CS221
Mock
Floating Point Number Representation

So far we have discussed signed and unsigned number representations. But how do we represent fractions and floating point numbers? For example, we also need a way to represent a number like 409.331. This document will discuss a binary representation for floating point numbers, as well as the IEEE 754 floating point standard.

Converting from Decimal to Binary

Let's say that we want to convert 252.390625 into binary. The first task is to convert the number 252 into binary. We already know how to do this, we just divide 252 by 2 and keep the remainders, repeating the process with the non-fractional part. 252 = 11111100

The next step is to convert 0.390625 into binary. To do this, **instead of dividing by 2, we multiply by 2**. Each time we record whatever is to the left of the decimal place after the operation. The first number becomes the leftmost bit, and the final number will be the rightmost bit. We then repeat this process using whatever is to the right of the decimal place.

0.390625 * 2 =	0.78125	0 as leftmost bit
0.78125 * 2 =	1.5625	1 as the next bit
0.5625 * 2 =	1.125	1
0.125 * 2 =	0.25	0
0.25 * 2 =	0.5	0
0.5 * 2 =	1.0	1
0		

Upon hitting 0, we're finished. The binary representation of this number is then:

11111100.011001

Exercise:

What is 3.625 in binary?
What is 0.1 in binary?

Note that with the last example, we could continue forever. In practice, we continue the process until we reach the precision desired to represent the number. Also note that if we wanted to use this process on something other than base 2, we would just multiply by whatever base we were interested in (e.g., base 16).

Converting Binary to Decimal

Given a floating point number in binary like 1100.011001, how do we convert this back to decimal? The process is almost identical to the process for unsigned binary. The stuff to the left of the decimal point is the same:

$$
\begin{aligned}
1100 = &\\
& 0 * 2^0 + 0 * 2^1 + 1*2^2 + 1*2^3 \\
= \quad & 4+8 \\
= \quad & 12
\end{aligned}
$$

For the fractional part, .011001 we multiply and sum each bit, but starting with 2^{-1} power and continuing up to 2^{-2}, 2^{-3}, etc.

.	0	1	1	0	0	1
	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}

Recall that 2^{-1} is just 1/2, 2^{-2} is 1/4, 2^{-3} is 1/8, etc.

Summing this up gives us:

$$
\begin{aligned}
& 0 * 2^{-1} + 1*2^{-2} + 1*2^{-3} + 0*2^{-4} + 0*2^{-5} + 1*2^{-6} \\
= \quad & 1/4 \quad + \quad 1/8 \quad + \quad 1/64 \\
= \quad & .25 + .125 + .015625 \\
= \quad & .390625
\end{aligned}
$$

Putting these together gives us 12.390625.

Exercise: What is 100.1111 in decimal?

Scientific Notation

What we've described is conceptually how to convert fractions from decimal to binary, but typically the data isn't stored in the computer in the same format. First we need to normalize the data, using scientific notation.

Consider the decimal number 0201.0900 . First we need to determine which digits are significant. The formal rules for significant digits is:

1. A nonzero digit is always significant
2. The digit 0 is never significant when it precedes nonzero digits

Using these rules, we can discard the initial 0 leaving us with 201.0900. Notice that we kept the trailing zeros; they may or may not be significant. Without more information we can't tell if we want precisely 201.0900, or if perhaps 201.09 is all we knew and the extra zeros are padding.

To represent this in scientific notation, we move the decimal point to the position immediately to the right of the leftmost significant digit, and multiply by the correct factor of 10 to get back the original value:

$2.010900 * 10^2$

In this case, we moved the decimal point two places to the left, so we multiply by 10^2. If we had a fraction we would do the opposite and multiply by a fraction of 10:

$0.0020109 = 2.0109 * 10^{-3}$

We can apply the same process to binary fractions, but use powers of 2 instead of powers of 10. Say that we have the binary value 100.1011. Converting this to scientific notation results in:

$1.001011 * 2^2$

Similarly, 0.00100 converted to scientific notation results in:

$1.00 * 2^{-3}$

Exercise: What is 101011.101 in scientific notation?

IEEE 754 Representation

To represent a number using the IEEE 754 format, first convert it to binary scientific notation. For example, let's say that we end up with $1.001011 * 2^3$. In general terms, this value is: (Sign) * (Mantissa) * $2^{(\text{exponent})}$

In storing this value, by default, we will assume that the power is 2. To get this value back, we will need to store the "1.001011", the sign using a sign bit, and then the exponent 3.

