Computer Arithmetic

Dianne P. O’Leary
AMSC 662 Notes

- What's in a word?
- Integers and arithmetic
- Floating point and arithmetic

Reference

Michael Overton,
Numerical Computing with IEEE Floating Point Arithmetic,
SIAM Press, 2001.

What's in a word?

Suppose a memory location contains:

0xc70425f20060

This could be an:

- Instruction: movl $0x6000e0, ...
- Character string
- Integer
- Floating point number
- Half of a double precision number
- . . .

What's in a word?

The context tells us how to interpret the bitstring.

In these notes, we'll usually work with integers and floating point numbers stored in a single word (32 bits) of memory.

The extension to double words (64 bits) should be clear.

And sometimes in examples we'll use very short words, for simplicity.

Integer arithmetic

Prerequisite: Binary numbers

You need to know, for example, that

$$1011_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$
$$= 11_{10}$$

and

$$0.1011_2 = 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 1 \times 2^{-4}$$
$$= (11/16)_{10}$$

Fixed Point: 2's complement Integers

Integers are stored as bitstrings in binary notation.

Positive integers: 1^{st} bit is zero, others give the magnitude.

0	0	1	0	1	1

= + 11

Negative integers: 2's complement means

1	0	1	0	1	1

$= -21$, since $2^6 - 21 = 101011_2$.

Key idea in 2's complement:

To change the sign of an integer represented by k bits:

- Subtract it from 2^k,
- Or, equivalently,
- Complement every bit and then add 1.

This makes arithmetic easy.

Example of binary addition

	0	0	0	1	1
+	0	1	0	1	0
	0	1	1	0	1

Note the “**carry**” here!

$0 + 0 = 0$

$0 + 1 = 1$

$1 + 0 = 1$

$1 + 1 = 10$ (binary) $= 10_2 = 2$

$1 + 1 + 1 = 11$ (binary)$=11_2 = 3$

Example of binary addition

	0	0	0	1	1
+	0	1	0	1	0
	0	1	1	0	1

In decimal notation, 3

+10

=13

Note the "**carry**" here!

2^{nd} Example of binary addition

	0	0	1	1	1
+	1	1	1	0	1
	0	0	1	0	0

In decimal notation, 7

- 3

= 4

Note the "**carry**" here!

Circuit design is easy with 2's complement!

Range of fixed point numbers

Largest 5-digit (5 bit) binary number:

0	1	1	1	1

= 15

Range of fixed point numbers

Largest 5-digit (5 bit) binary number:

0	1	1	1	1

= 15

Smallest:

Range of fixed point numbers

Largest 5-digit (5 bit) binary number: | 0 | 1 | 1 | 1 | 1 | = 15

Smallest: | 1 | 0 | 0 | 0 | 0 | = -16

Smallest positive:

Range of fixed point numbers

Largest 5-digit (5 bit) binary number:	0	1	1	1	1	= 15
Smallest:	1	0	0	0	0	= -16
Smallest positive:	0	0	0	0	1	= 1
There is only one representation for zero:	0	0	0	0	0	= 0

Overflow

If we try to add these numbers:

	0	1	1	1	1	= 15
+	0	1	0	0	0	= 8

we get

1	0	1	1	1	= -9.

We call this overflow: the answer is too large to store, since it is outside the range of this number system.

Features of fixed point arithmetic

Easy: always get an integer answer.

