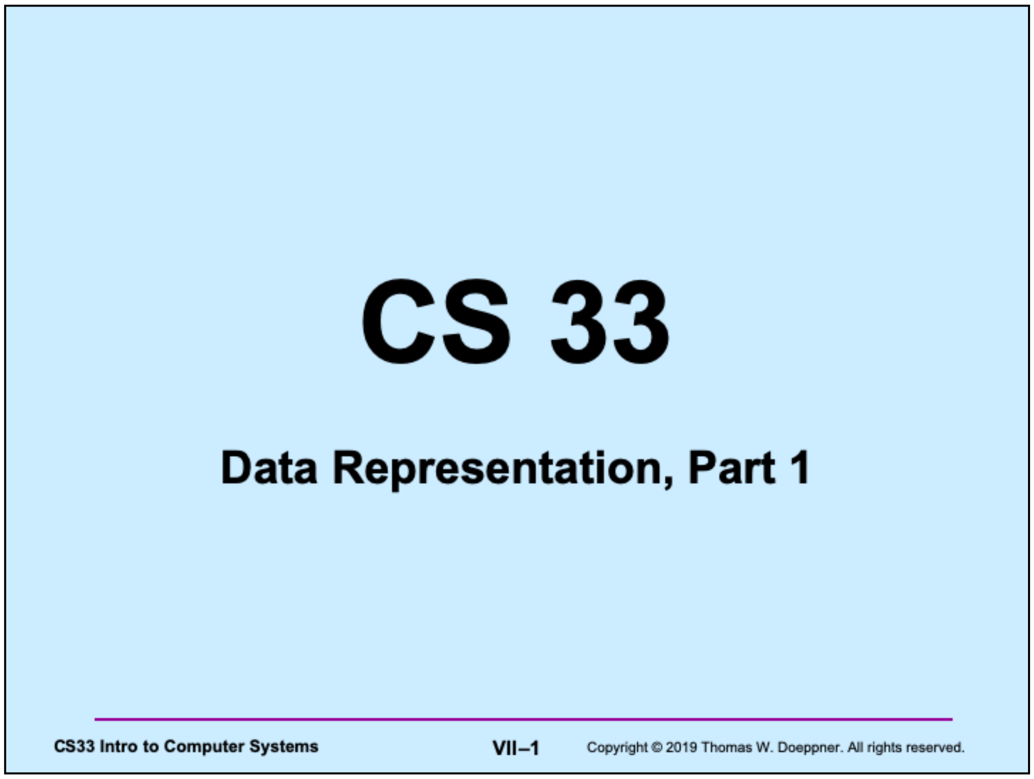
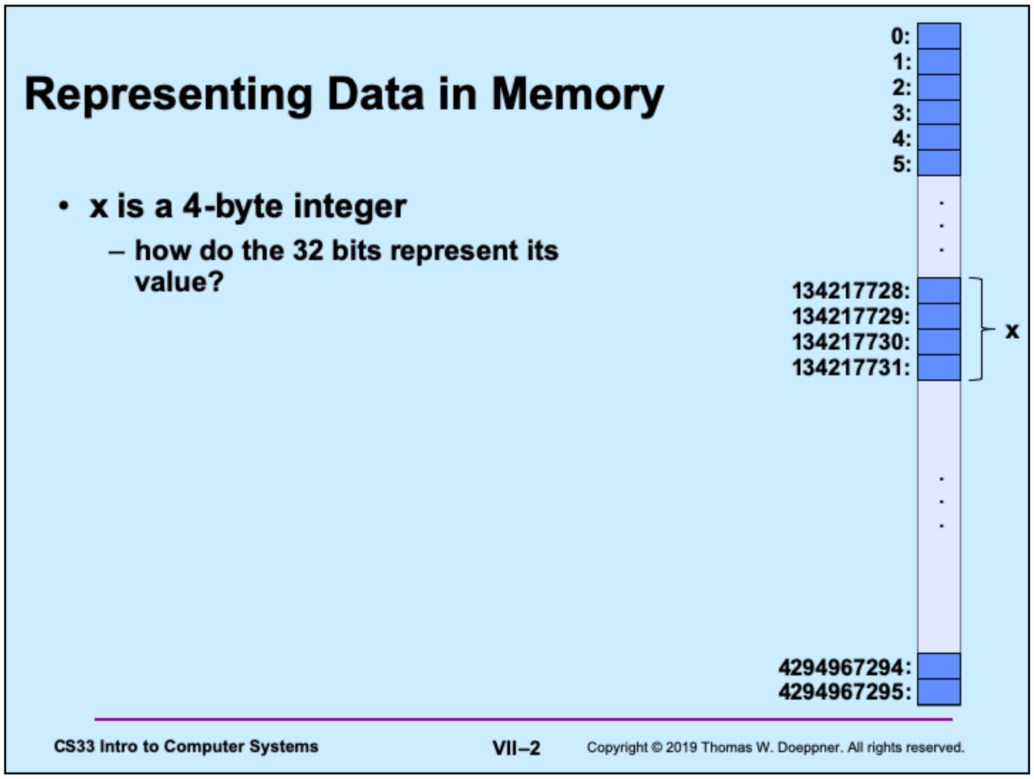


CS 33

Data Representation, Part 1

Many of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook "Computer Systems: A Programmer's Perspective." 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O'Hallaron in Fall 2010. These slides are indicated "Supplied by CMU" in the notes section of the slides.



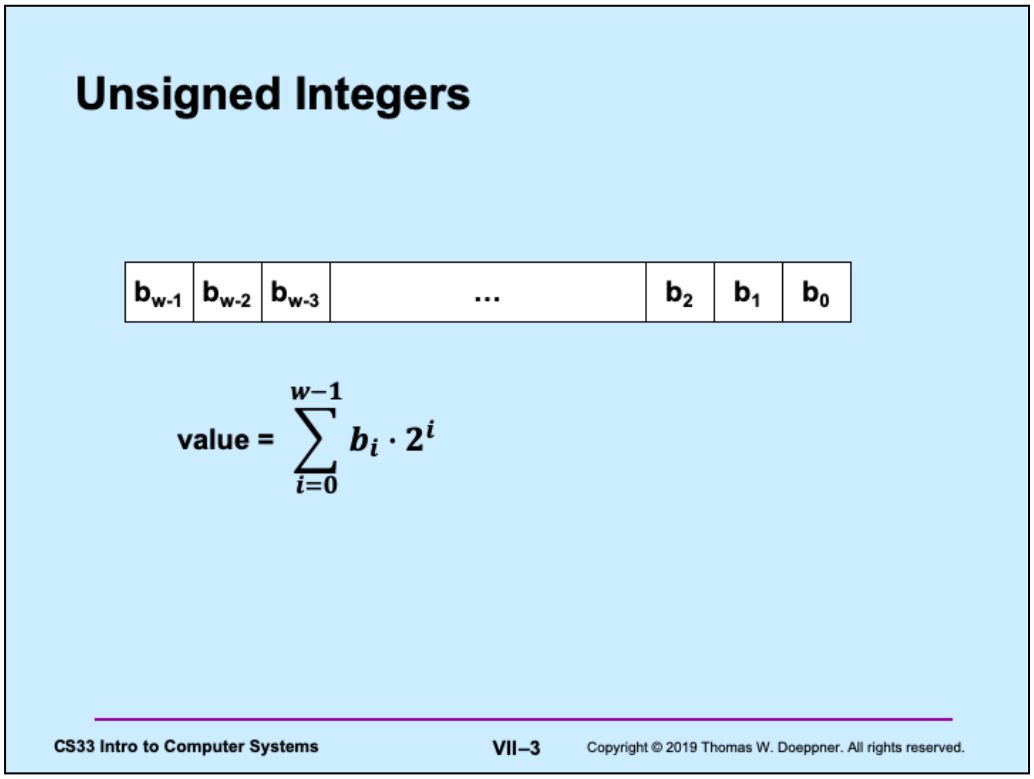
Representing Data in Memory

- x is a 4-byte integer
  - how do the 32 bits represent its value?

0:
1:
2:
3:
4:
5:
134217728:
134217729:
134217730:
134217731:
x
4294967294:
4294967295:

CS33 Intro to Computer Systems VII–2 Copyright © 2019 Thomas W. Doeppner. All rights reserved.

In the diagram, x is an int occupying bytes 134217728, 134217729, 134217730, and 134217731. Its address is 134217728; its size is 4 (bytes).

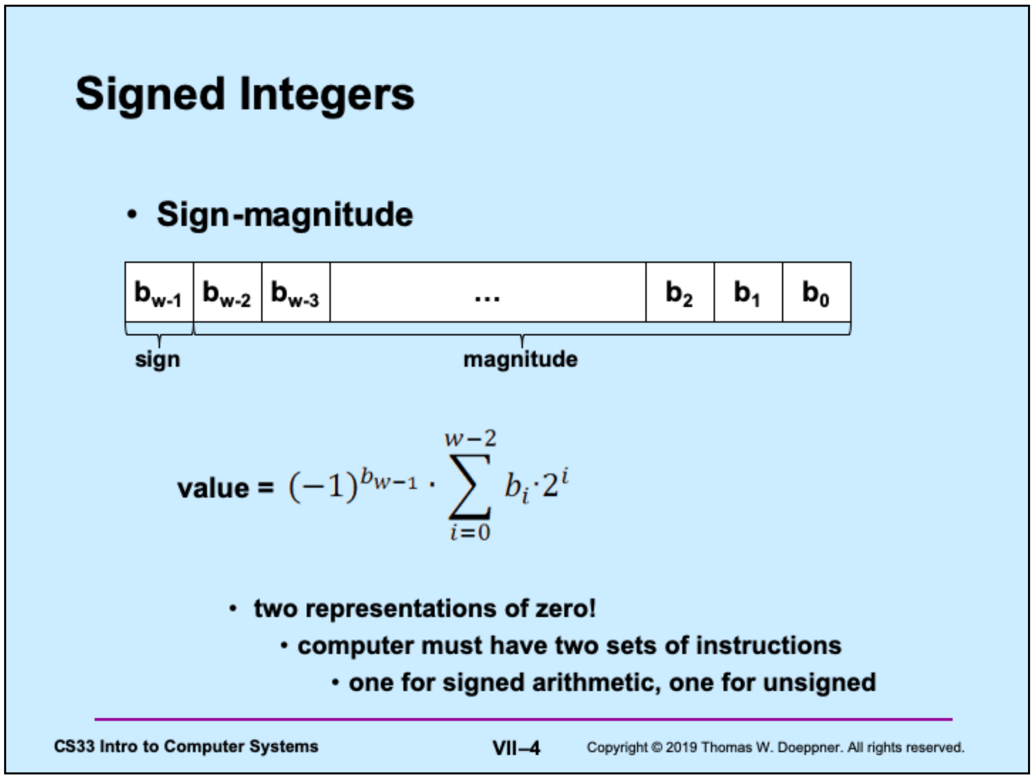


Unsigned Integers

b_{w-1}	b_{w-2}	b_{w-3}	…	b_2	b_1	b_0

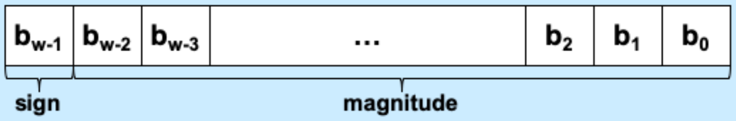
$$\text{value} = \sum_{i=0}^{w-1} b_i \cdot 2^i$$