The pieces that we must store are the:

- Sign
- Exponent (the 3)
- Mantissa (the 1.001011 part)

In a single precision floating point number using the IEEE 754 format, 32 bits are used to store all of these values. The format is:

```
S EEEEEEEE MMMMMMMMMMMMMMMMMMMMMMM
1     8              23
```

1 bit is allocated to the sign field (the leftmost bit).
8 bits are allocated to store the exponent field.
23 bits are allocated to store the mantissa field.

Sign Field

This is just a sign bit, like we used with signed binary numbers. It is either 0 or 1. 0 indicates a positive number, and 1 indicates a negative number. For our example number of $1.001011 * 2^3$, this is positive so the sign bit would hold a 0.

Exponent Field

The exponent section is eight bits long and is also called the *characteristic*. Since we are using 8 bits, this means we have values from 0 to 255. However, how would we handle negative exponents? To address negative values, the exponent is **biased** by a value of 127. In other words, the value 127 is added to the actual exponent we want to represent:

BiasedExponent = 127 + ActualExponent

A short listing of values for the exponent is:

Decimal	Biased Decimal	Biased Binary
0	127 + 0 = 127	01111111
1	127 + 1 = 128	10000000
2	127 + 2 = 129	10000001
128	127 + 128 = 255	11111111*
-1	127 – 1 = 126	01111110
-127	127 – 127 = 0	00000000*

For our example number of $1.001011 * 2^3$, we want to represent 3. Using the bias of 127, we store 127+3 = 130 or 10000010 for the exponent field.

Mantissa Field

The mantissa section is twenty-three bits long and is sometimes called the *significand*. Since the power of 2 is implicit, all that is left to store are the significant digits of the

* We actually can't represent these values in a single precision IEEE 754 floating point value. These are special cases, described later

number to represent. In the case of our example of $1.001011 * 2^3$ this corresponds to the 1.001011 part.

Initially, if we have 23 bits to use, you might think that the mantissa would hold 10010110000000000000000 to hold our significant digits. However, it does not. The reason is that in scientific notation there will never be a leading 0. For example, we would never have $0.345 * 10^2$. To be in scientific notation, this would have to be expressed as $3.45 * 10^3$. We have the same issue in binary scientific notation. However, since we can't have a leading 0, this means there is only one alternative: there **must** be a leading 1! Since 0 and 1 are the only digits, the leading digit has to be a 1. So, we really don't need to represent the leading 1 in the binary scientific notation. We can just assume it is there. This is referred to as the *hidden bit*. This scheme has the advantage that it gives us one additional bit worth of precision.

The mantissa for $1.001011 * 2^3$ would then be

00101100000000000000000

This long stream of 0's has one additional zero than the initial "guess" at the mantissa's value.

Putting all the pieces together for this example gives us:

Sign bit = 0
Exponent = 10000010
Mantissa = 00101100000000000000000

Or: 0100 0001 0001 0110 0000 0000 0000 0000
Or: 41160000 in hex

Exercises:

Represent –10.4375 in IEEE 754 representation and express as a hex number.

Given the hex number: 3EA80000 what is this in decimal?

How would you represent 0 in IEEE 754?

Double Precision

In addition to the single precision floating point described here, there is also double precision floating point. These have 64 bits instead of 32, and instead of field lengths of 1, 8, and 23 as in single precision, have field lengths of 1, 11, and 52. The exponent field contains a value that is actually 1023 larger than the "true" exponent, rather than being larger by 127 as in single precision. Otherwise, it is exactly the same. The advantage of double precision floating point is the ability to represent larger numbers and also numbers with more precision (more decimal points). However, generally it is slower for the computer to operate upon the 64 bits than it is upon the 32 bits. Chapter 8.5 of Stallings has a table illustrating the range of values that can be represented using double precision.

Special Numbers

There are a couple of numbers that are special. Generally, these are numbers reserved for cases when some kind of error occurs. The most common such case is when you might attempt to divide by zero. Or you might want to represent the number 0 exactly! These special cases are assigned codes, where the exponent field is all 1's or 0's.

The first special case refers to *denormalized* numbers. Denormalized numbers occur when the exponent field is all 0's. Earlier we said that this would generally be 0 – 127 or –127 as the exponent. However, that is not the case. Instead, it is a special case to represent 0.mantissa * 2^{-126} instead of 1.mantissa * 2^{-127}. Why? One reason is this lets us encode the value 0 because we no longer have an implied 1 bit. If the mantissa is all 0's and the exponent is all 0's, then we have 0 * 2^{-126} or 0. This format also lets us represent really small numbers between 0 and 1*2^{-126}. This is referred to as *gradual underflow*. It is possible to have values that are too small to represent. By having denormalized numbers, we can at least extend the lower range of representable numbers. Eventually though, we will have an *underflow* condition where a number is too small to represent, and we end up with 0.