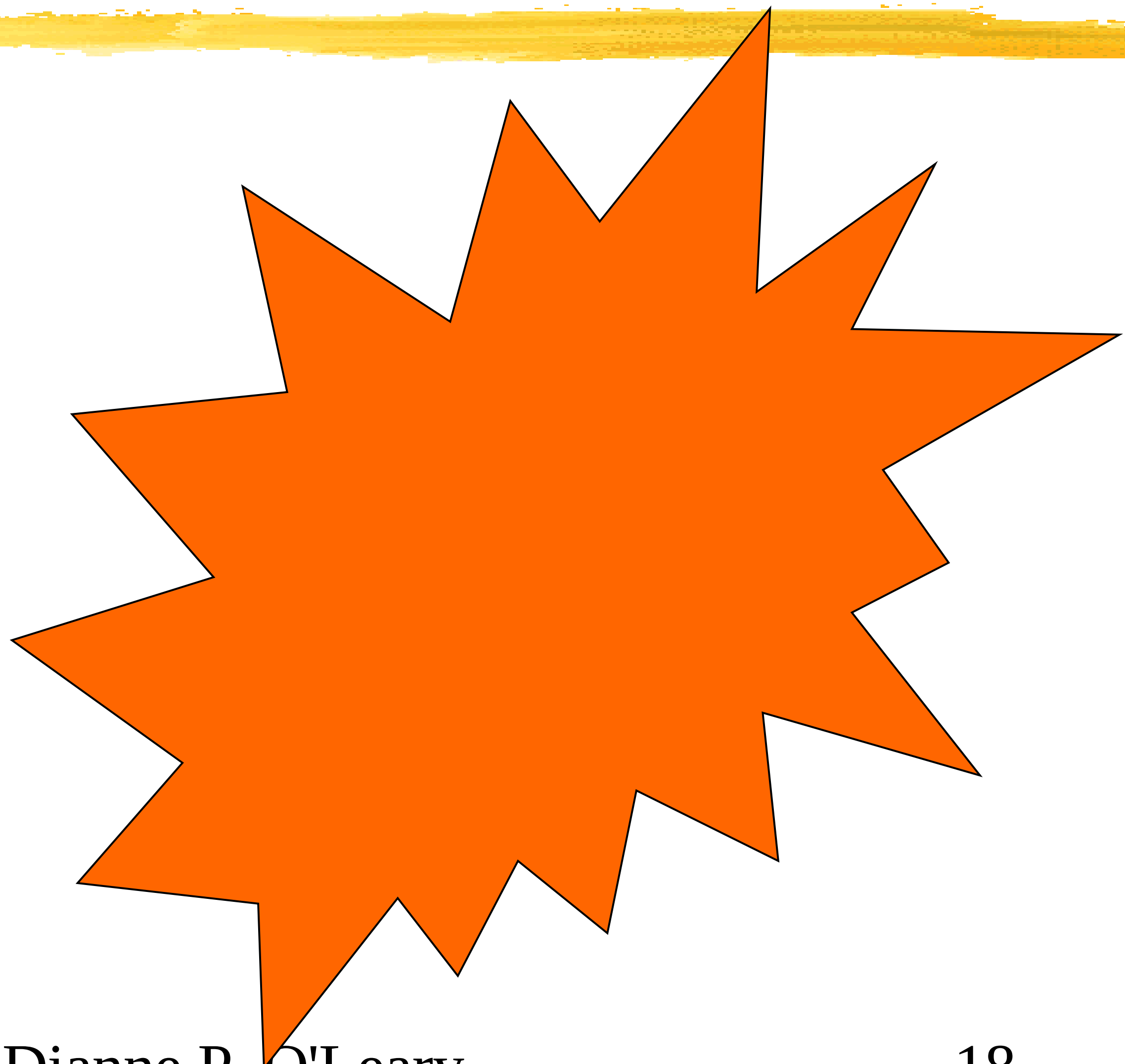
Representation is 2's complement (on most computers).

Either we get exactly the right answer for addition, subtraction, or multiplication, or we can detect overflow.

The numbers that we can store are equally spaced.

Disadvantage: **very** limited range of numbers.

Floating point arithmetic



Floating point arithmetic

If we wanted to store 15×2^{11} , we would need 16 bits:

0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0

Instead, let's agree to code numbers as **two** fixed point binary numbers:

$z \times 2^p$, with z = 15 saved as 01111
and p = 11 saved as 01011.

Now we can have fractions, too:

binary $0.101 = 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}$.

Floating point arithmetic

Jargon: z is called the **mantissa** or **significand**.
p is called the **exponent.**

$$\pm z \times 2^p$$

To make the representation unique (since, for example, $2 \times 2^1 = 4 \times 2^0$), we **normalize** to make $1 \le z < 2$.

We store d digits for the mantissa, and limit the range of the exponent to $m \le p \le M$, for some integers m and M.

Floating point representation

Example: Suppose we have a machine with $d = 5$, $m = -15$, $M = 15$.

$15 \times 2^{10} = 1111_2 \times 2^{10} = 1.111_2 \times 2^{13}$

mantissa $z = +1.1110$
exponent $p = +1101$

$15 \times 2^{-10} = 1111_2 \times 2^{-10} = 1.111_2 \times 2^{-7}$

mantissa $z = +1.1110$
exponent $p = -0111$

Floating point standard

Up until the mid-1980s, each computer manufacturer had a different choice for d, m, and M, and even a different way to select answers to arithmetic problems.

