CS 261
Spring 2024

Mike Lam, Professor

Binary Arithmetic

Binary Arithmetic

- Topics
  - Basic addition
  - Overflow
  - Multiplication & division
  - Floating-point preview

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

Dec

```
  12540
+  4683
```

Bin

```
  10011100
+  1010110
```

Hex

```
  b0994f
+   7120
```

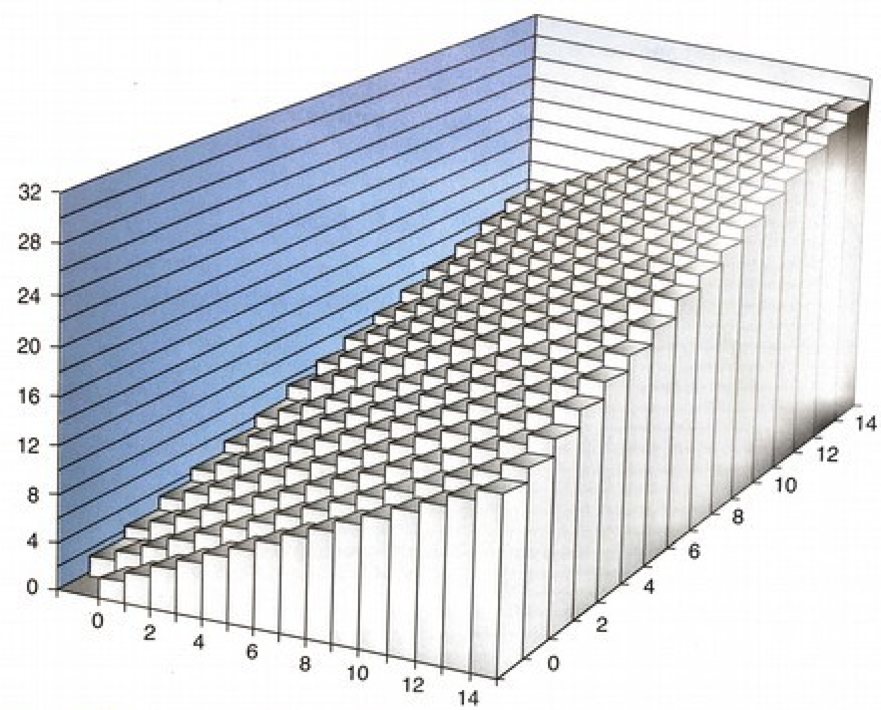


Figure 2.21 **Integer addition.** With a 4-bit word size, the sum could require 5 bits.

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

Dec

```
    11
  12540
+  4683
  17223
```

Bin

```
     111
  10011100
+  1010110
  11110010
```

Hex

```
    1
  b0994f
+   7120
  b10a6f
```

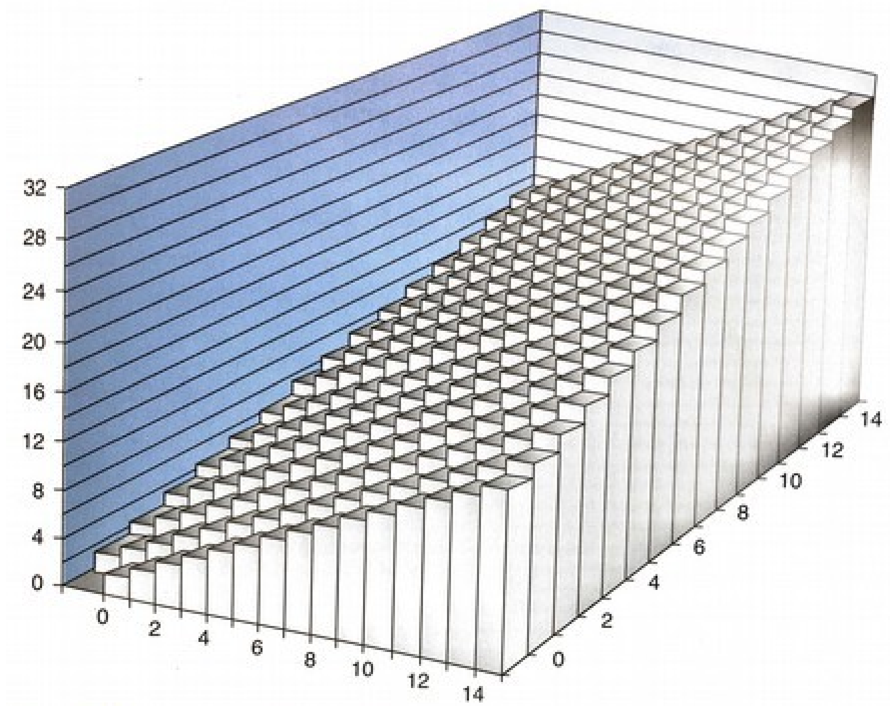


Figure 2.21 **Integer addition.** With a 4-bit word size, the sum could require 5 bits.

