Floating Point
Objectives

- look at floating point representation in its basic form
- expose errors of a different form: rounding error
- highlight IEEE-754 standard
Why this is important:

• Errors come in two forms: truncation error and rounding error
  • we always have them ...
  • case study: Intel
  • our jobs as developers: reduce impact
Flaw Undermines Accuracy of Pentium Chips

By JOHN MARROFF

SAN FRANCISCO, Nov. 23 — A massive circuitry error is causing a chip used in millions of computers to generate erroneous results in certain divisions of problem-solving routines, swamping scientific and engineering work that relies on computers to handle precise calculations.

The flaw, in error division, has been found in the Pentium, the current top of the line of Intel Corp.'s microprocessor product line.

Intel declined to say how many Pentium chips had been sold, but Disappoant, a market research company in San Jose, Calif., estimated 60 million for the first six months of the year. 

The computer maker, in several different configurations, is used in many companies, sold for home and business use, including some made by I.B.M., Compaq, Dell, Gateway 2000 and other manufacturers.

Intel said yesterday that it did not believe the chip needed to be recalled, asserting that the typical user would be unlikely to detect a problem in its computer's operation. 

However, an Intel spokesman noted that the error could produce an inaccurate result as a consequence of the error, and then there was no noticeable consequence in the accuracy of business or home computer systems.

The company declined to say whether it was continuing to send computer makers Pentium chips built before the problem was disclosed.

William Kahn of the University of California, San Diego, one of the nation's experts on computer mathematics, expressed skepticism about Intel's conclusions that the error would only occur in extremely rare instances.

"These kinds of statistics have to come somewhere," he said in a telephone interview. "I don't think it is likely they have a complete understanding of the probability of errors whose probability we don't know.

At the Jet Propulsion Laboratory in Pasadena, Calif., one satellite communications expert said he was concerned but not alarmed by the discovery he learned of the error this week, and that it might eventually be blamed for errors in his group and their use had been suspended for now.

"The Pentium appeared as a constant in the equation," he said. "It's not likely to be involved in the future."

Gibson Suit
On Trades
Is Settled

Bankers Trust Gets
30% of Debt Claimed

By MICHAEL QUINT

Gibson Greetings Inc. and the Bankers Trust Company said yesterday that they had reached an out-of-court settlement of a lawsuit accusing the bank of improperly lending the company in engendering risky financial trades.

Under their agreement, Bankers Trust will pay Gibson $4.2 million, or about 38 percent of the $8.8 million that Bankers Trust contended was owed under the lawsuit.

While neither side blamed the other of victory, Douglas Kaid, a spokesman for Bankers Trust, acknowledged that...
Flaw Undermines Accuracy of Pentium Chips

By JOHN MARKOFF Special to The New York Times

Flaw Undermines Accuracy of Pentium Chips

SAN FRANCISCO, Nov. 23 — An elusive circuitry error is causing a chip used in millions of computers to generate inaccurate results in certain rare cases, heightening anxiety among many scientists and engineers who rely on their machines for precise calculations.

The flaw, an error in division, has been found in the Pentium, the current top microprocessor of the Intel Corporation, the world’s largest chip maker. The chip, in several different configurations, is used in many computers sold for home and business use, including those made by I.B.M., Compaq, Dell, Gateway 2000 and others.

The flaw appears in all Pentium chips now on the market, in certain types of division problems involving more than five significant digits, a mathematical term that can include numbers before and after a decimal point.

Intel declined to say how many Pentium chips it made or sold, but Dataquest, a market research company in San Jose, Calif., estimated that in 1994 Intel would sell 5.5 million to 6 million Pentiums, roughly 10 percent of the number of personal computers sold worldwide.

Intel said yesterday that it did not believe the chip needed to be recalled, asserting that the typical user would have but one chance in more than nine billion of encountering an inaccurate result as a consequence of the error, and thus there was no noticeable consequence to users of business or home computers. Indeed, the company said it was continuing to send computer makers Pentium chips built before the problem was detected.

William Kahan of the University of California at Berkeley, one of the nation’s experts on computer mathematics, expressed skepticism about Intel’s contentions that the error would only occur in extremely rare instances.

“These kinds of statistics have to cause some wonderment,” he said. “They are based on assertions about the probability of events whose probability we don’t know.”

At the Jet Propulsion Laboratory in Pasadena, Calif., one satellite communications researcher who learned of the error this week, said six Pentium machines were used in his group and their use had been suspended for now.

