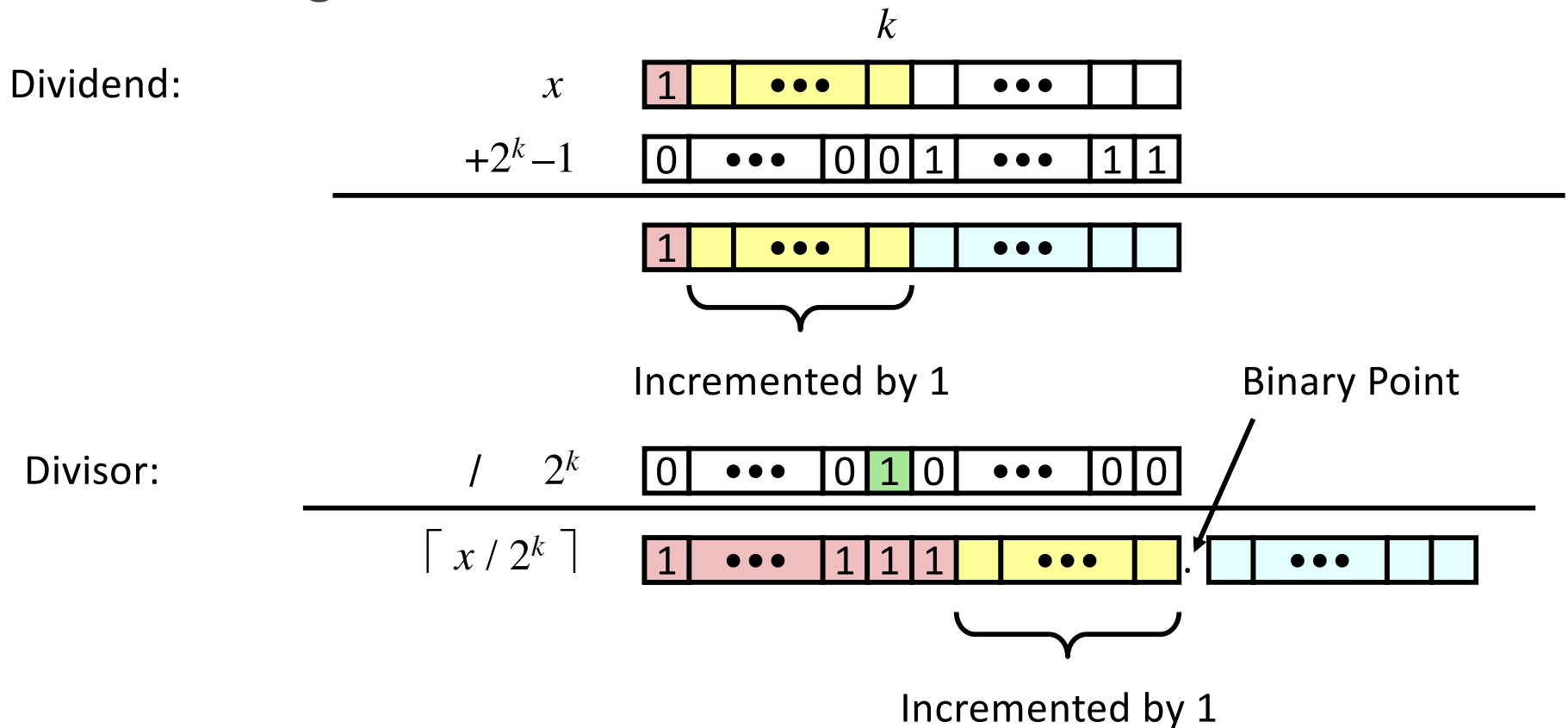
$\lceil u / 2^k \rceil$



Biasing has no effect

Correct Power-of-2 Divide (Cont.)

Case 2: Rounding



Biasing adds 1 to final result

Stop here lecture #4 part 1