If a computer word is to be interpreted as an unsigned integer, we can do so as shown in the slide, where w is the number of bits in the word.



Signed Integers

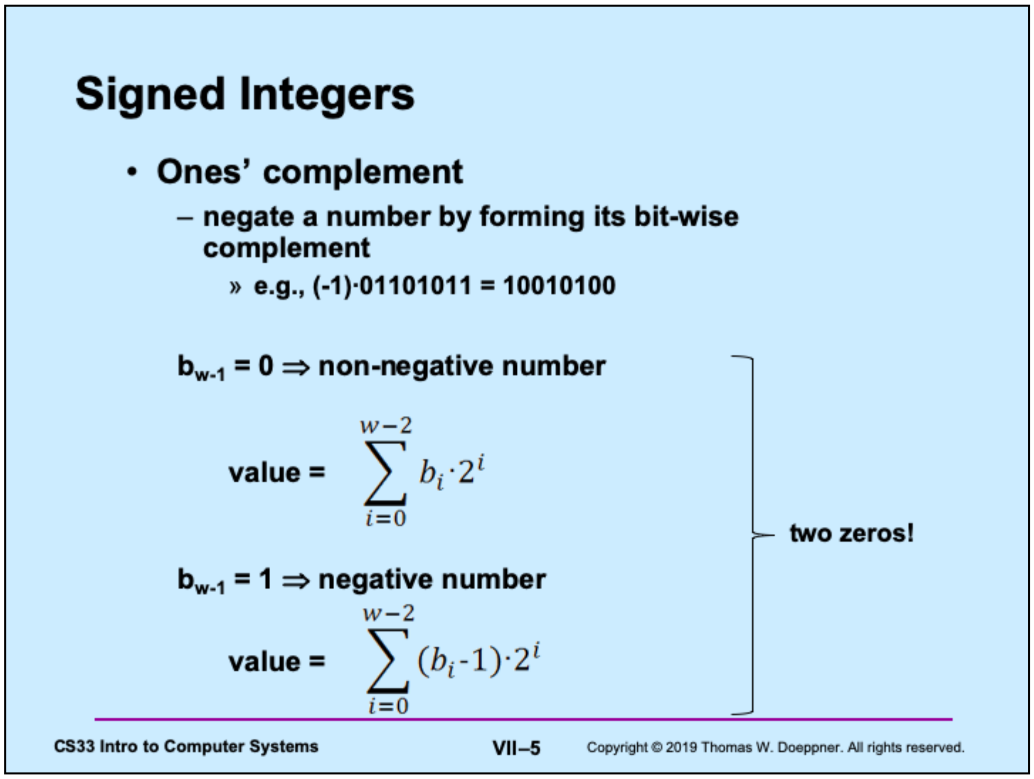
- **Sign-magnitude**



$$\textbf{value} = (-1)^{b_{w-1}} \cdot \sum_{i=0}^{w-2} b_i \cdot 2^i$$

- **two representations of zero!**
  - **computer must have two sets of instructions**
    - **one for signed arithmetic, one for unsigned**

We might also want to interpret the contents of a computer word as a signed integer. There are a few options for how to do this. One straightforward approach is shown in the slide, where we use the high-order (leftmost) bit as the "sign bit": 0 means positive and 1 means negative. However, this has the somewhat weird result that there are two representations of zero. This further means that the computer would have to have two implementations of arithmetic instructions: one for signed arithmetic, the other for unsigned arithmetic.



Signed Integers

- **Ones' complement**
  - **negate a number by forming its bit-wise complement**
    - » **e.g., (-1)·01101011 = 10010100**

$b_{w-1} = 0 \Rightarrow$ non-negative number

$$\text{value} = \sum_{i=0}^{w-2} b_i \cdot 2^i$$

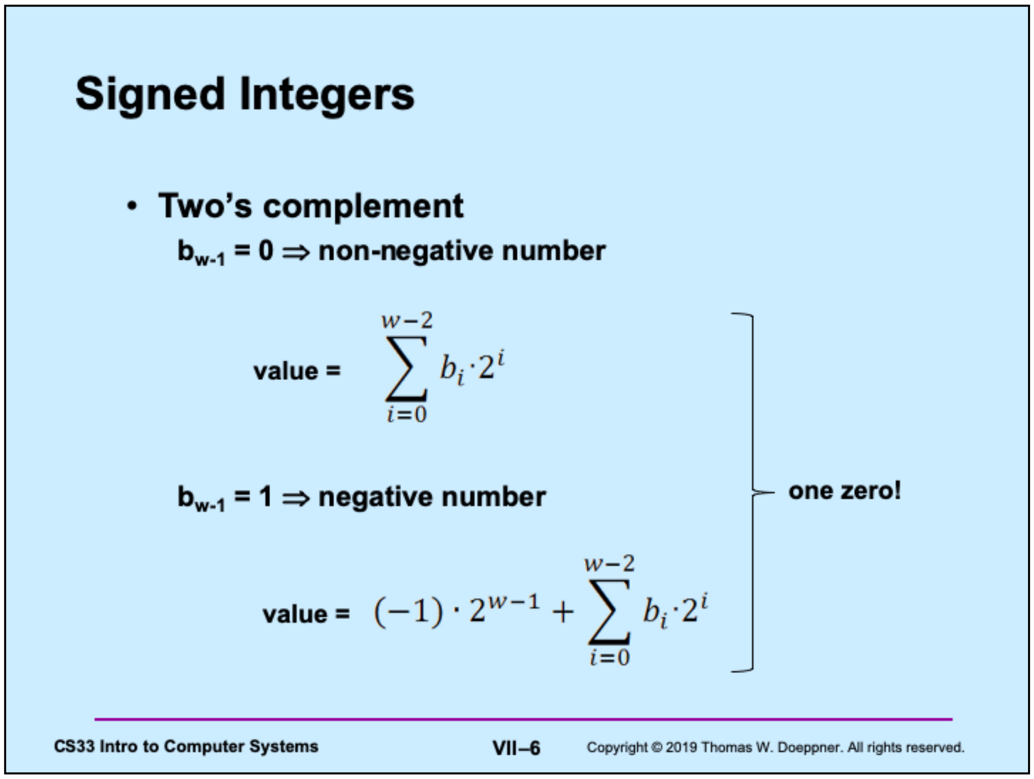
$b_{w-1} = 1 \Rightarrow$ negative number

$$\text{value} = \sum_{i=0}^{w-2} (b_i - 1) \cdot 2^i$$

two zeros!

In ones' complement, a number is positive if its leftmost bit is zero negative otherwise. We negate a number by complementing *all* its bits. Thus if the leftmost bit is zero, a one in position *i* of the remaining bits contributes a value of 2^i and a zero contributes nothing. But if the leftmost bit is one, a zero in position *i* contributes a value of -2^i and a one contributes nothing.

Note that the most-significant bit serves as the sign bit. But, as with sign-magnitude, the computer would need two sets of instructions: one for signed arithmetic and one for unsigned.



Signed Integers

- **Two's complement**

  $b_{w-1} = 0 \Rightarrow$ non-negative number

$$\text{value} = \sum_{i=0}^{w-2} b_i \cdot 2^i$$

  $b_{w-1} = 1 \Rightarrow$ negative number

$$\text{value} = (-1) \cdot 2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i$$

There's only one zero!

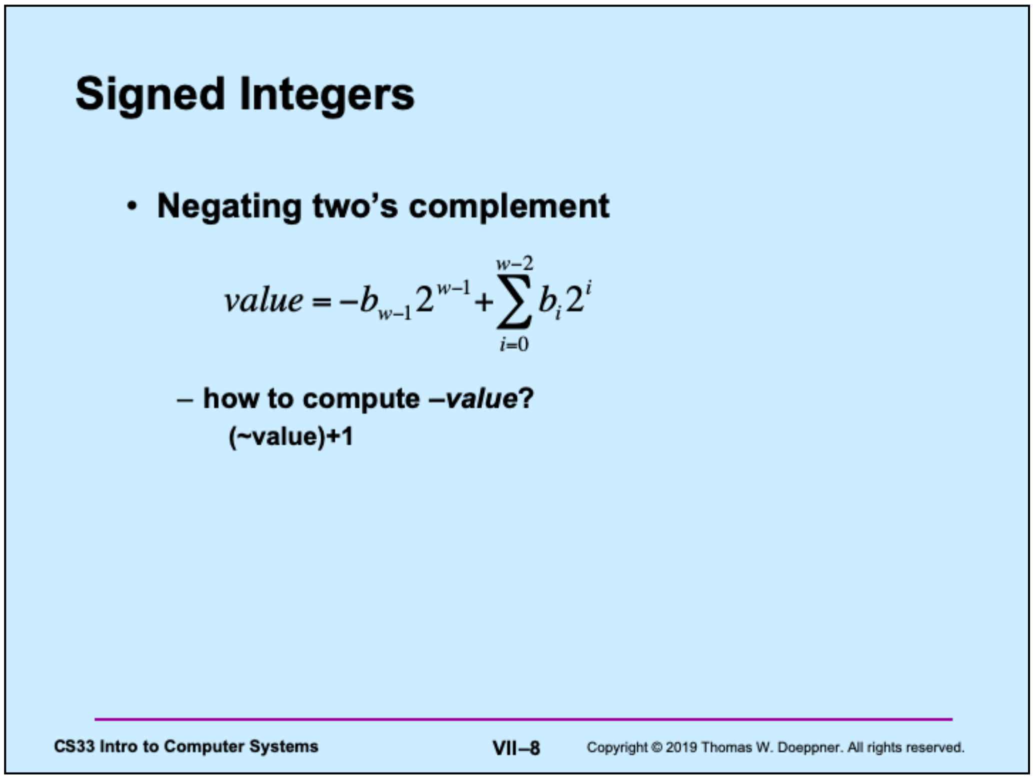
Two's complement is used on pretty much all of today's computers to represent signed integers.