“The Pentium appeared as a cost-Continued on Page D5
Intel Chips

In some complex division problems, annoying errors.

corrected.
Some computer users said they believed that Intel had not acted quickly enough after discovering the error.
"Intel has known about this since the summer; why didn't they tell anyone?" said Andrew Schulman, the author of a series of technical books on PC's. "It's a hot issue, and I don't think they've handled this well.
The company said that after it discovered the problem this summer, it ran months of simulations of different applications, with the help of outside experts, to determine whether the problem was serious.
The Pentium error occurs in a portion of the chip known as the floating point unit, which is used for extremely precise computations. In rare cases, the error shows up in the result of a division operation.
Intel said the error occurred because of an omission in the translation of a formula into computer

Close, but Not Close Enough

The owners of computers that use Intel's Pentium microprocessors have found that the chips sometimes do not perform division calculations accurately enough.
The problems arise when the chip has to round a number in a preliminary calculation to get the final result, a task that all processors normally perform. In these cases, however, the Pentium's figures are exact to only 5 digits, not 16, as are those of other computer processors. The Pentium's error, while small, can be 10 billion times as large as those of most chips.
Here is an example of the way the imprecise rounding changes the result of a calculation and the way the deviation from the expected result is calculated:

\[
\text{PROBLEM} \\
\text{CORRECT CALCULATION} \\
= 4,195,835 - [(1.3338204) \times 3,145,727] = 0 \\
\text{PENTIUM'S CALCULATION} \\
= 4,195,835 - [(1.3337391) \times 3,145,727] = 256 \\
\text{DEVIATION} \\
256 + 4,195,835 = 6.1 \times 10^{-5}, \text{ or } 61/100,000
\]

Source: Cleve Moler, the Mathworks Inc.
June 1994  Intel engineers discover the division error. Managers decide the error will not impact many people. Keep the issue internal.

June 1994  Dr Nicely at Lynchburg College notices computation problems

Oct 19, 1994  After months of testing, Nicely confirms that other errors are not the cause. The problem is in the Intel Processor.


Oct 30, 1994  After no action from Intel, Nicely sends an email
TO: Whom it may concern

RE: Bug in the Pentium FPU

DATE: 30 October 1994

It appears that there is a bug in the floating point unit (numeric coprocessor) of many, and perhaps all, Pentium processors.

In short, the Pentium FPU is returning erroneous values for certain division operations. For example,

0001/824633702441.0

is calculated incorrectly (all digits beyond the eighth significant digit are in error). This can be verified in compiled code, an ordinary spreadsheet such as Quattro Pro or Excel, or even the Windows calculator (use the scientific mode), by computing

00(824633702441.0)*(1/824633702441.0),

which should equal 1 exactly (within some extremely small rounding error; in general, coprocessor results should contain 19 significant decimal digits). However, the Pentiums tested return

0000.999999996274709702...
Intel Timeline

Nov 1, 1994  Software company Phar Lap Software receives Nicely’s email. Sends to colleagues at Microsoft, Borland, Watcom, etc. decide the error will not impact many people. Keep the issue internal.

Nov 2, 1994  Email with description goes global.


Nov 22, 1994  CNN Moneyline interviews Intel. Says the problem is minor.


Dec 12, 1994  IBM halts shipment of Pentium based PCs

Dec 16, 1994  Intel stock falls again.
Numerical "bugs"

**Obvious**
Software has bugs

**Not-SO-Obvious**
Numerical software has two unique bugs:

1. roundoff error
2. truncation error
Roundoff

Roundoff occurs when digits in a decimal point (0.3333...) are lost (0.3333) due to a limit on the memory available for storing one numerical value.

Truncation

Truncation error occurs when discrete values are used to approximate a mathematical expression.
Errors in input data can cause *uncertain* results

- input data can be experimental or rounded, leading to a certain variation in the results
- **well-conditioned**: numerical results are insensitive to small variations in the input
- **ill-conditioned**: small variations lead to drastically different numerical calculations (a.k.a. poorly conditioned)
As numerical analysts, we need to

1. solve a problem so that the calculation is not susceptible to large roundoff error
2. solve a problem so that the approximation has a *tolerable* truncation error

How?

• incorporate roundoff-truncation knowledge into
  • the mathematical model
  • the method
  • the algorithm
  • the software design
• awareness → correct interpretation of results
Wanted: Real Numbers... in a computer

Computers can represent integers, using bits:

\[ 23 = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = (10111)_2 \]

How would we represent fractions, e.g. 23.625?

Idea:

Keep going down past zero exponent:

\[ 23.625 = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} \]

So:

Could store

• a fixed number of bits with exponents \( \geq 0 \)
• a fixed number of bits with exponents \( < 0 \)

This is called fixed-point arithmetic.
Computers can represent integers, using bits:

\[ 23 = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = (10111)_2 \]

How would we represent fractions, e.g. 23.625?