Overflow

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

```
   1
    994f
+   7120
   10a6f = 0a6f
```

Truncation!

(assume a 16-bit integer)

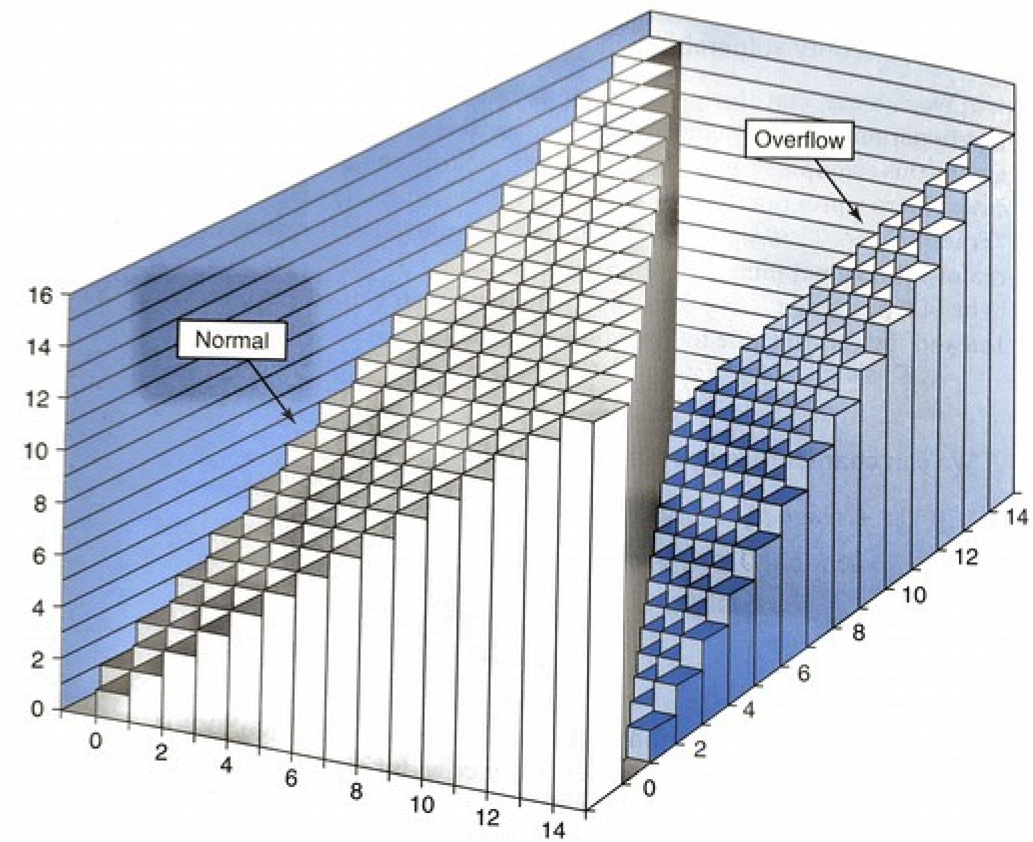
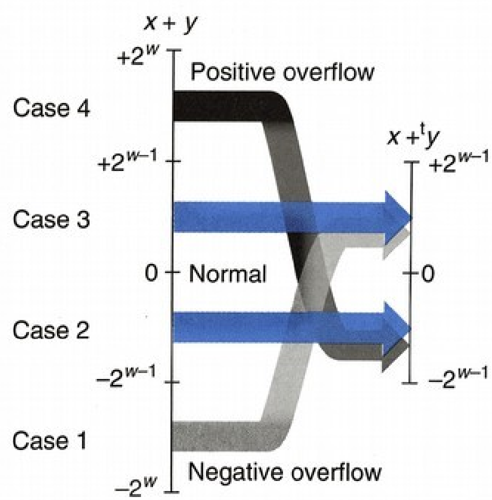


Figure 2.23 Unsigned addition. With a 4-bit word size, addition is performed modulo 16.

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
  - Same for subtraction (overflows if borrow-in and borrow-out of sign bit differ)

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.