Note that the high-order (most-significant) bit represents -2^{w-1}. All the other bits represent positive numbers.

Example

- w = 4

0000: 0	1000: -8
0001: 1	1001: -7
0010: 2	1010: -6
0011: 3	1011: -5
0100: 4	1100: -4
0101: 5	1101: -3
0110: 6	1110: -2
0111: 7	1111: -1



Signed Integers

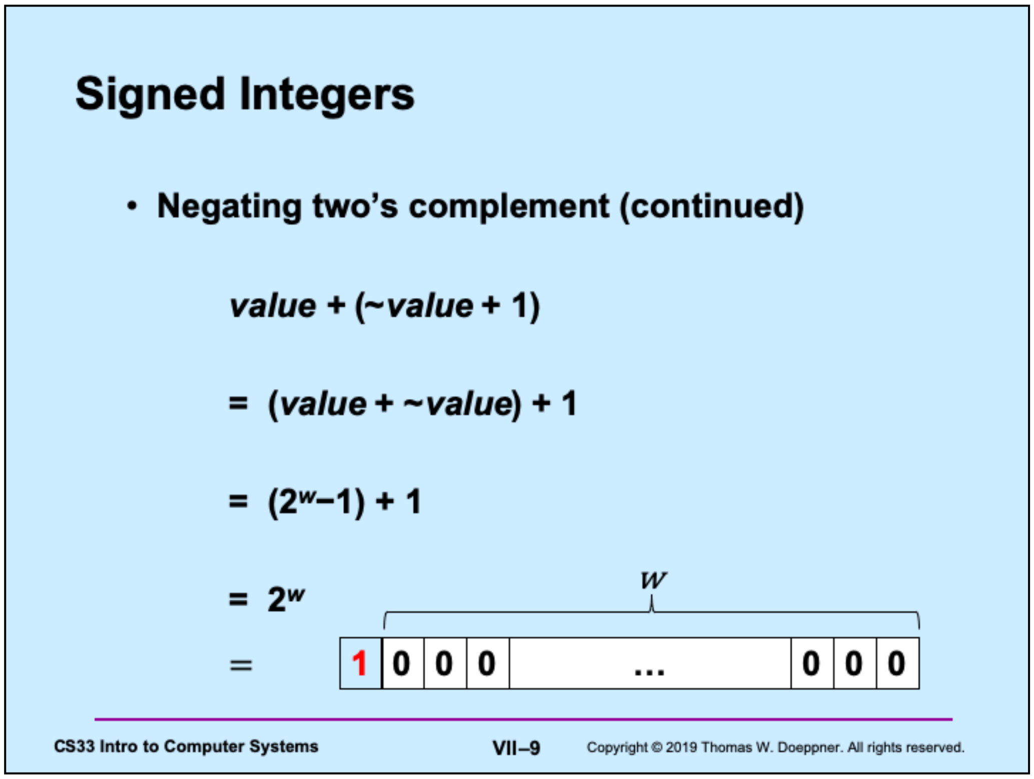
- **Negating two's complement**

$$value = -b_{w-1}2^{w-1} + \sum_{i=0}^{w-2} b_i 2^i$$

  - **how to compute *–value*?**

    (~value)+1

To negate a two's-complement number, simply complement each of its bits, then add one to the result. We show why this works in the next slide.



Signed Integers

- **Negating two's complement (continued)**

$$\mathbf{\mathit{value} + (\sim\mathit{value} + 1)}$$

$$= (\mathit{value} + \sim\mathit{value}) + 1$$

$$= (2^w - 1) + 1$$

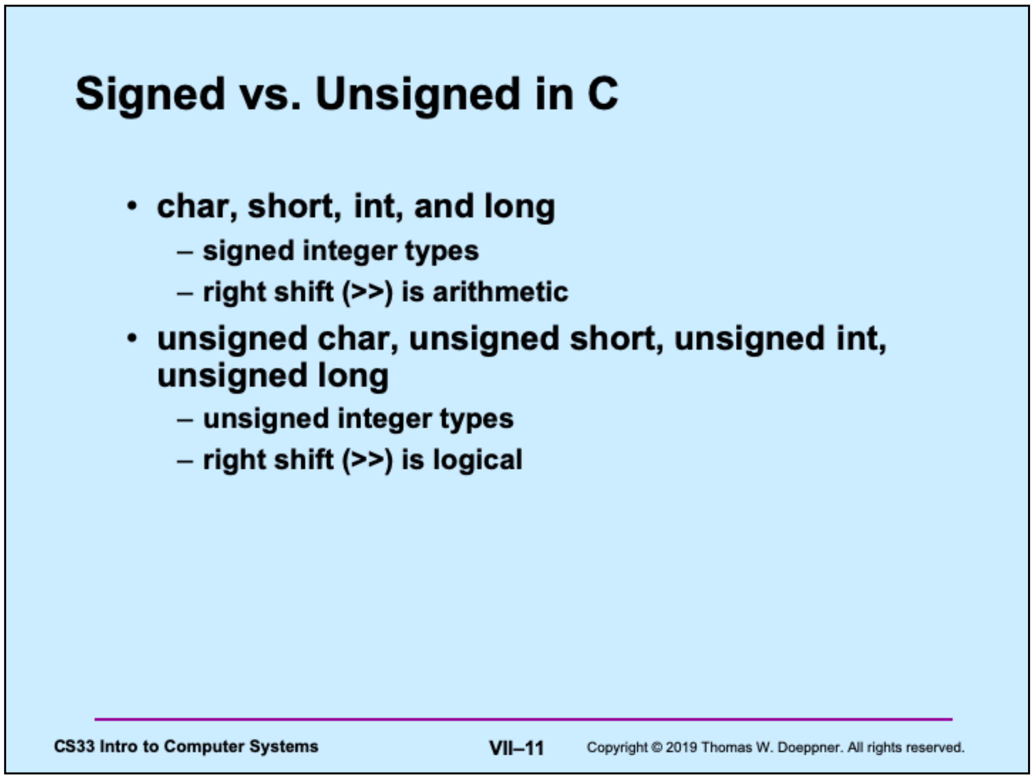
$$= 2^w$$

$$= \begin{array}{|c|c|c|c|c|c|c|c|} \hline 1 & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ \hline \end{array}$$

If we add to the two's complement representation of a w-bit number the result of adding one to its bitwise complement, we get a w+1-bit number whose low-order w bits are zeroes and whose high-order bit is one. However, since we're constrained to only w bits, the result is a w-bit value of all zeroes, plus an overflow. If we ignore the overflow, the result is zero.

Quiz 1

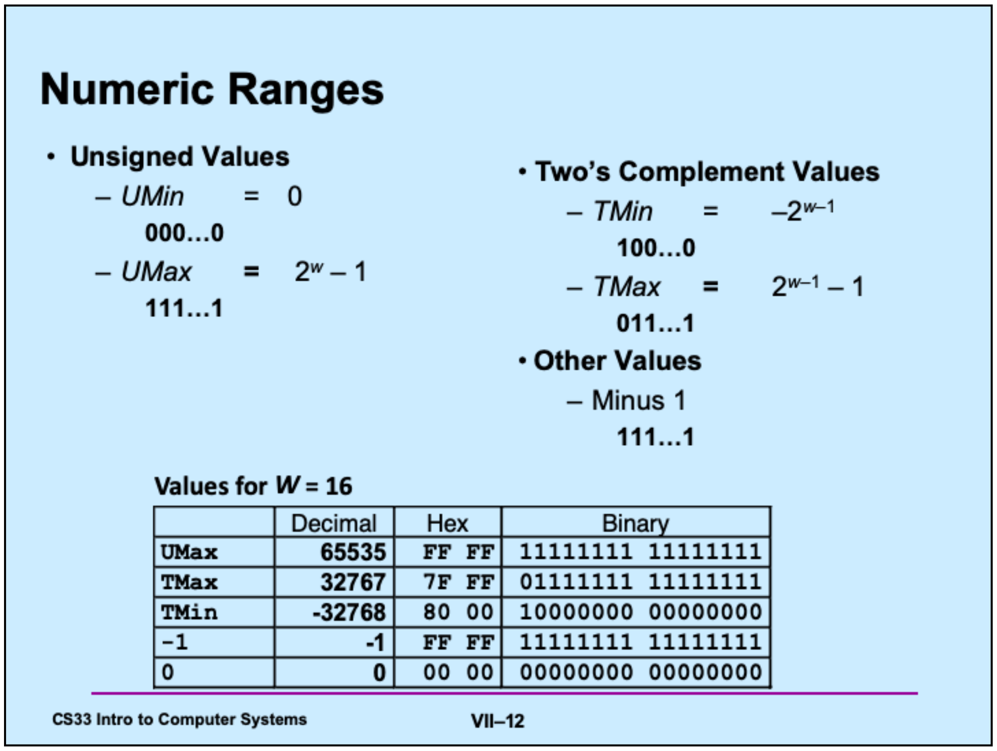
- We have a computer with 4-bit words that uses two's complement to represent negative numbers. What is the result of subtracting 0010 (2) from 0001 (1)?
  a) 0111
  b) 1001
  c) 1110
  d) 1111



Signed vs. Unsigned in C

- char, short, int, and long
  - signed integer types
  - right shift (>>) is arithmetic
- unsigned char, unsigned short, unsigned int, unsigned long
  - unsigned integer types
  - right shift (>>) is logical

Why the signed integer types use the arithmetic right shift will be clear by the end of this lecture.