Idea: Keep going down past zero exponent:

\[ 23.625 = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} \]

So: Could store

- a fixed number of bits with exponents \( \geq 0 \)
- a fixed number of bits with exponents \( < 0 \)

This is called fixed-point arithmetic.
Suppose we use units of 64 bits, with 32 bits for exponents $\geq 0$ and 32 bits for exponents $< 0$. What numbers can we represent?
Suppose we use units of 64 bits, with 32 bits for exponents $\geq 0$ and 32 bits for exponents $< 0$. What numbers can we represent?

\[
\begin{array}{cccccc}
2^{31} & \ldots & 2^0 & 2^{-1} & \ldots & 2^{-32}
\end{array}
\]
Suppose we use units of 64 bits, with 32 bits for exponents $\geq 0$ and 32 bits for exponents $< 0$. What numbers can we represent?

\[
\begin{array}{cccccc}
2^{31} & \ldots & 2^0 & 2^{-1} & \ldots & 2^{-32} \\
\end{array}
\]

**Smallest:** $2^{-32} \approx 10^{-10}$

**Largest:** $2^{31} + \cdots + 2^{-32} \approx 10^9$
How many ‘digits’ of relative accuracy (think relative rounding error) are available for the smallest vs. the largest number with 64 bits of fixed point?

\[
\begin{array}{cccccc}
2^{31} & \ldots & 2^0 & 2^{-1} & \ldots & 2^{-32}
\end{array}
\]
How many ‘digits’ of relative accuracy (think relative rounding error) are available for the smallest vs. the largest number with 64 bits of fixed point?

\[
\begin{array}{cccc}
2^{31} & \ldots & 2^0 & 2^{-1} & \ldots & 2^{-32} \\
\end{array}
\]

For large numbers: about 19

For small numbers: few or none

Idea: Don’t fix the location of the 0 exponent. Let it float.
Normalized Floating-Point Representation
Real numbers are stored as

\[ x = \pm (0.d_1d_2d_3\ldots d_m)_\beta \times \beta^e \]

- \( d_1d_2d_3\ldots d_m \) is the significand, \( e \) is the exponent
- \( e \) is negative, positive or zero
- the general normalized form requires \( d_1 \neq 0 \)
Example

In base 10

- 1000.12345 can be written as
  \[(0.100012345)_{10} \times 10^4\]

- 0.000812345 can be written as
  \[(0.812345)_{10} \times 10^{-3}\]
Floating Point

Suppose we have only 3 bits for a (non-normalized) significand and a 1 bit exponent stored like

\[
\begin{array}{cccc}
. & d_1 & d_2 & d_3 & e_1 \\
\end{array}
\]

All possible combinations give:

\[
000_2 = 0 \\
\cdots \times 2^{-1,0,1} \\
111_2 = 7
\]

So we get \(0, \frac{1}{16}, \frac{2}{16}, \ldots, \frac{7}{16}, 0, \frac{1}{4}, \frac{2}{4}, \ldots, \frac{7}{4}, \) and \(0, \frac{1}{8}, \frac{2}{8}, \ldots, \frac{7}{8}\). On the real line:
• computations too close to zero may result in **underflow**
• computations too large may result in **overflow**
• overflow error is considered more severe
• underflow can just fall back to 0
Normalizing

If we use the normalized form in our 4-bit case, we lose $0.001_2 \times 2^{-1,0,1}$ along with other. So we cannot represent $\frac{1}{16}$, $\frac{1}{8}$, and $\frac{3}{16}$.
• We’re familiar with base 10 representation of numbers:

\[ 1234 = 4 \times 10^0 + 3 \times 10^1 + 2 \times 10^2 + 1 \times 10^3 \]

and

\[ .1234 = 1 \times 10^{-1} + 2 \times 10^{-2} + 3 \times 10^{-3} + 4 \times 10^{-4} \]

• we write 1234.1234 as an integer part and a fractional part:

\[ a_3a_2a_1a_0.b_1b_2b_3b_4 \]

• For some (even simple) numbers, there may be an infinite number of digits to the right:

\[ \pi = 3.14159 \ldots \]
\[ 1/9 = 0.11111 \ldots \]
\[ \sqrt{2} = 1.41421 \ldots \]
Other bases

- So far, we have just base 10. What about base $\beta$?
- binary ($\beta = 2$), octal ($\beta = 8$), hexadecimal ($\beta = 16$), etc
- In the $\beta$-system we have

$$ (a_n \ldots a_2 a_1 a_0 . b_1 b_2 b_3 b_4 \ldots)_\beta = \sum_{k=0}^{n} a_k \beta^k + \sum_{k=0}^{\infty} b_k \beta^{-k} $$
An algorithm to compute the base 2 representation of a base 10 integer

\[(N)_{10} = (a_ja_{j-1} \ldots a_2a_0)_2\]

\[= a_j \cdot 2^j + \cdots + a_1 \cdot 2^1 + a_0 \cdot 2^0\]

Compute \((N)_{10}/2 = Q + R/2:\)

\[\frac{N}{2} = a_j \cdot 2^{j-1} + \cdots + a_1 \cdot 2^0 + \frac{a_0}{2}\]

\[= Q + \frac{R}{2}\]

**Example**

Example: compute \((11)_{10}\) base 2

\[
\begin{align*}
11/2 &= 5R1 \quad \Rightarrow \quad a_0 = 1 \\
5/2 &= 2R1 \quad \Rightarrow \quad a_1 = 1 \\
2/2 &= 1R0 \quad \Rightarrow \quad a_2 = 0 \\
1/2 &= 0R1 \quad \Rightarrow \quad a_3 = 1
\end{align*}
\]

So \((11)_{10} = (1011)_2\)
The other way...

