

# CS 261 Fall 2019

Mike Lam, Professor



## Binary Arithmetic

# Binary Arithmetic

- Topics
  - Basic addition
  - Overflow
  - Multiplication & Division

# Basic addition

- Binary and hex addition are fundamentally the same as decimal addition
  - Add digit-by-digit, using a carry as necessary
  - Result could require one more bit than the operands

	<b>Dec</b>		<b>Bin</b>
12540		10011100	
+ <u>4683</u>		+ <u>1010110</u>	

b0994f	<b>Hex</b>
+ <u>7120</u>	

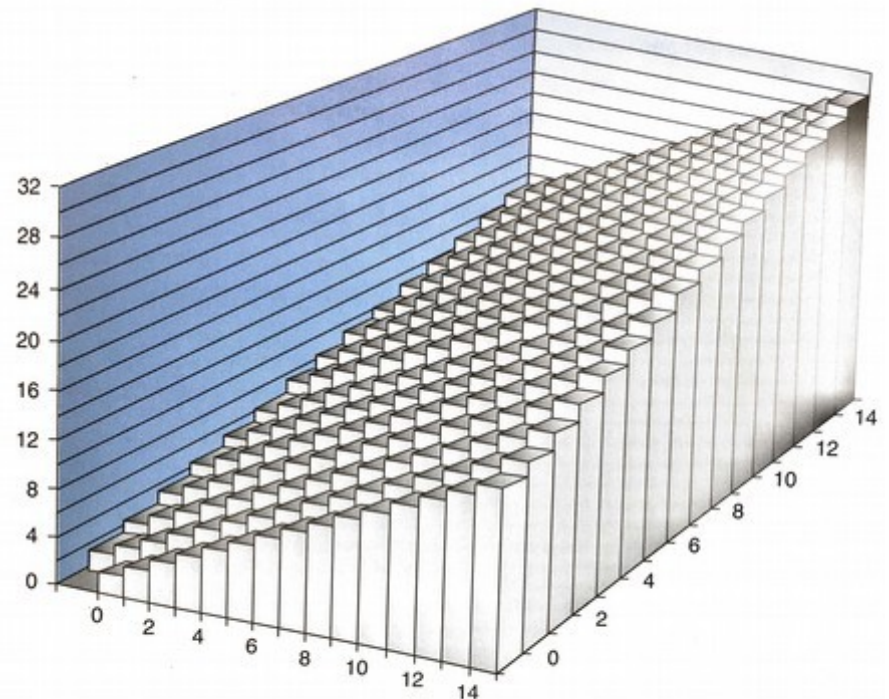


Figure 2.21 Integer addition. With a 4-bit word size, the sum could require 5 bits.

# Basic addition

- Binary and hex addition are fundamentally the same as decimal addition
  - Add digit-by-digit, using a carry as necessary
  - Result could require one more bit than the operands

<b>11</b>	<b>Dec</b>	<b>111</b>	<b>Bin</b>
12540		10011100	
+ 4683		+ 1010110	
<b>17223</b>		<b>11110010</b>	