A program written for one machine often would not compute the same answers on other machines.

The situation improved somewhat with the introduction in 1987 of **IEEE standard** floating point arithmetic.

Floating point standard

On most machines today,

single precision: $d = 24$, $m = -126$, $M = 127$

double precision: $d = 53$, $m = -1022$, $M = 1023$.

The mantissa bits store the absolute value of the mantissa|, not the 2's complement representation. This is called sign-magnitude representation.

(And gradual underflow should be allowed (but isn't always).)

Sanity check

single precision: $d = 24$, $m = -126$, $M = 127$

So we need 24 bits for the mantissa and 8 for the exponent and 1 for the sign of the mantissa. 33 bits total.

This is one too many!

But since the mantissa is always 1 followed by a fractional part, we agree not to store the 1.

Floating point addition

Machine arithmetic is more complicated for floating point.

Example: In fixed point, we added 3 + 10.
Here it is in floating point:

3 = 11 (binary) = 1.100×2^1 z = 1.100, p = 1

10 = 1010 (binary) = 1.010×2^3 z = 1.010, p = 11.

1. Shift the smaller number so that the exponents are equal

   z = 0.0110 p = 11
2. Add the mantissas

   z = 0.0110 + 1.010 = 1.1010, p = 11
3. Shift if necessary to normalize.

Roundoff in floating point addition

Sometimes we cannot store the exact answer.

Example: $1.1001 \times 2^{0} + 1.0001 \times 2^{-1}$

1. Shift the smaller number so that the exponents are equal

   $z = 0.10001 \quad p = 0$

2. Add the mantissas

   $$\begin{array}{r} 0.10001 \\ +\ 1.1001 \\ = 10.00011, \quad p = 0 \end{array}$$

3. Shift if necessary to normalize: 1.000011×2^{1}

But we can only store 1.0000×2^{1}! The error is called **roundoff**.

Underflow, overflow....

Convince yourself that roundoff cannot occur in fixed point.

Other floating point troubles:

Overflow: exponent grows too large.

Underflow: exponent grows too small.

Range of floating point

Example: Suppose that d = 5 and
exponents range between -15 and 15.

Smallest positive normalized number: 1.0000 (binary) x 2^{-15}
(since mantissa needs to be normalized)

Largest positive number: 1.1111 (binary) x 2^{15}

Rounding

IEEE standard arithmetic uses rounding as its default.

Rounding: Store x as r, where r is the machine number nearest to x.

In order to have the result of arithmetic correctly rounded to d digits, we require up to 3 extra guard digits.

For example, $1.000 \times 2^{0} - 1.100 \times 2^{-5} = 0.111\ 101$

(rounded) $= 1.111 \times 2^{-1}$

The hardware we studied uses 80 bits for intermediate results.

An important number: machine epsilon

Machine epsilon is defined to be gap between 1 and the next larger number that can be represented exactly on the machine.

Example: Suppose that d = 5 and
exponents range between -15 and 15.

What is machine epsilon in this case?

Note: Machine epsilon depends on d, but does not depend on m or M!

Features of floating point arithmetic

- The numbers that we can store are **not** equally spaced. (Try to draw them on a number line.)

- A wide range of variably-spaced numbers can be represented exactly.

- For addition, subtraction, and multiplication, either we get exactly the right answer or a rounded version of it, or we can detect underflow or overflow.

An Important Detail: Zero should be zero!

We agree to bias the exponents by adding 127, so true exponents -126, -125, ..., 126, 127 map to 1, 2, ..., 253, 254.

- This allows us to use the biased exponent 0 when representing the number 0. Therefore, floating point zero has the same bit pattern as integer zero: 32 zeros.

- This helps implementation of the CPU's zero flag.

- But note that there are two floating point zeros: 000...000 and 100...000.

What about other mantissas with a biased exponent of zero?