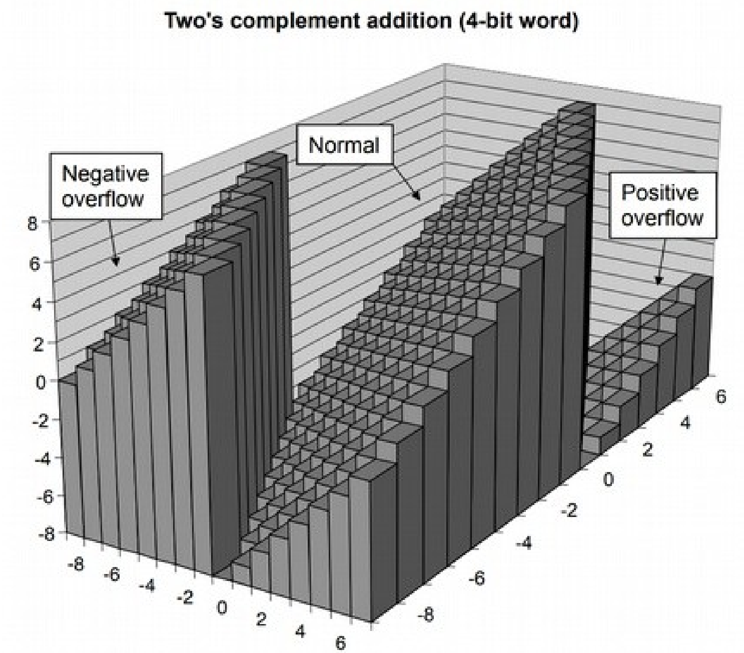


Figure 2.26 **Two’s-complement addition.** With a 4-bit word size, addition can have a negative overflow when $x + y < -8$ and a positive overflow when $x + y \geq 8$.

NOTE: this figure is printed incorrectly in your textbook!

Overflow

(sign bits in blue)

- Examples (in 4-bit two's complement):

```
    1            2's Comp.
  0 0 1 1           3
+ 0 0 1 0         + 2
  -------         ---
  0 1 0 1           5
```

No carry in, no carry out
(OK)

```
1 1              2's Comp.
  1 1 0 1          -3
+ 0 1 0 0         + 4
  -------         ---
  0 0 0 1           1
```

Carry in, carry out
(OK)

```
  1              2's Comp.
  0 1 0 1           5
+ 0 1 0 0         + 4
  -------         ---
  1 0 0 1          -7
```

Carry in, no carry out
(OVERFLOW!)

```
                 2's Comp.
  0 0 0 1           1
+ 1 1 1 0         + -2
  -------         ----
  1 1 1 1          -1
```

No carry in, no carry out
(OK)

```
1 1 1 2          2's Comp.
  0 0 0 1           1
- 0 0 1 0         -  2
  -------         ----
  1 1 1 1          -1
```

Borrow in, borrow out
(OK)

Observation: In two's complement, adding the inverse is equivalent to subtracting!

Case study: MTG Arena

- “Evra, Halcyon Witness”
  - Card from Magic: The Gathering Arena (PC video game)
  - **Ability**: gain player life equal to Evra’s power (“lifelink”)
  - **Ability**: exchange player life total w/ Evra’s power
  - Alternate abilities to double life every few turns
  - Overflows at ~2 billion b/c player life is stored as a signed 32-bit integer

https://www.youtube.com/watch?v=8cqID9lpC3I

Multiplication & division

```
 101 (5)
x_11 (3)
 101
101_
1111 (15)
```

- 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)

```
2 << 1 = 4        (2 * 2)
1 << 2 = 4        (1 * 2 * 2)

1 << 4 = 16       (1 * 2 * 2 * 2 * 2)
4 << 1 = 8        (4 * 2)
4 << 2 = 16       (4 * 2 * 2)
```

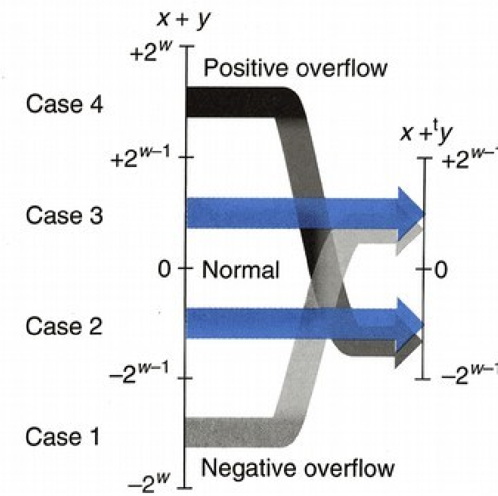
- Division is expensive!
  - Special case: divide by powers of two (shift right)
    - Logical shift for unsigned numbers, arithmetic shift for signed numbers

Review

- One-byte integers:

Binary	Unsigned	Two’s C
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

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.



Overflow when x + y > 255

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

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!)

$$101.101$$

1	0	1	.	1	0	1
4	2	1		1/2	1/4	1/8

4 + 1 + 0.5 + 0.125 = **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: ~2 x 10^{-9} through ~2 x 10^{9}

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

$$
\begin{aligned}
& 0.0123 \times 10^{2} \\
& 0.123 \times 10^{1} \\
1.23 \quad = \quad & 1.23 \times 10^{0} \\
& 12.3 \times 10^{-1} \\
& 123.0 \times 10^{-2}
\end{aligned}
$$

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

2.5 → 0 1000 010

Sign (+)

Significand: (1).01 = 2.5

Exponent (8 - 7 = 1)

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