Numeric Ranges

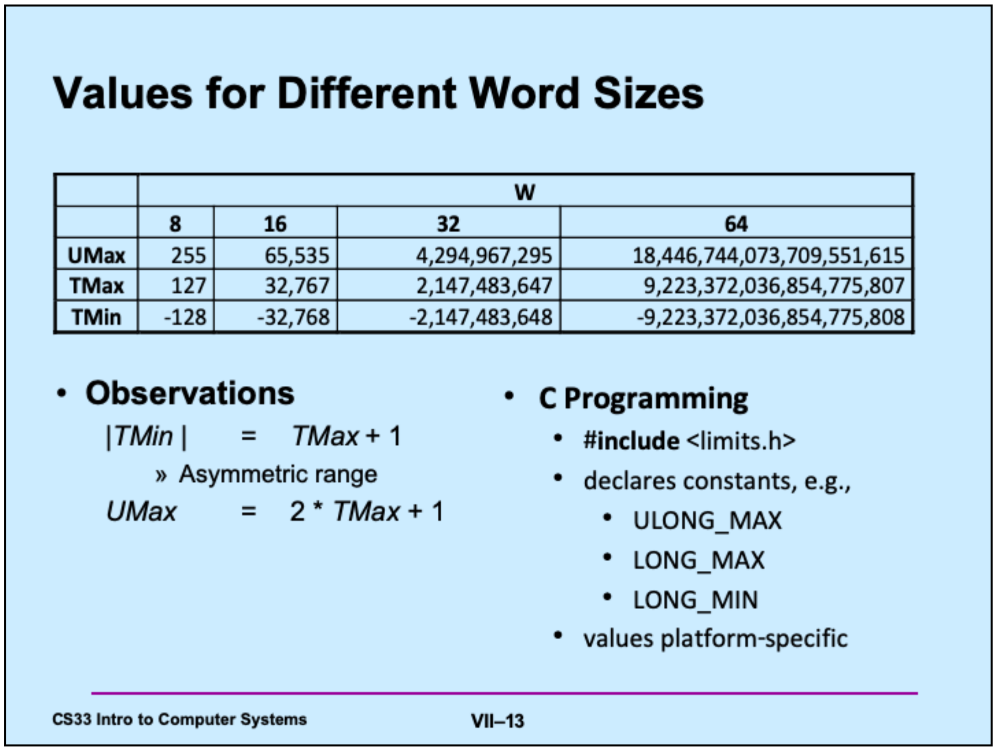
- **Unsigned Values**
  - *UMin* = 0
    000…0
  - *UMax* = $2^w - 1$
    111…1

- **Two's Complement Values**
  - *TMin* = -2^{w-1}
    100…0
  - *TMax* = $2^{w-1} - 1$
    011…1
- **Other Values**
  - Minus 1
    111…1

Values for *W* = 16

	Decimal	Hex	Binary
UMax	65535	FF FF	11111111 11111111
TMax	32767	7F FF	01111111 11111111
TMin	-32768	80 00	10000000 00000000
-1	-1	FF FF	11111111 11111111
0	0	00 00	00000000 00000000

Supplied by CMU.



Values for Different Word Sizes

	W			
	8	16	32	64
UMax	255	65,535	4,294,967,295	18,446,744,073,709,551,615
TMax	127	32,767	2,147,483,647	9,223,372,036,854,775,807
TMin	-128	-32,768	-2,147,483,648	-9,223,372,036,854,775,808

- **Observations**
  - *\|TMin \|* = *TMax* + 1
    - » Asymmetric range
  - *UMax* = 2 * *TMax* + 1

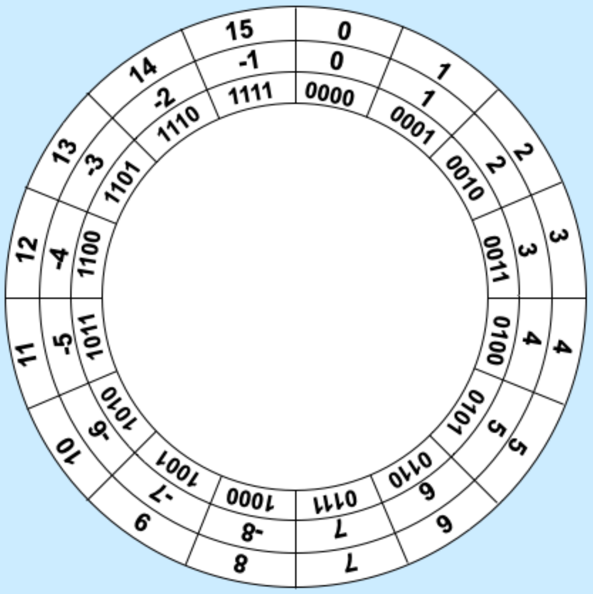
- **C Programming**
  - **#include** <limits.h>
  - declares constants, e.g.,
    - ULONG_MAX
    - LONG_MAX
    - LONG_MIN
  - values platform-specific

Supplied by CMU.

Quiz 2

- What is –TMin (assuming two's complement signed integers)?
  a) TMin
  b) TMax
  c) 0
  d) 1

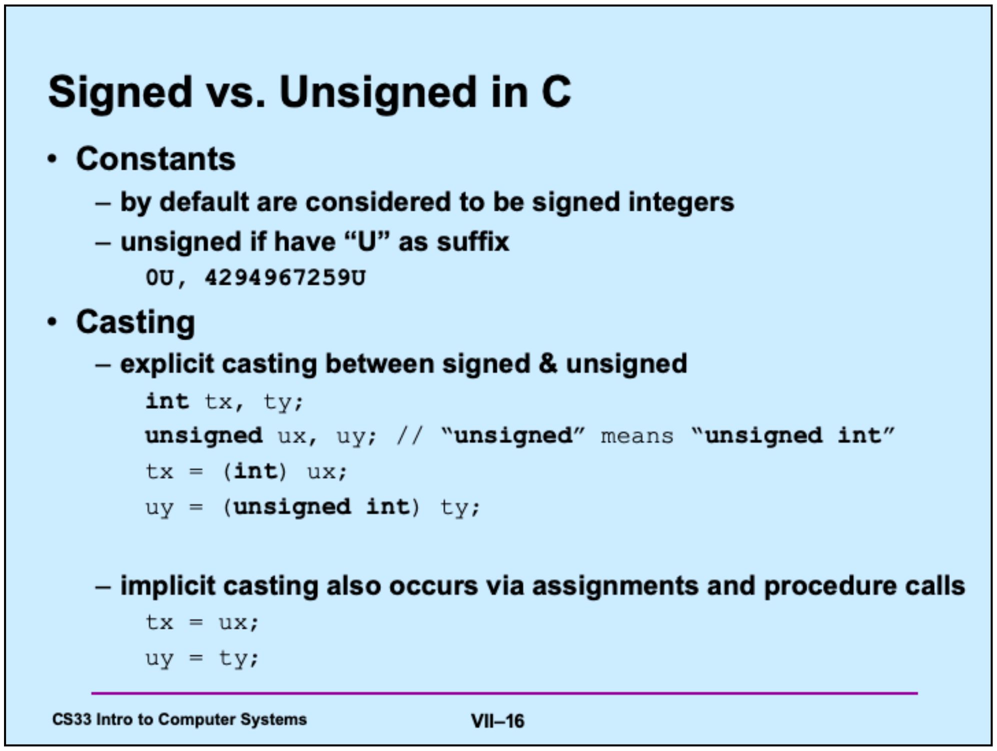
4-Bit Computer Arithmetic



Unsigned computer arithmetic is performed modulo 2 to the power of the computer's word size. The outer ring of the figure demonstrates arithmetic modulo 2^4. To see the result, for example, of adding 3 to 2, start at 2 and go around the ring three units in the clockwise direction. If we add 5 to 14, we start at 14 and move 5 units clockwise, to 3. Similarly, to subtract 3 from 1, we start at one and move three units counterclockwise to 14.

What about two's-complement computer arithmetic? We know that the values encoded in a 4-bit computer word range from -8 to 7. How do we arrange them in the ring? As shown in the second ring, it makes sense for the non-negative numbers to be in the same positions as the corresponding unsigned values. It clearly makes sense for the integer coming just before 0 to be -1, the integer just before -1 to be -2, etc. Thus, since we have a ring, the integer following 7 is -8. Now we can see how arithmetic works for two's-complement numbers. Adding 3 to 2 works just as it does for unsigned numbers. Subtracting 3 from 1 results in -2. But adding 3 to 6 results in -7; and adding 5 to -2 results in 3.

The innermost ring shows the bit encodings for the unsigned and two's-complement values. The point of all this is that, with only one implementation of arithmetic, we can handle both unsigned and two's-complement values. Thus adding unsigned 5 and 9 is equivalent to adding two's-complement 5 and -7. The result will 1110, which, if interpreted as an unsigned value is 14, but if interpreted as a two's-complement value is -2.