Convert a base-2 number to base-10:

\[(11 \, 000 \, 101)_2\]

\[= 1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0\]

\[= 1 + 2(0 + 2(1 + 2(0 + 2(0 + 2(0 + 2(1 + 2(1)))))))\]

\[= 197\]
Converting fractions

• straightforward way is not easy
• goal: for $x \in [0, 1]$ write

$$x = 0.b_1b_2b_3b_4\cdots = \sum_{k=1}^{\infty} c_k\beta^{-k} = (0.c_1c_2c_3\ldots)_\beta$$

• $\beta(x) = (c_1.c_2c_3c_4\ldots)_\beta$
• multiplication by $\beta$ in base-$\beta$ only shifts the radix
Fraction Algorithm

An algorithm to compute the binary representation of a fraction $x$:

$$x = 0.b_1b_2b_3b_4\ldots$$
$$= b_1 \cdot 2^{-1} + \ldots$$

Multiply $x$ by 2. The integer part of $2x$ is $b_1$

$$2x = b_1 \cdot 2^0 + b_2 \cdot 2^{-1} + b_3 \cdot 2^{-2} + \ldots$$

**Example**

Example: Compute the binary representation of 0.625

$$2 \cdot 0.625 = 1.25 \quad \Rightarrow \quad b_{-1} = 1$$
$$2 \cdot 0.25 = 0.5 \quad \Rightarrow \quad b_{-2} = 0$$
$$2 \cdot 0.5 = 1.0 \quad \Rightarrow \quad b_{-3} = 1$$

So $(0.625)_{10} = (0.101)_2$
A problem with precision

\[ r_0 = x \]
\[ \text{for } k = 1, 2, \ldots, m \]
\[ \text{if } r_{k-1} \geq 2^{-k} \]
\[ b_k = 1 \]
\[ r_k = r_{k-1} - 2^{-k} \]
\[ \text{else} \]
\[ b_k = 0 \]
\[ \text{end} \]
\[ \text{end} \]

<table>
<thead>
<tr>
<th>( k )</th>
<th>( 2^{-k} )</th>
<th>( b_k )</th>
<th>( r_k = r_{k-1} - b_k 2^{-k} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.8125</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0.5</td>
<td>1</td>
<td>0.3125</td>
</tr>
<tr>
<td>2</td>
<td>0.25</td>
<td>1</td>
<td>0.0625</td>
</tr>
<tr>
<td>3</td>
<td>0.125</td>
<td>0</td>
<td>0.0625</td>
</tr>
<tr>
<td>4</td>
<td>0.0625</td>
<td>1</td>
<td>0.0000</td>
</tr>
</tbody>
</table>
Floating Point numbers

Convert \(13.5 = (1101.1)_2\) into floating point representation.
Convert $13.5 = (1101.1)_2$ into floating point representation.

$$13.5 = 2^3 + 2^2 + 2^0 + 2^{-1} = (1.1011)_2 \cdot 2^3$$
Convert $13.5 = (1101.1)_2$ into floating point representation.

\[
13.5 = 2^3 + 2^2 + 2^0 + 2^{-1} = (1.1011)_2 \cdot 2^3
\]

What pieces do you need to store an FP number?
Convert $13.5 = (1101.1)_2$ into floating point representation.

\[ 13.5 = 2^3 + 2^2 + 2^0 + 2^{-1} = (1.1011)_2 \cdot 2^3 \]

What pieces do you need to store an FP number?

**Significand:** $(1.1011)_2$

**Exponent:** 3

**Idea:** Notice that the leading digit (in binary) of the significand is always one. Only store ‘1011’.
Floating Point numbers

Final storage format:

\[ 13.5 = 2^3 + 2^2 + 2^0 + 2^{-1} = (1.1011)_2 \cdot 2^3 \]

**Significand:** 1011 – a fixed number of bits

**Exponent:** 3 – a *(signed!)* integer allowing a certain range

Exponent is most often stored as a positive ‘offset’ from a certain negative number. E.g.

\[ 3 = -1023 + 1026 \]

 implicit offset  stored

Actually stored: 1026, a positive integer.
Can you think of a somewhat central number that we cannot represent as

\[ x = (1.\_\_\_\_\_\_\_\_)_2 \cdot 2^{-p}? \]
Can you think of a somewhat central number that we cannot represent as

\[ x = (1.\underline{\phantom{0}})_{2} \cdot 2^{-p}? \]

Zero. Which is somewhat embarrassing.

**Core problem:** The implicit 1. It’s a great idea, were it not for this issue.
Unrepresentable numbers?

Have to break the pattern. **Idea:**

- Declare one exponent ‘special’, and turn off the leading one for that one.
  (say, -1023, a.k.a. stored exponent 0)
- For all larger exponents, the leading one remains in effect.