<b>1</b>	<b>Hex</b>
b0994f	
+ 7120	
<b>b10a6f</b>	

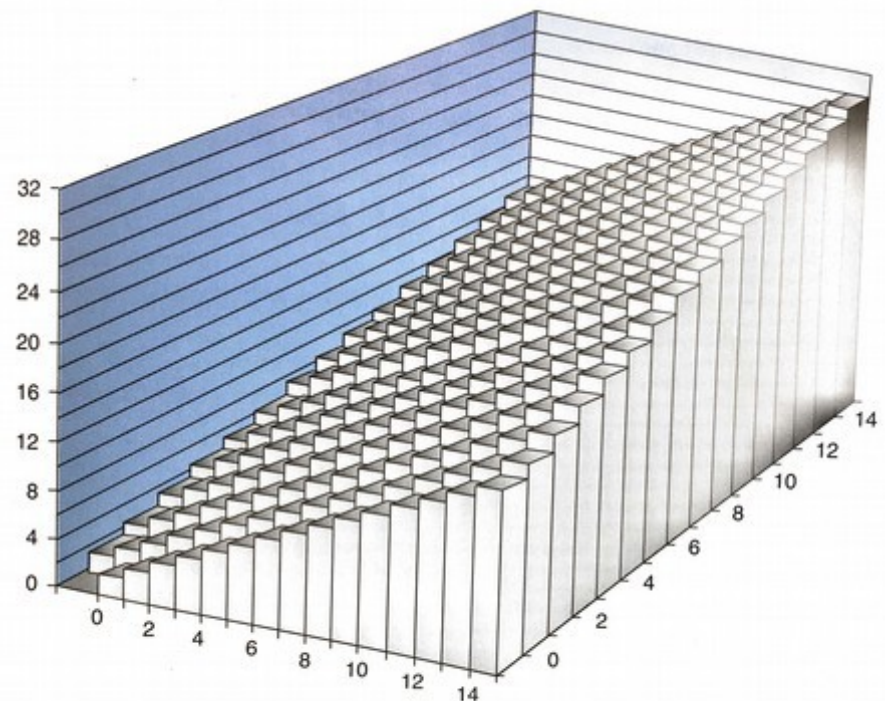


Figure 2.21 Integer addition. With a 4-bit word size, the sum could require 5 bits.

# Overflow

- Unsigned addition is subject to overflow
  - Caused by truncation to integer size

$$\begin{array}{r} 1 \\ + \quad 994f \\ \hline 10a6f = 0a6f \end{array}$$

Truncation!

(assume a 16-bit integer)

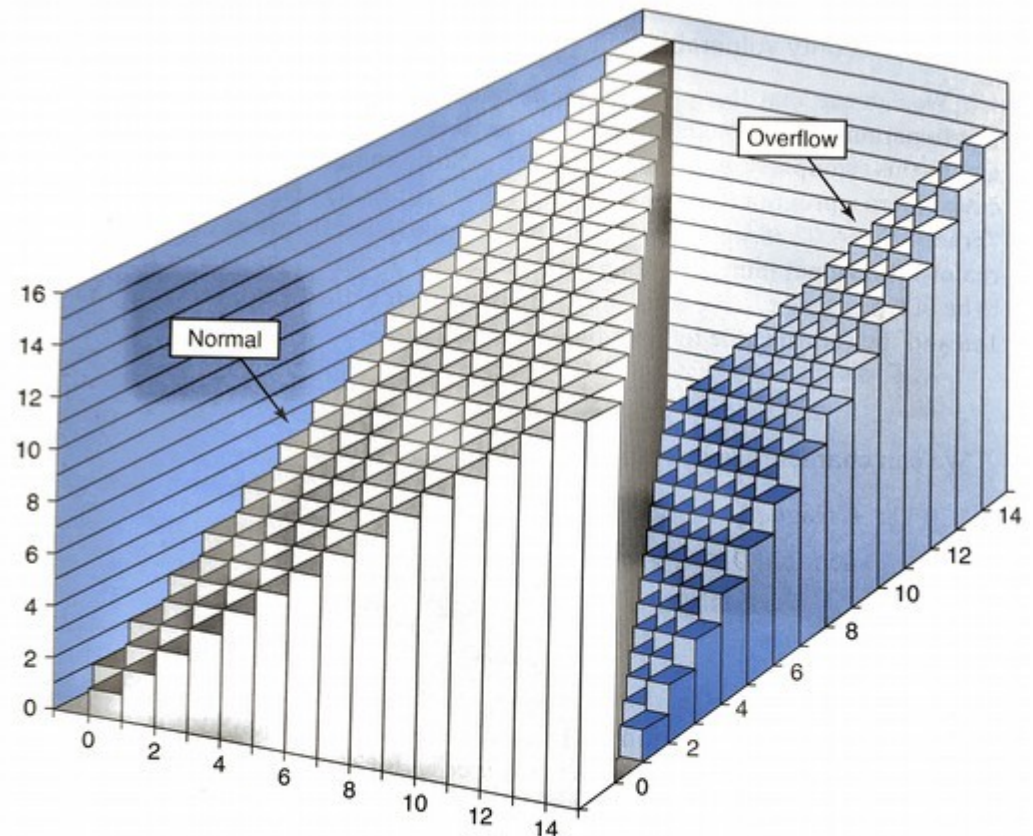
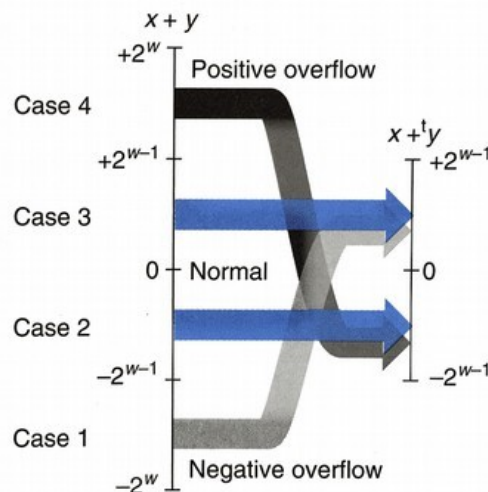


Figure 2.23 Unsigned addition. With a 4-bit word size, addition is performed modulo 16.