Signed vs. Unsigned in C

- **Constants**
  - by default are considered to be signed integers
  - unsigned if have "U" as suffix

```
0U, 4294967259U
```

- **Casting**
  - explicit casting between signed & unsigned

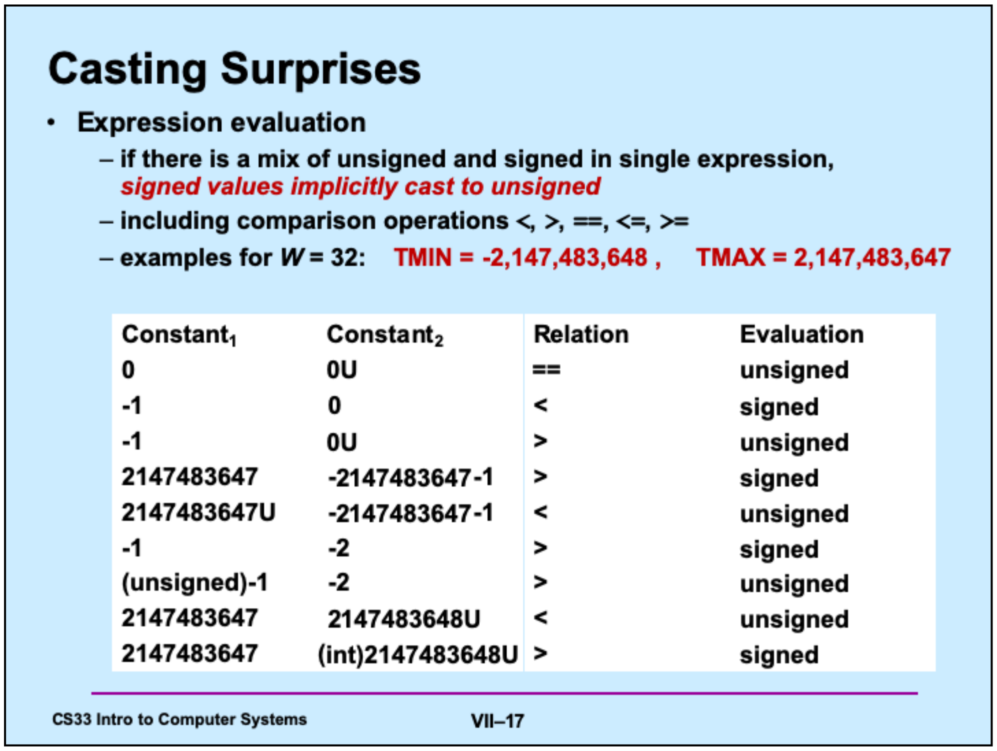
```
int tx, ty;
unsigned ux, uy; // "unsigned" means "unsigned int"
tx = (int) ux;
uy = (unsigned int) ty;
```

  - implicit casting also occurs via assignments and procedure calls

```
tx = ux;
uy = ty;
```

Supplied by CMU.

Note that the kind of casting done here is what we called "intimidation" in the previous lecture: no actual conversion takes place, but the value is reinterpreted according to the cast.



Casting Surprises

- Expression evaluation
  - if there is a mix of unsigned and signed in single expression, *signed values implicitly cast to unsigned*
  - including comparison operations <, >, ==, <=, >=
  - examples for *W* = 32: TMIN = -2,147,483,648 , TMAX = 2,147,483,647

Constant$_1$	Constant$_2$	Relation	Evaluation
0	0U	==	unsigned
-1	0	<	signed
-1	0U	>	unsigned
2147483647	-2147483647-1	>	signed
2147483647U	-2147483647-1	<	unsigned
-1	-2	>	signed
(unsigned)-1	-2	>	unsigned
2147483647	2147483648U	<	unsigned
2147483647	(int)2147483648U	>	signed

Supplied by CMU.

Quiz 3

What is the value of

```
      (long)ULONG_MAX - (unsigned long)-1
```

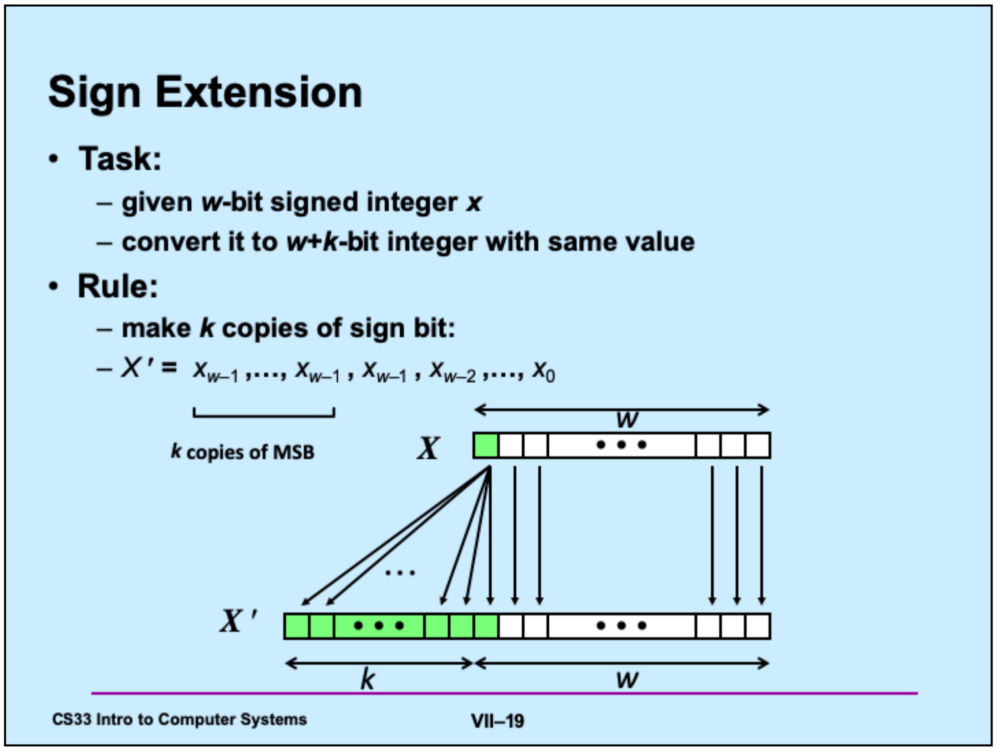
???

a) -1

b) 0

c) 1

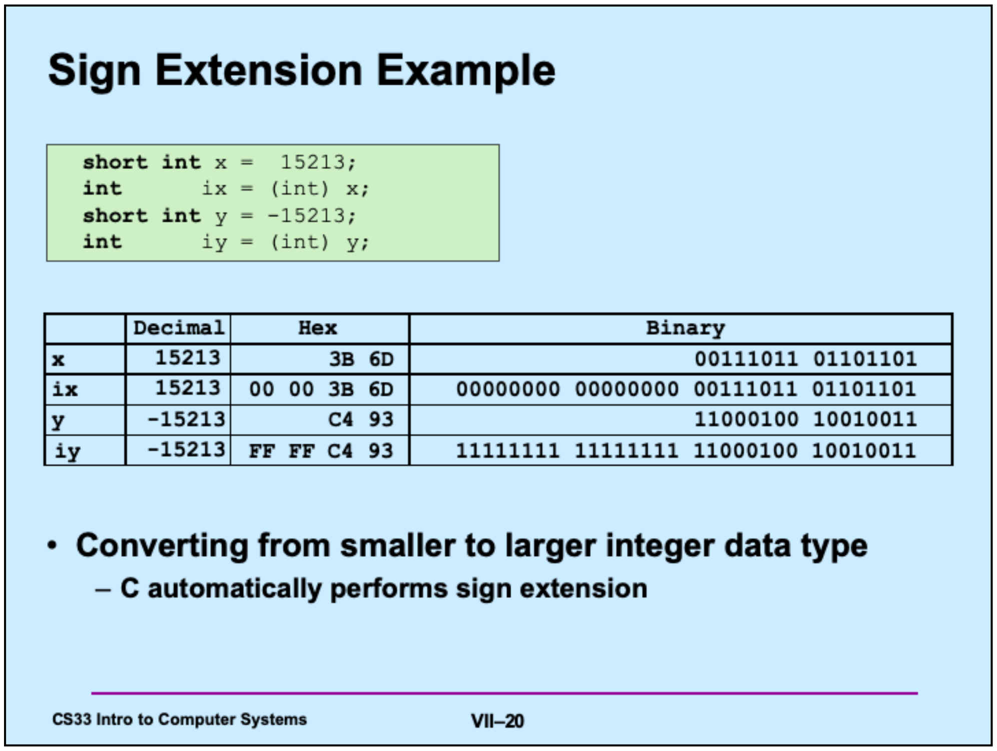
d) ULONG_MAX



Sign Extension

- **Task:**
  - given w-bit signed integer x
  - convert it to $w+k$-bit integer with same value
- **Rule:**
  - make k copies of sign bit:
  - $X' = \underbrace{x_{w-1}, \ldots, x_{w-1}}_{k \text{ copies of MSB}}, x_{w-1}, x_{w-2}, \ldots, x_0$

Supplied by CMU.

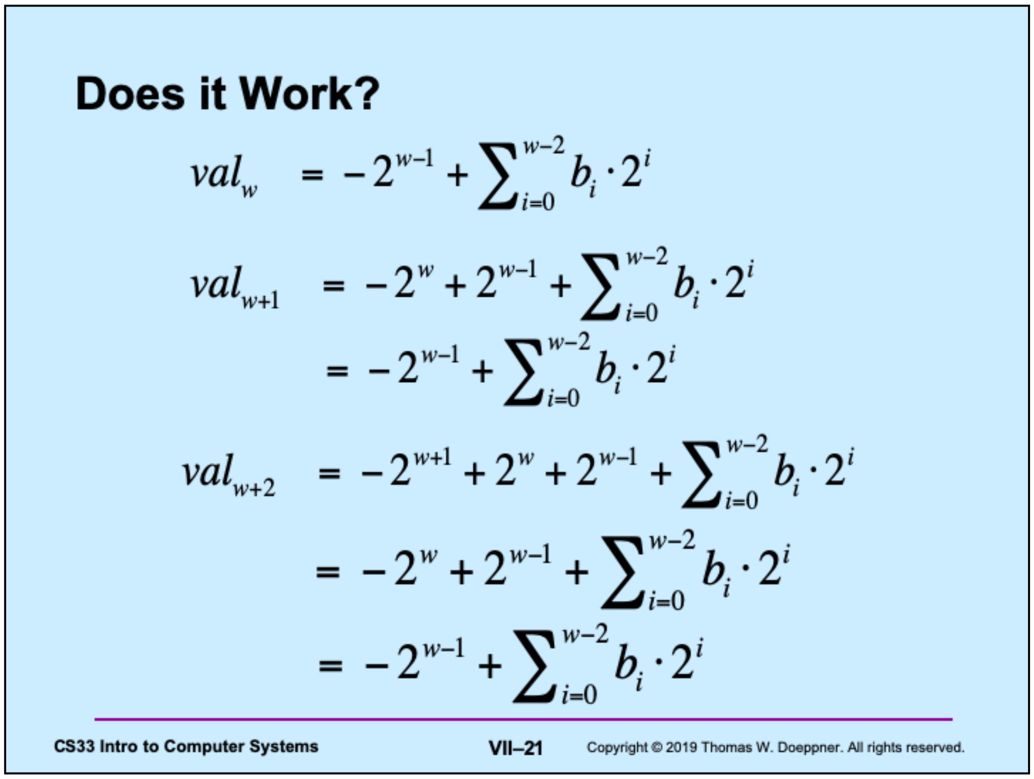

Sign Extension Example

```
short int x =  15213;
int       ix = (int) x;
short int y = -15213;
int       iy = (int) y;
```

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
ix	15213	00 00 3B 6D	00000000 00000000 00111011 01101101
y	-15213	C4 93	11000100 10010011
iy	-15213	FF FF C4 93	11111111 11111111 11000100 10010011

- **Converting from smaller to larger integer data type**
  - **C automatically performs sign extension**

Supplied by CMU.