**Bonus Q:** With this convention, what is the binary representation of a zero?
Web link: IEEE 754

Web link: What Every Computer Scientist Should Know About Floating-Point Arithmetic
IEEE-754: Why?!?!?

- IEEE-754 is a widely used standard accepted by hardware/software makers
  - defines the floating point distribution for our computation
  - offer several rounding modes which effect accuracy
- Floating point arithmetic emerges in nearly every piece of code
  - even modest mathematical operation yield loss of significant bits
  - several pitfalls in common mathematical expressions
IEEE Floating Point (v. 754)

- How much storage do we actually use in practice?
- 32-bit word lengths are typical
- IEEE Single-precision floating-point numbers use 32 bits
- IEEE Double-precision floating-point numbers use 64 bits
- Goal: use the 32-bits to best represent the normalized floating point number
IEEE Single Precision (Marc-32)

\[ x = \pm q \times 2^m \]

- 1-bit sign
- 8-bit exponent \(|m|\)
- 23-bit fraction \(q\)
- The leading one in the normalized significand \(q\) does not need to be represented: \(b_1 = 1\) is hidden bit
- IEEE 754: put \(x\) in 1.f normalized form
- \(0 < m + 127 = c < 255\)
- Largest exponent = 127, Smallest exponent = \(-126\)
- Special cases: \(c = 0, 255\)
IEEE Single Precision

\[ x = \pm q \times 2^m \]

Process for \( x = -52.125 \):

1. Convert both integer and fractional to binary:
   \[ x = -(110100.001000000000)_2 \]

2. Convert to 1.f form:
   \[ x = -1.101000010000 \ldots 0_2 \times 2^5 \]

3. Convert exponent
   \[ 5 = c - 127 \Rightarrow c = 132 \Rightarrow c = (10000100)_2 \]

\[ 1 \text{ 10000100 101000010000 \ldots 0} \]
IEEE Single Precision

Special Cases:

- denormalized/subnormal numbers: use 1 extra bit in the significant: exponent is now $-126$ (less precision, more range), indicated by $00000000_2$ in the exponent field
- two zeros: $+0$ and $-0$ (0 fraction, 0 exponent)
- two $\infty$’s: $+\infty$ and $-\infty$
- $\infty$ (0 fraction, $11111111_2$ exponent)
- NaN (any fraction, $11111111_2$ exponent)
IEEE Double Precision

• 1-bit sign
• 11-bit exponent
• 52-bit fraction
• single-precision: about 6 decimal digits of precision
• double-precision: about 15 decimal digits of precision
• $m = c - 1023$
## Precision vs. Range

<table>
<thead>
<tr>
<th>type</th>
<th>range</th>
<th>approx range</th>
</tr>
</thead>
<tbody>
<tr>
<td>single</td>
<td>$-3.40 \times 10^{38} \leq x \leq -1.18 \times 10^{-38}$</td>
<td>$2^{-126} \rightarrow 2^{128}$</td>
</tr>
<tr>
<td></td>
<td>$0$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$1.18 \times 10^{-38} \leq x \leq 3.40 \times 10^{38}$</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td>$-1.80 \times 10^{318} \leq x \leq -2.23 \times 10^{-308}$</td>
<td>$2^{-1022} \rightarrow 2^{1024}$</td>
</tr>
<tr>
<td></td>
<td>$0$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$2.23 \times 10^{-308} \leq x \leq 1.80 \times 10^{308}$</td>
<td></td>
</tr>
</tbody>
</table>

**small numbers example**
What is the smallest representable number in an FP system with 4 stored bits in the significand and an exponent range of \([-7, 7]\)?
What is the smallest representable number in an FP system with 4 stored bits in the significand and an exponent range of \([-7, 7]\)?

First attempt:

- Significand as small as possible $\rightarrow$ all zeros after the implicit leading one
- Exponent as small as possible: $-7$

$$
(1.0000)_2 \cdot 2^{-7}.
$$

Unfortunately: **wrong**. We can go way smaller by using the special exponent (which turns off the implicit leading one).
We’ll assume that the special exponent is $-8$. So:

$$(0.0001)_2 \cdot 2^{-7}$$

Numbers with the special exponent are called subnormal (or denormal) FP numbers. Technically, zero is also a subnormal.

**Note:** It is thus quite natural to ‘park’ the special exponent at the low end of the exponent range.
Why would you want to know about subnormals? Because computing with them is often slow, because it is implemented using ‘FP assist’, i.e. not in actual hardware. Many C compilers support options to ‘flush subnormals to zero’.
Why would you want to know about subnormals? Because computing with them is often slow, because it is implemented using ‘FP assist’, i.e. not in actual hardware. Many C compilers support options to ‘flush subnormals to zero’.

• FP systems without subnormals will **underflow** (return 0) as soon as the exponent range is exhausted.
• This smallest representable **normal** number is called the **underflow level**, or **UFL**.
Subnormal Numbers

• Beyond the underflow level, subnormals provide for **gradual underflow** by ‘keeping going’ as long as there are bits in the significand, but it is important to note that subnormals don’t have as many accurate digits as normal numbers.