- We can use these to store numbers too small to normalize. We agree that numbers with a nonzero mantissa and a biased exponent of 0 will be interpreted as the mantissa (with a leading 0, not a 1, before the binary point) times $2^{-126.}$

  Example: $0.01_2 \times 2^{-126}$. This is stored as

  exponent: 00000000

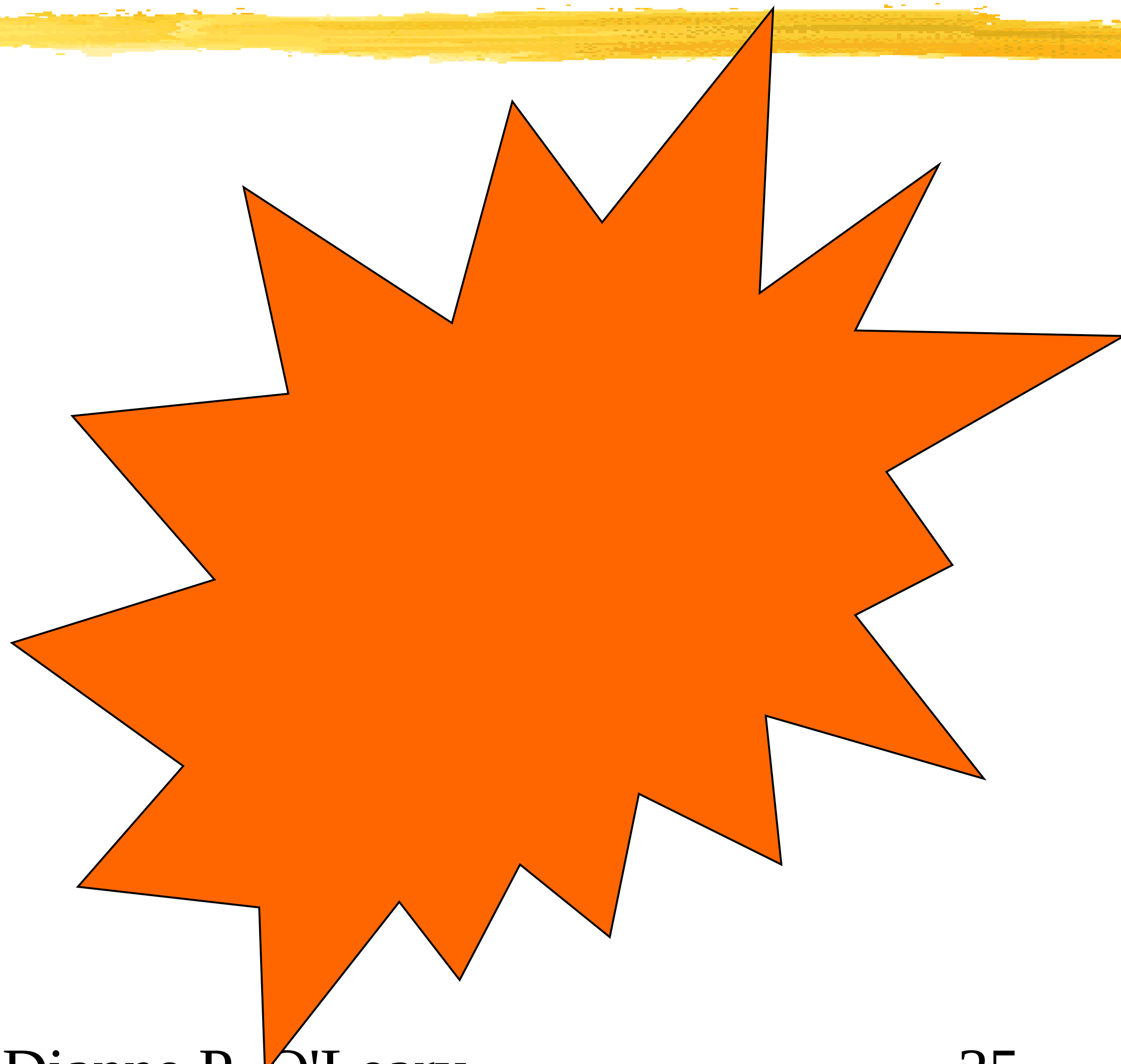
  mantissa: 01000000000000000000000

- This allows gradual underflow.

NaN and Inf

- We haven't used the (biased) exponent 255.
- We reserve it for special values:
- Exponent 255 and mantissa 0 means +Inf or -Inf.
- Exponent 255 and nonzero mantissa means NaN.

Summary



Summary

-- The meaning of the content of a computer word depends on context.

-- Computers use

- ✓ 2's complement to store integers, because it makes arithmetic fast.
- ✓ Sign-magnitude to store mantissas in floating point because adder speed is not critical
- ✓ Biased integers to store exponents in floating point, so that 000...000 means zero.