Does it Work?

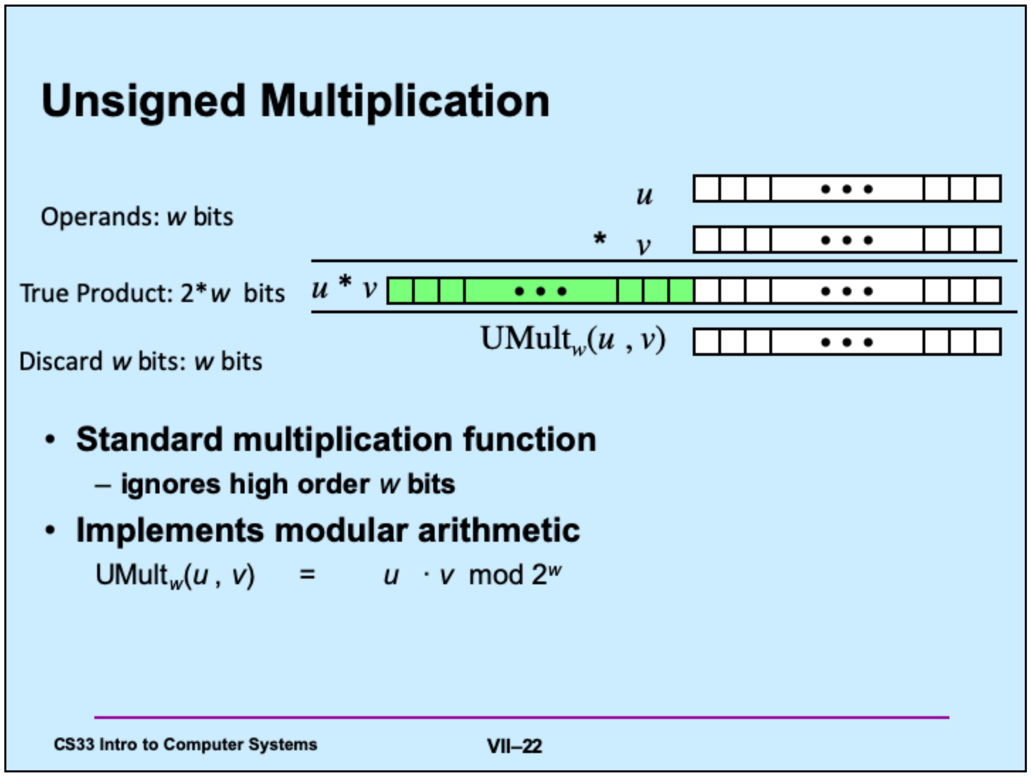
$$val_w = -2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i$$

$$\begin{aligned} val_{w+1} &= -2^w + 2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i \\ &= -2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i \end{aligned}$$

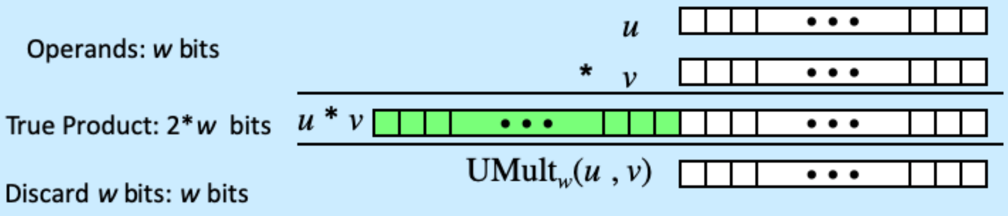
$$\begin{aligned} val_{w+2} &= -2^{w+1} + 2^w + 2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i \\ &= -2^w + 2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i \\ &= -2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i \end{aligned}$$

Sign extension clearly works for positive and zero values (where the sign bit is zero). But does it work for negative values? The first line of the slide shows the computation of the value of a w-bit item with a sign bit of one (i.e., it's negative). The next two lines show what happens if we extend this to a w+1-bit item, extending the sign bit. What had been the sign bit becomes one of the value bits, and its contribution to the value is now positive rather than negative. But this is compensated by the new sign bit, whose contribution is a negative value, twice as large as the original sign bit. Thus the net effect is for there to be no change in the value.

We do this again, extending to a w+2-bit item, and again, the resulting value is the same as what we started with.



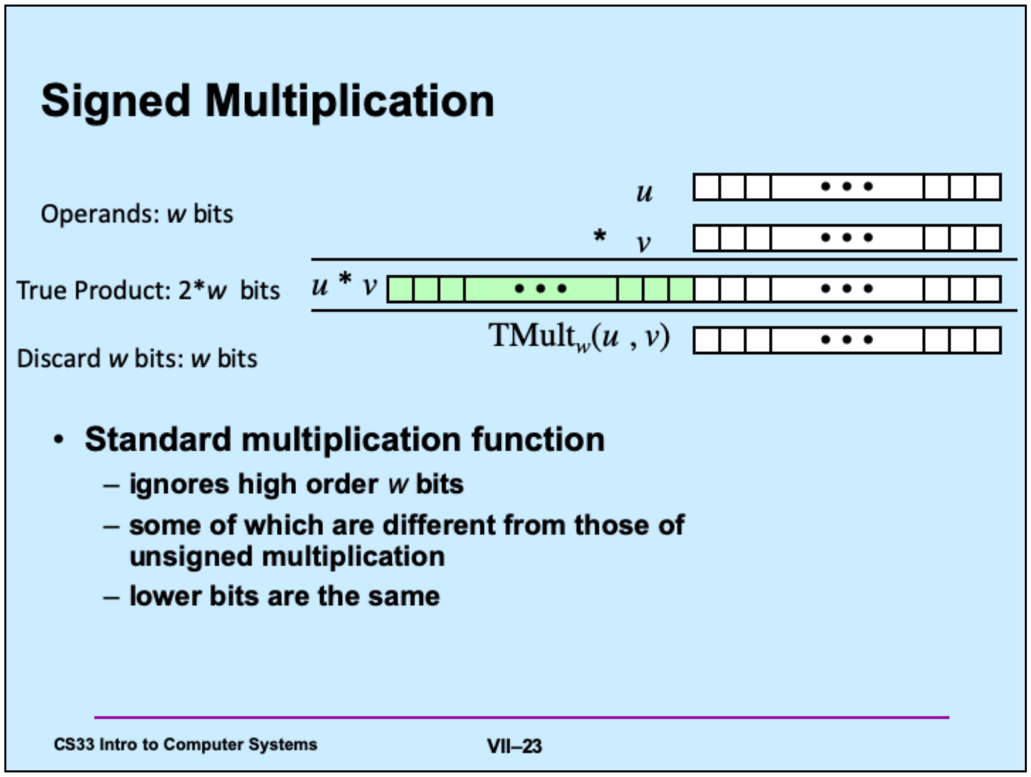
Unsigned Multiplication



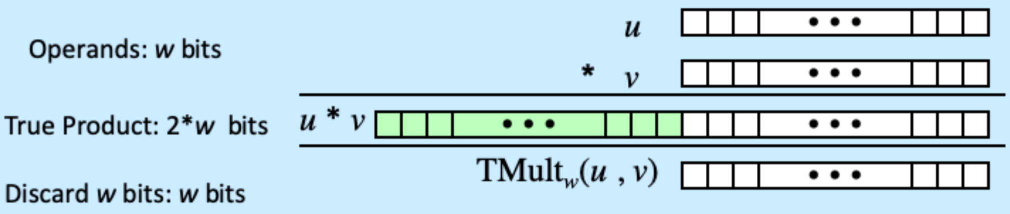
- **Standard multiplication function**
  - **ignores high order w bits**
- **Implements modular arithmetic**

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

Supplied by CMU.