• Analogously (but much more simply—no ‘supernormals’): the overflow level, **OFL**.
To summarize: To translate a stored (double precision) floating point value consisting of the stored fraction (a 52-bit integer) and the stored exponent value $e_{\text{stored}}$ the into its (mathematical) value, follow the following method:

$$\text{value} = \begin{cases} 
(1 + \text{fraction} \cdot 2^{-52}) \cdot 2^{-1023+e_{\text{stored}}} & e_{\text{stored}} \neq 0, \\
(\text{fraction} \cdot 2^{-52}) \cdot 2^{-1022} & e_{\text{stored}} = 0.
\end{cases}$$
A problem with precision

For other numbers, such as $\frac{1}{5} = 0.2$, an infinite length is needed.

$$0.2 \rightarrow .0011\ 0011\ 0011\ \ldots$$

So 0.2 is stored just fine in base-10, but needs infinite number of digits in base-2

!!!

This is *roundoff* error in its basic form...
What is the relative error produced by working with floating point numbers?

What is smallest floating point number > 1? Assume 4 stored bits in the significand.
What is the relative error produced by working with floating point numbers?

What is smallest floating point number $> 1$? Assume 4 stored bits in the significand.

$$(1.0001)_2 \cdot 2^0 = x \cdot (1 + 0.0001)_2$$
What is the relative error produced by working with floating point numbers?

What is smallest floating point number > 1? Assume 4 stored bits in the significand.

\[(1.0001)_2 \cdot 2^0 = x \cdot (1 + 0.0001)_2\]

What’s the smallest FP number > 1024 in that same system?
Floating Point and Rounding Error

What is the relative error produced by working with floating point numbers?

What is smallest floating point number $> 1$? Assume 4 stored bits in the significand.

$$(1.0001)_2 \cdot 2^0 = x \cdot (1 + 0.0001)_2$$

What’s the smallest FP number $> 1024$ in that same system?

$$(1.0001)_2 \cdot 2^{10} = x \cdot (1 + 0.0001)_2$$

Can we give that number a name?
Unit roundoff or machine precision or machine epsilon or $\varepsilon_{\text{mach}}$ is a bound on the relative error in a floating point representation for a given rounding procedure.

With chopping as the rounding procedure, it is the smallest number such that

$$\text{float}(1 + \varepsilon) \neq 1.$$

In the above system, $\varepsilon_{\text{mach}} = (0.0001)_2$. Another related quantity is ULP, or unit in the last place.

It is important to note, that if chopping is used, then $\varepsilon_{\text{mach}}$ is the distance between 1.0 and the next largest floating point number.
With chopping, take \( x = 1.0 \) and add \( 1/2, 1/4, \ldots, 2^{-i} \):

<table>
<thead>
<tr>
<th>Hidden bit</th>
<th>52 bits</th>
<th>→</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 0 0 0 0 0 0 0 0</td>
<td>e e</td>
<td></td>
</tr>
<tr>
<td>1 0 1 0 0 0 0 0 0 0 0 0 0</td>
<td>e e</td>
<td></td>
</tr>
<tr>
<td>1 0 0 1 0 0 0 0 0 0 0 0 0</td>
<td>e e</td>
<td></td>
</tr>
<tr>
<td>1 0 0 1 0 0 0 0 0 0 0 0 0</td>
<td>e e</td>
<td></td>
</tr>
</tbody>
</table>

• Ooops!
• use \( fl(x) \) to represent the floating point machine number for the real number \( x \)
• \( fl(1 + 2^{-52}) \neq 1 \), but \( fl(1 + 2^{-53}) = 1 \)
Machine epsilon $\epsilon_m$ is the smallest number such that

$$fl(1 + \epsilon_m) \neq 1$$

With rounding-to-nearest,

- The double precision machine epsilon is about $2^{-52}$.
- The single precision machine epsilon is about $2^{-23}$. 
Floating Point Number Line

overflow  usable range  under-flow  under-flow  usable range  overflow

\(-10^{+308}\)
\(-10^{-308}\)
\(-realmax\)
\(-realmin\)
\(0\)
\(10^{-308}\)
\(realmin\)
\(10^{+308}\)
\(realmax\)

zoom-in view
Floating Point Errors

- Not all reals can be exactly represented as a machine floating point number. Then what?
- Round-off error
- IEEE options:
  - Round to next nearest FP (preferred), Round to 0, Round up, and Round down

Let $x_+$ and $x_-$ be the two floating point machine numbers closest to $x$

- round to nearest: $\text{round}(x) = x_-$ or $x_+$, whichever is closest
- round toward 0: $\text{round}(x) = x_-$ or $x_+$, whichever is between 0 and $x$
- round toward $-\infty$ (down): $\text{round}(x) = x_-$
- round toward $+\infty$ (up): $\text{round}(x) = x_+$
What does this say about the relative error incurred in floating point calculations?