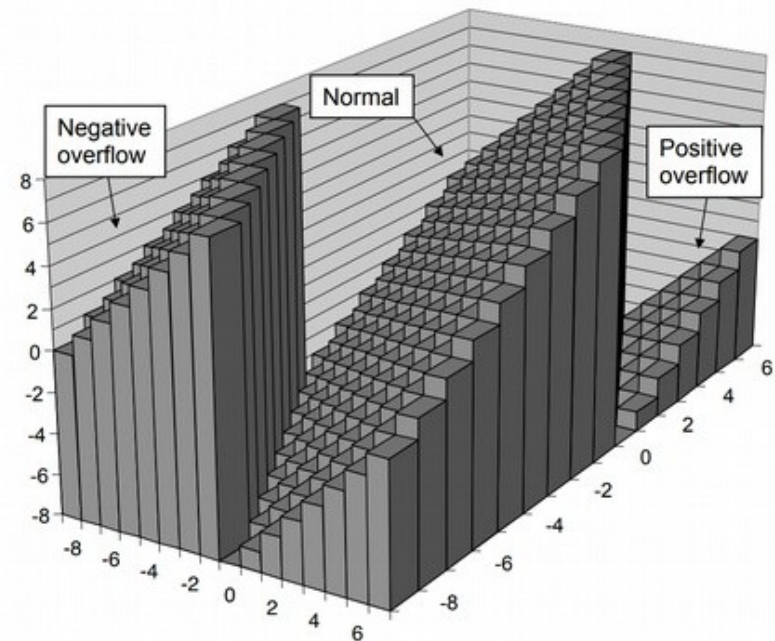
# Overflow

- Two's complement addition is identical to unsigned mechanically
  - Subject to both positive and negative overflow
  - Overflows if carry-in and carry-out differ for sign bit

**Figure 2.24**  
Relation between integer and two's-complement addition. When  $x + y$  is less than  $-2^{w-1}$ , there is a negative overflow. When it is greater than or equal to  $2^{w-1}$ , there is a positive overflow.



Two's complement addition (4-bit word)



**Figure 2.26** Two's-complement addition. With a 4-bit word size, addition can have a negative overflow when  $x + y < -8$  and a positive overflow when  $x + y \geq 8$ .

**NOTE: this figure is printed incorrectly in your textbook!**

# Multiplication & Division

- Like addition, fundamentally the same as base 10
  - Actually, it's even simpler!
  - Same regardless of encoding
- Special case: multiply by powers of 2 (shift left)

$$\begin{array}{ll} 2 \ll 1 = 4 & (2 * 2) \\ 1 \ll 2 = 4 & (1 * 2 * 2) \\ \\ 1 \ll 4 = 16 & (1 * 2 * 2 * 2 * 2) \\ 4 \ll 1 = 8 & (4 * 2) \\ 4 \ll 2 = 16 & (4 * 2 * 2) \end{array}$$

- Division is expensive!
  - Special case: divide by powers of two (shift right)
    - Logical shift for unsigned numbers, arithmetic shift for signed numbers

$$\begin{array}{r} 101 \quad (5) \\ \times \underline{11} \quad (3) \\ \hline 101 \\ 101 \\ \hline 1111 \quad (15) \end{array}$$

# Review

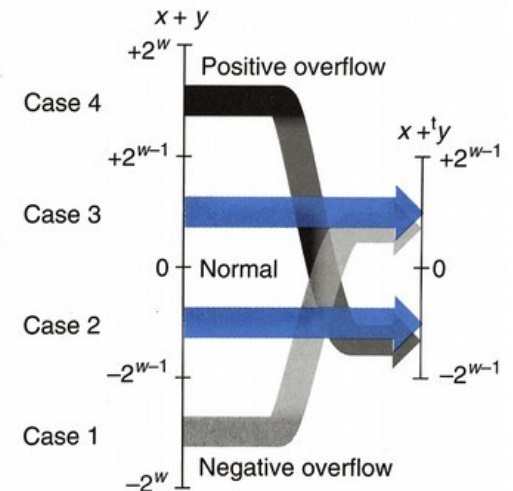
- One-byte integers:

<u>Binary</u> _____	<u>Unsigned</u>	<u>Two's C</u>
1111 1111	255	-1
1111 1110	254	-2
...	...	...
1000 0001	129	-127
1000 0000	128	-128
-----	-----	-----
0111 1111	127	127
0111 1110	126	126
...	...	...
0000 0001	1	1
0000 0000	0	0

**Overflow**  
when  $x + y > 255$

**Positive overflow** when  $x + y > 127$   
**Negative overflow** when  $x + y < -128$

**Figure 2.24**  
Relation between integer and two's-complement addition. When  $x + y$  is less than  $-2^{w-1}$ , there is a negative overflow. When it is greater than or equal to  $2^{w-1}$ , there is a positive overflow.





# Binary fractions

- Now we can store integers
  - But what about general real numbers?
- Extend positional binary integers to store fractions
  - Designate a certain number of bits for the fractional part
  - These bits represent negative powers of two
  - (Just like fractional digits in decimal fractions!)

**101.101**  
4    2    1        1/2   1/4   1/8

$$4 + 1 + 0.5 + 0.125 = \mathbf{5.625}$$

*(alternatively: 5 + 5/8)*

# Another problem

- For scientific applications, we want to be able to store a wide *range* of values
  - From the scale of galaxies down to the scale of atoms
- Doing this with fixed-precision numbers is difficult
  - Even signed 64-bit integers
    - Perhaps allocate half for whole number, half for fraction
    - Range:  $\sim 2 \times 10^{-9}$  through  $\sim 2 \times 10^9$

Floating-point demonstration using Super Mario 64:

<https://www.youtube.com/watch?v=9hdFG2GcNuA>



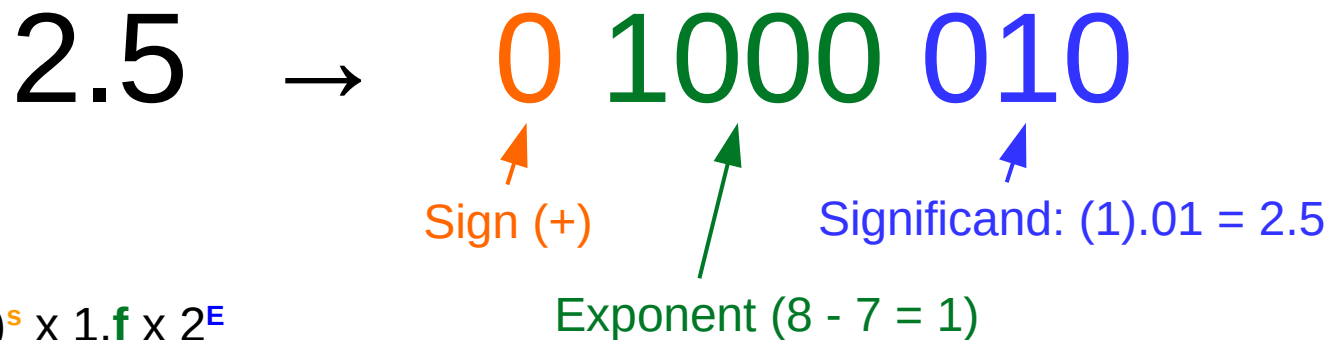
# Floating-point numbers

- Scientific notation to the rescue!
  - Traditionally, we write large (or small) numbers as  $x \cdot 10^e$
  - This is how **floating-point** representations work
    - Store **exponent** and fractional parts (the **significand**) separately
    - The decimal point “floats” on the number line
    - Position of point is based on the exponent

$$1.23 = \begin{array}{l} 0.0123 \times 10^2 \\ 0.123 \times 10^1 \\ \mathbf{1.23 \times 10^0} \\ 12.3 \times 10^{-1} \\ 123.0 \times 10^{-2} \end{array}$$

# Floating-point numbers

- However, computers use binary
  - So floating-point numbers use base 2 scientific notation ( $x \cdot 2^e$ )
- Fixed width field
  - Reserve one bit for the sign bit (0 is positive, 1 is negative)
  - Reserve n bits for **biased** exponent (bias is  $2^{n-1} - 1$ )
    - Avoids having to use two's complement
  - Use remaining bits for normalized fraction (implicit leading 1)
    - Exception: if the exponent is zero, don't normalize



$$\text{Value} = (-1)^S \times 1.f \times 2^E$$