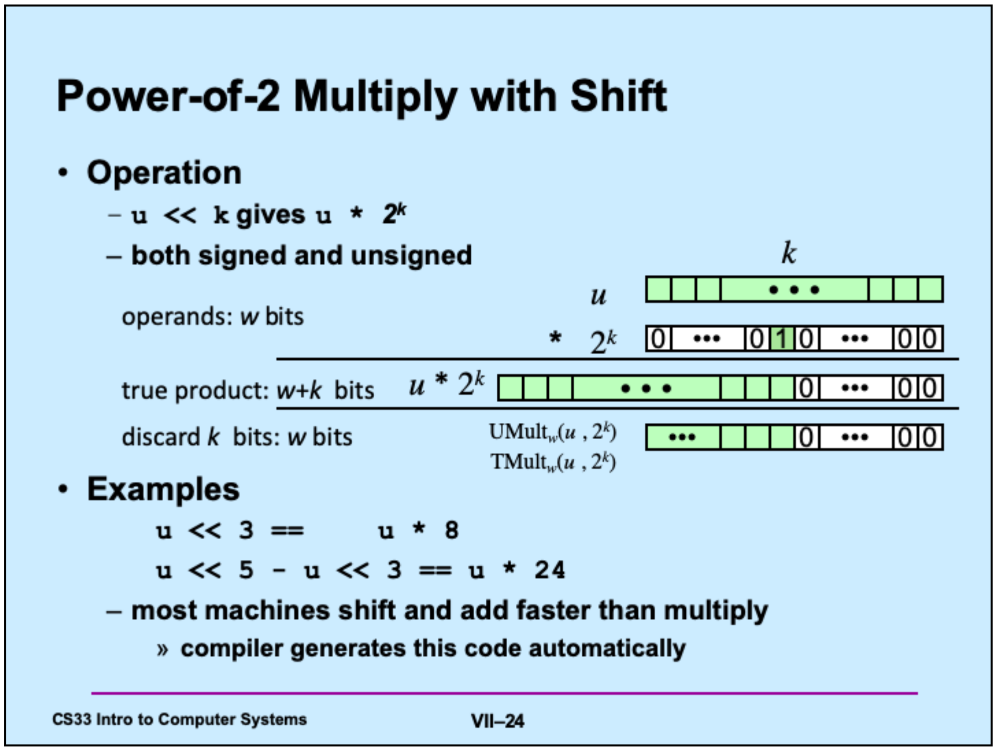
Signed Multiplication



- **Standard multiplication function**
  - **ignores high order *w* bits**
  - **some of which are different from those of unsigned multiplication**
  - **lower bits are the same**

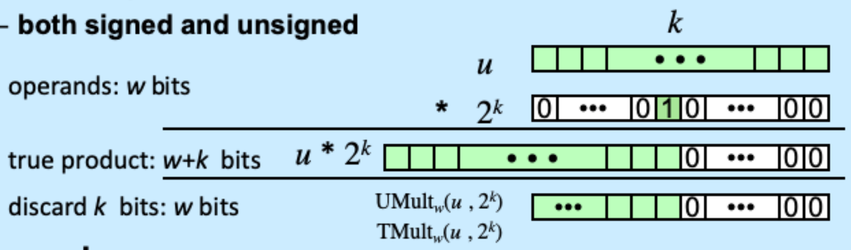
Supplied by CMU.

How is it that the "true product" is different from that of unsigned multiplication? Consider what the true product should be is the multiplier is -1 and the multiplicand is 1. Thus the multiplier is a w-bit word of all ones and the multiplicand is a w-bit word of all zeroes except for the least-significant bit, which is 1. The high-order w bits of the true product should be all ones (since it's negative), but with unsigned multiplication they'd be all zeroes. However, since we're ignoring the high-order w bits, this doesn't matter.



Power-of-2 Multiply with Shift

- **Operation**
  - `u << k` gives `u * 2`^{k}
  - both signed and unsigned

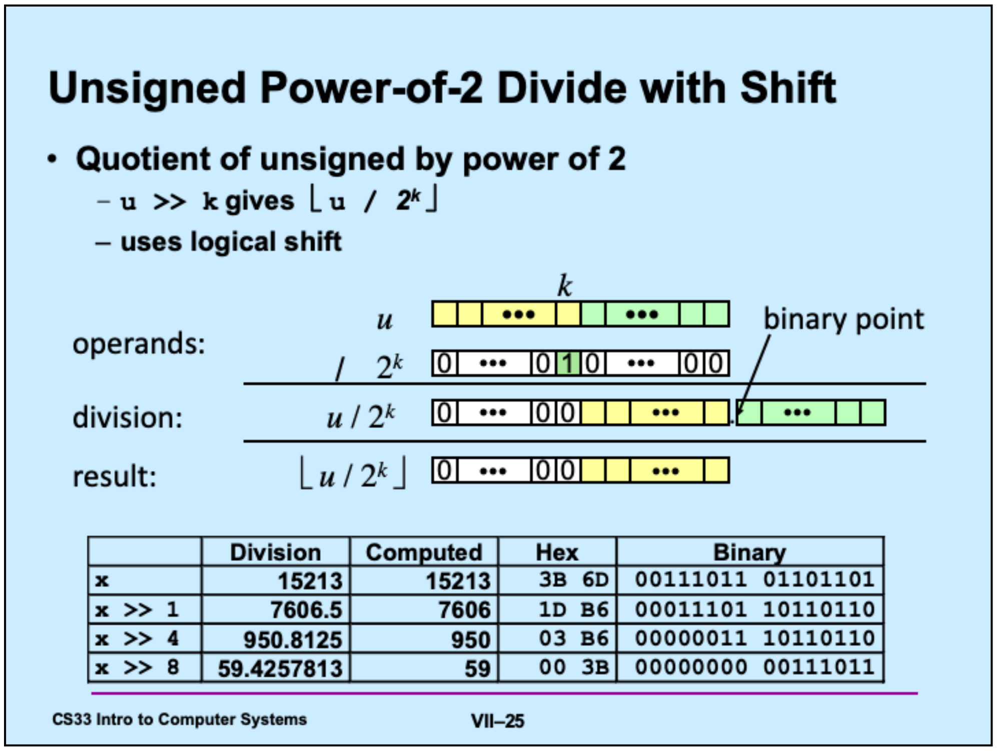


- **Examples**

```
u << 3 ==      u * 8
u << 5 - u << 3 == u * 24
```

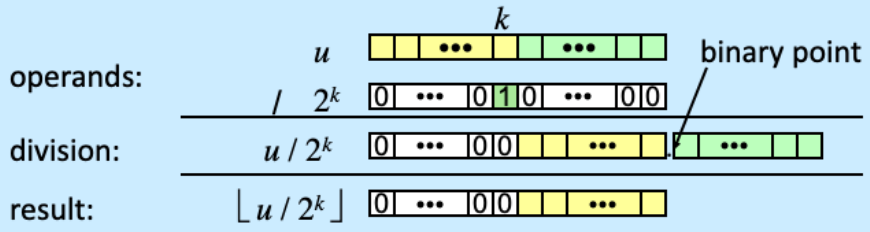
  - most machines shift and add faster than multiply
    - » compiler generates this code automatically

Supplied by CMU.



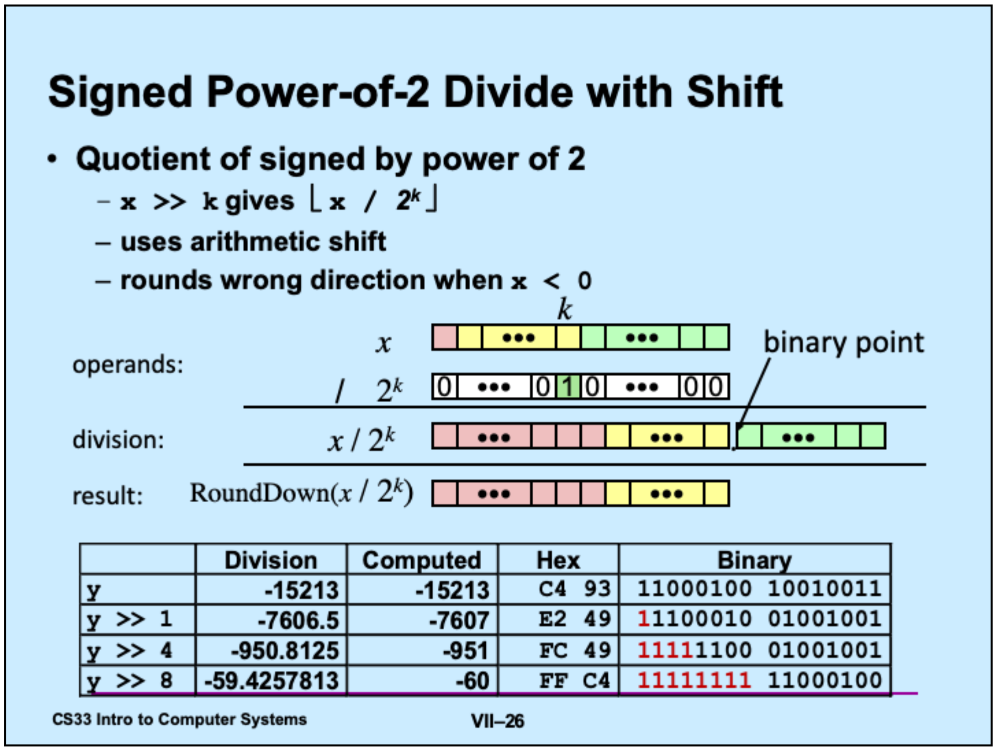
Unsigned Power-of-2 Divide with Shift

- **Quotient of unsigned by power of 2**
  - `u >> k` **gives** $\lfloor u / 2^k \rfloor$
  - **uses logical shift**



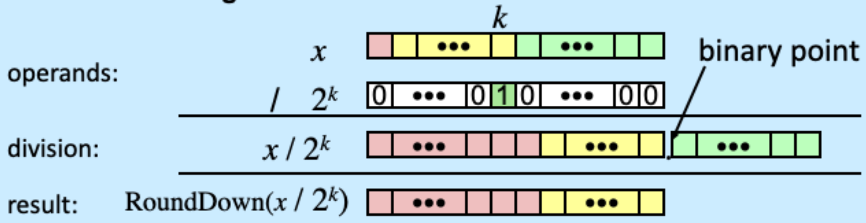
	Division	Computed	Hex	Binary
`x`	15213	15213	`3B 6D`	`00111011 01101101`
`x >> 1`	7606.5	7606	`1D B6`	`00011101 10110110`
`x >> 4`	950.8125	950	`03 B6`	`00000011 10110110`
`x >> 8`	59.4257813	59	`00 3B`	`00000000 00111011`

Supplied by CMU.