• The factor to get from one FP number to the next larger one is (mostly) independent of magnitude: $1 + \epsilon_{\text{mach}}$.

• Since we can't represent any results between $x$ and $x \cdot (1 + \epsilon_{\text{mach}})$, that's really the minimum error incurred.

• In terms of relative error:

$$\frac{|\tilde{x} - x|}{|x|} = |x(1 + \epsilon_{\text{mach}}) - x| = \epsilon_{\text{mach}}.$$

At least theoretically, $\epsilon_{\text{mach}}$ is the maximum relative error in any FP operations. (Practical implementations do fall short of this.)
What does this say about the relative error incurred in floating point calculations?

- The factor to get from one FP number to the next larger one is (mostly) independent of magnitude: $1 + \varepsilon_{\text{mach}}$.
- Since we can’t represent any results between $x$ and $x \cdot (1 + \varepsilon_{\text{mach}})$, that’s really the minimum error incurred.
- In terms of relative error:
  \[
  \left| \frac{\tilde{x} - x}{x} \right| = \left| \frac{x(1 + \varepsilon_{\text{mach}}) - x}{x} \right| = \varepsilon_{\text{mach}}.
  \]

At least theoretically, $\varepsilon_{\text{mach}}$ is the maximum relative error in any FP operations. (Practical implementations do fall short of this.)
What’s that same number for double-precision floating point? (52 bits in the significand)
What’s that same number for double-precision floating point? (52 bits in the significand)

\[ 2^{-52} \approx 10^{-16} \]

We can expect FP math to consistently introduce relative errors of about \(10^{-16}\).

Working in double precision gives you about 16 (decimal) accurate digits.
How is floating point addition implemented? Consider adding $a = (1.101)_2 \cdot 2^1$ and $b = (1.001)_2 \cdot 2^{-1}$ in a system with three bits in the significand.
How is floating point addition implemented?
Consider adding \( a = (1.101)_2 \cdot 2^1 \) and \( b = (1.001)_2 \cdot 2^{-1} \) in a system with three bits in the significand.

Rough algorithm:

1. Bring both numbers onto a common exponent
2. Do grade-school addition from the front, until you run out of digits in your system.
3. Round result.

\[
a = 1.101 \cdot 2^1
\]
\[
b = 0.01001 \cdot 2^1
\]
\[
a + b \approx 1.111 \cdot 2^1
\]
Problems with FP Addition

What happens if you subtract two numbers of very similar magnitude?
As an example, consider $a = (1.1011)_2 \cdot 2^1$ and $b = (1.1010)_2 \cdot 2^1$. 
What happens if you subtract two numbers of very similar magnitude?
As an example, consider $a = (1.1011)_2 \cdot 2^1$ and $b = (1.1010)_2 \cdot 2^1$.

$$a = 1.1011 \cdot 2^1$$
$$b = 1.1010 \cdot 2^1$$
$$a - b \approx 0.0001???? \cdot 2^1$$

or, once we normalize,

$$1.???? \cdot 2^{-3}.$$
Problems with FP Addition

→ Machine fills them with its ‘best guess’, which is not often good.

This phenomenon is called **Catastrophic Cancellation**.
Floating Point Arithmetic

- Problem: The set of representable machine numbers is FINITE.
- So not all math operations are well defined!
- Basic algebra breaks down in floating point arithmetic

**Example**

\[ a + (b + c) \neq (a + b) + c \]
Floating Point Arithmetic

Rule 1.

\[ fl(x) = x(1 + \epsilon), \quad \text{where} \quad |\epsilon| \leq \epsilon_m \]

Rule 2.
For all operations \( \circ \) (one of +, −, *, /)

\[ fl(x \circ y) = (x \circ y)(1 + \epsilon_\circ), \quad \text{where} \quad |\epsilon_\circ| \leq \epsilon_m \]

Rule 3.
For +, * operations

\[ fl(a \circ b) = fl(b \circ a) \]

There were many discussions on what conditions/rules should be satisfied by floating point arithmetic. The IEEE standard is a
Consider the sum of 3 numbers: $y = a + b + c$.

Done as $fl(fl(a + b) + c)$

$$\begin{align*}
\eta &= fl(a + b) = (a + b)(1 + \epsilon_1) \\
y_1 &= fl(\eta + c) = (\eta + c)(1 + \epsilon_2) \\
&= [(a + b)(1 + \epsilon_1) + c](1 + \epsilon_2) \\
&= [(a + b + c) + (a + b)\epsilon_1](1 + \epsilon_2) \\
&= (a + b + c) \left[ 1 + \frac{a + b}{a + b + c} \epsilon_1(1 + \epsilon_2) + \epsilon_2 \right]
\end{align*}$$

So disregarding the high order term $\epsilon_1\epsilon_2$

$$fl(fl(a + b) + c) = (a + b + c)(1 + \epsilon_3) \quad \text{with} \quad \epsilon_3 \approx \frac{a + b}{a + b + c} \epsilon_1 + \epsilon_2$$
Floating Point Arithmetic

If we redid the computation as $y_2 = fl(a + fl(b + c))$ we would find

$$fl(a + fl(b + c)) = (a + b + c)(1 + \epsilon_4) \text{ with } \epsilon_4 \approx \frac{b + c}{a + b + c} \epsilon_1 + \epsilon_2$$

Main conclusion:

The first error is amplified by the factor $(a + b)/y$ in the first case and $(b + c)/y$ in the second case.

In order to sum $n$ numbers more accurately, it is better to start with the small numbers first. [However, sorting before adding is usually not worth the cost!]
One of the most serious problems in floating point arithmetic is that of cancellation. If two large and close-by numbers are subtracted the result (a small number) carries very few accurate digits (why?). This is fine if the result is not reused. If the result is part of another calculation, then there may be a serious problem.

**Example**
Roots of the equation

\[ x^2 + 2px - q = 0 \]

Assume we want the root with smallest absolute value:

\[ y = -p + \sqrt{p^2 + q} = \frac{q}{\sqrt{p^2 + q}} \]
Catastrophic Cancellation

Adding \( c = a + b \) will result in a large error if

- \( a \gg b \)
- \( a \ll b \)

Let

\[
\begin{align*}
a &= x.xxx \cdots \times 10^0 \\
b &= y.yyy \cdots \times 10^{-8}
\end{align*}
\]

Then

\[
\begin{array}{c}
\text{finite precision} \\
\hline \\
x.xxx \text{ xxx} xxx \xxx \xxx xxx
\end{array}
\]

\[
\begin{array}{c}
\text{lost precision} \\
\hline \\
???? ????
\end{array}
\]

\[
\begin{array}{c}
\text{Then} \\
\hline \\
+ 0.000 0000 yyy \text{ yyy} \\
= x.xxx \xxx zzzz zzzz
\end{array}
\]
Subtracting $c = a - b$ will result in large error if $a \approx b$. For example

$$a = x.\ldots ss\ldots$$

$$b = x.\ldots tt\ldots$$

Then

$$\begin{align*}
\underbrace{x.\ldots}_{\text{finite precision}} + x.\ldots & = 0.000\ 0000\ 0001 \\
& \underbrace{\ldots????\ldots}_{\text{lost precision}}
\end{align*}$$
Summary

- addition: \( c = a + b \) if \( a \gg b \) or \( a \ll b \)
- subtraction: \( c = a - b \) if \( a \approx b \)
- catastrophic: caused by a single operation, not by an accumulation of errors
- can often be fixed by mathematical rearrangement
Example

$x = 0.37214\, 48693$ and $y = 0.37202\, 14371$. What is the relative error in $x - y$ in a computer with 5 decimal digits of accuracy?

\[
\frac{|x - y - (\bar{x} - \bar{y})|}{|x - y|} = \frac{|0.37214\, 48693 - 0.37202\, 14371 - 0.37214 + 0.37202\, 14371|}{|0.37214\, 48693 - 0.37202\, 14371|}
\approx 3 \times 10^{-2}
\]
Loss of Precision Theorem

Let \( x \) and \( y \) be (normalized) floating point machine numbers with \( x > y > 0 \).

If \( 2^{-p} \leq 1 - \frac{y}{x} \leq 2^{-q} \) for positive integers \( p \) and \( q \), the significant binary digits lost in calculating \( x - y \) is between \( q \) and \( p \).
Loss of Significance

Example
Consider \( x = 37.593621 \) and \( y = 37.584216 \).

\[
2^{-11} < 1 - \frac{y}{x} = 0.0002501754 < 2^{-12}
\]

So we lose 11 or 12 bits in the computation of \( x - y \). yikes!

Example
Back to the other example (5 digits): \( x = 0.37214 \) and \( y = 0.37202 \).

\[
10^{-4} < 1 - \frac{y}{x} = 0.00032 < 10^{-5}
\]

So we lose 4 or 5 bits in the computation of \( x - y \). Here,

\( x - y = 0.00012 \) which has only 1 significant digit that we can be sure about.
Loss of Significance

So what to do? Mainly rearrangement.

\[ f(x) = \sqrt{x^2 + 1} - 1 \]
So what to do? Mainly rearrangement.

\[ f(x) = \sqrt{x^2 + 1} - 1 \]

Problem at \( x \approx 0 \).
So what to do? Mainly rearrangement.

\[ f(x) = \sqrt{x^2 + 1} - 1 \]

Problem at \( x \approx 0 \).

One type of fix:

\[
\begin{align*}
f(x) &= \left( \sqrt{x^2 + 1} - 1 \right) \left( \frac{\sqrt{x^2 + 1} + 1}{\sqrt{x^2 + 1} + 1} \right) \\
&= \frac{x^2}{\sqrt{x^2 + 1} + 1}
\end{align*}
\]

no subtraction!