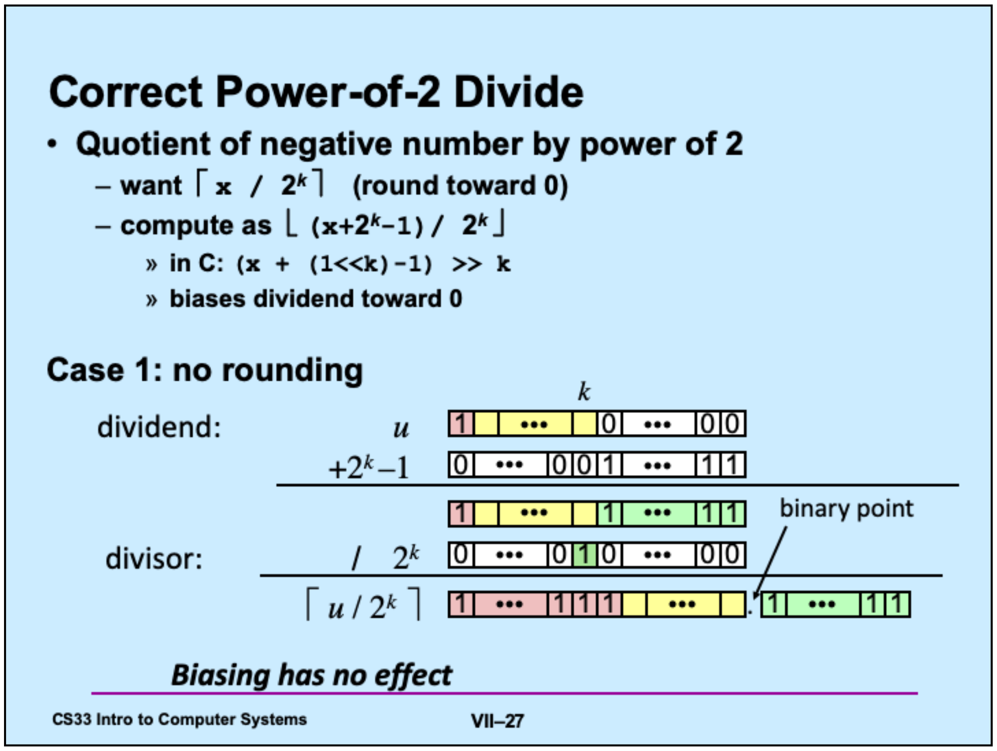
Signed Power-of-2 Divide with Shift

- **Quotient of signed by power of 2**
  - `x >> k` gives $\lfloor x / 2^k \rfloor$
  - uses arithmetic shift
  - rounds wrong direction when `x < 0`



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

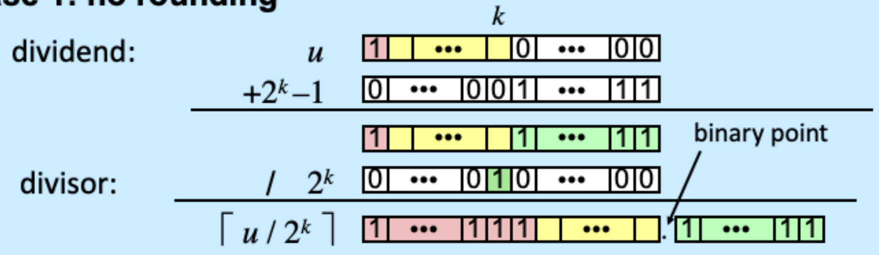
Supplied by CMU.



Correct Power-of-2 Divide

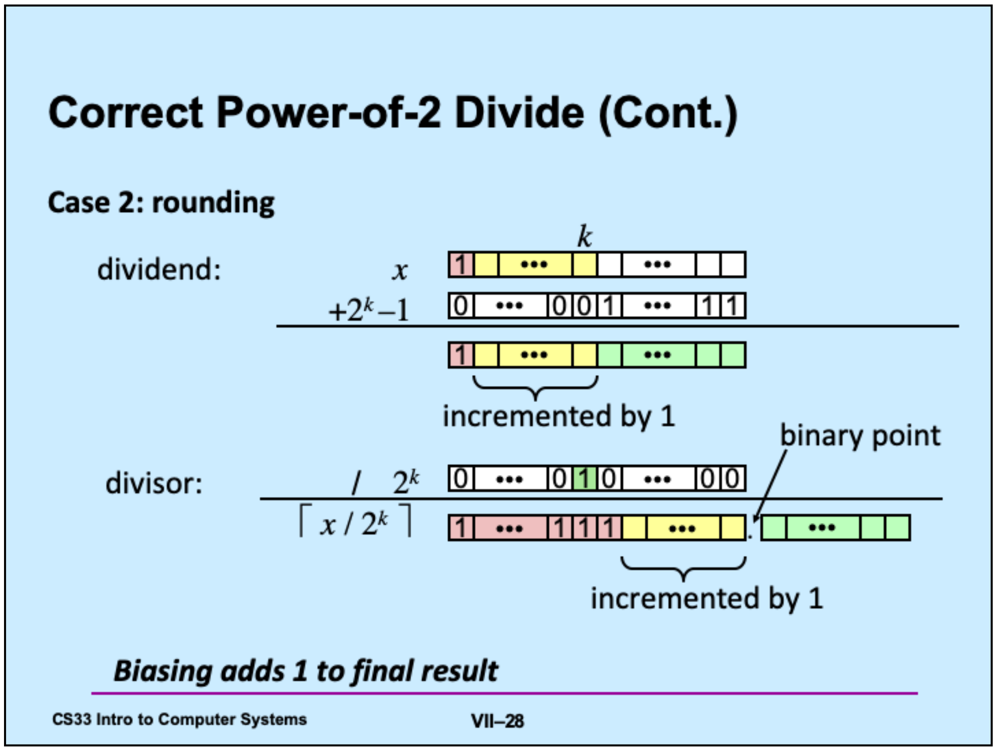
- **Quotient of negative number by power of 2**
  - want $\lceil$ `x / 2`$^k\rceil$ (round toward 0)
  - compute as $\lfloor$ `(x+2`^{k}`-1)/ 2`$^k\rfloor$
    - » in C: `(x + (1<<k)-1) >> k`
    - » biases dividend toward 0

Case 1: no rounding



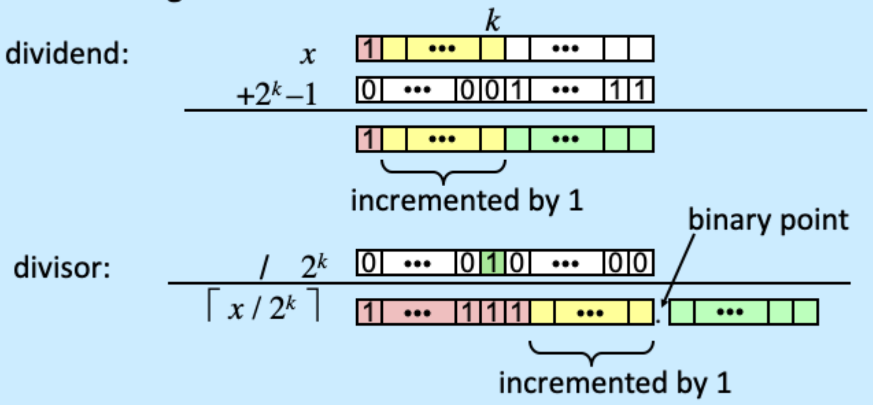
Biasing has no effect

Supplied by CMU.



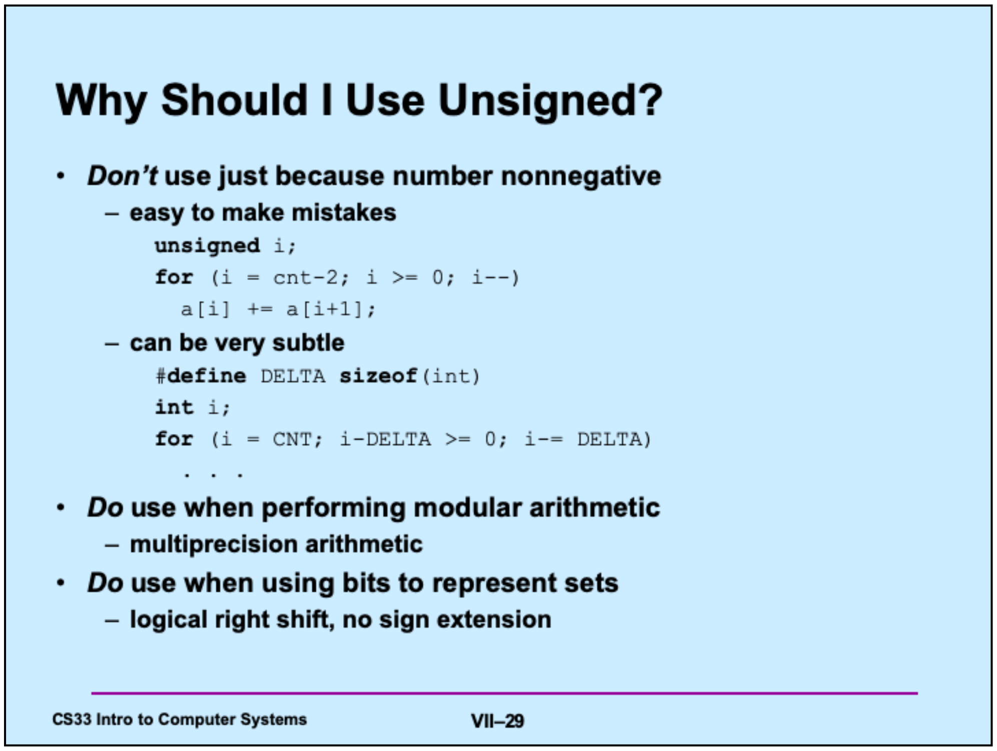
Correct Power-of-2 Divide (Cont.)

Case 2: rounding



Biasing adds 1 to final result

Supplied by CMU.



Why Should I Use Unsigned?

- ***Don't*** **use just because number nonnegative**
  - **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)
  . . .
```

- ***Do*** **use when performing modular arithmetic**
  - **multiprecision arithmetic**
- ***Do*** **use when using bits to represent sets**
  - **logical right shift, no sign extension**

Supplied by CMU.

Note that "sizeof" returns an unsigned value. (Recall that, when mixing signed and unsigned items in an expression, the result will be unsigned.)