The next special case occurs when the exponent contains all 1's. Again, we said earlier that this would normally be 255 – 127 or an exponent of 128. However, there is a special case encoded for *infinity*. Infinity is encoded when the exponent field is all 1's and the mantissa is all 0's. Note that there can be + and – infinity. The special value of infinity is what happens in an *overflow* condition – when we try to represent a number that is too large for the allocated space.

Another special case also occurs when the exponent field is all 1's. If the exponent contains all 1's but the mantissa does not contain all 0's, then this is referred to as *NaN* or *Not a Number*. The sign bit is unused.

You will encounter NaN if you attempt to divide 0 by 0 or perform some other ambiguous computation. If you divide a number by 0, you will get infinity. Note that some systems will return NaN if you attempt to divide by 0, instead of infinity. Also note that not all CPUs will check for these conditions. Adding hardware to check for these special cases can be expensive and actually slow the system down as more transistors are utilized. For this reason, some CPUs will simply generate an *exception* (an error) and refuse to complete the operation.

Computing with Floating Point

Due to truncation, precision, and rounding errors in the exponent and mantissa you are not guaranteed that floating point numbers will be identical! As we noticed with the number 0.2, we need an infinite number of bits if we want to represent it exactly.

For example, if you compute add 0.2 to a number 100,000 times, you won't get a value that is exactly 20,000 larger. Due to rounding errors, you will get a slightly different value. For these reasons, it is best to check for a range of numbers when dealing with floating point (e.g., >=19999.99 and <= 20000.01) instead of strict equality.

Floating Point Inaccuracies

Here is an example illustrating how a floating point representation can lose accuracy compared to regular unsigned integers, especially as we increase the magnitude of the value.

Informally, if we have the same number of bits to represent both a floating point value and an integer, in the floating point format we have to use some bits to represent the exponent and sign. This leaves fewer bits for the mantissa. An unsigned integer is able to use all bits to represent mantissa. This means the floating point format will have less precision than the integer format.

Consider the following floating point format that uses 8 bits:

```
EEE MMMMM
  3      5
```

The exponent is unbiased, so the exponent field can represent exponents from 0 to 7. The mantissa uses the hidden one bit. There are no special cases as with IEEE 754.

If the mantissa is all zero's and the exponent is 0, then the smallest number we can represent is $1.00000 * 2^0$ which equals 1.

If the mantissa is all one's and the exponent is 111, then the largest number we can represent is $1.11111 * 2^7$ which equals 11111100 or 252.

With all eight bits as an unsigned integer, we can represent 2^8 patterns, or the integers 0 to 255.

Right off the bat, we can see that our floating point format can't represent the numbers 253, 254, or 255, while the integer format can. Of course, the floating point format can represent lots of fractional numbers that the integer format can't represent.

Here is an example of error with the floating point format. If the biggest number we can represent is 252, what is the next smallest number representable? In binary that would be the value:

```
111 11110
```

This is $1.11110 * 2^7$ which equals 11111000 or 248. We're missing four whole integers in between this and the largest number representable.

The next smallest value in binary is:

111 11101

This is 1.11101 * 2^7 which equals 11110100 or 244. We dropped another four values.

The amount of error is worst for the largest values, since any lack of precision in the mantissa is amplified by the exponent. For small values, we lose little to no precision. For example, we said that the smallest value representable was 1. What is the next smallest value? It is:

000 00001

This is 1.00001 * 2^0 which is just 1.03125. No loss of precision here compared to integers!

We can also represent exactly the number two:

001 00000

This is 1.00000 * 2^1 which is two.

In summary, the floating point representation suffers in precision for large numbers, but is more accurate for small values. This is something to take into account whenever you are converting integer numbers to floating point format, especially if the integer is large. The resulting floating point number may not be particularly close to the actual integer and could cause an error in your program.

Floating Point Arithmetic

To briefly describe floating point arithmetic, all numbers are stored and processed in exponential form.

Addition and Subtraction: Unlike twos complement, addition and subtraction are more complicated than multiplication and division when operating with floating point values. As with two's complement, subtraction is performed by changing the sign of the subtrahend and then performing an addition.

1. The first step is to check for zeros. If either operand is zero, the other is reported as the result.

2. The second step is to align the mantissa / significand. For example, given:

$$123 * 10^0 + 456 * 10^{-2}$$

we can't just add the 123 and the 456. The digits have to be set to an equivalent exponent.

$$123 * 10^0 + 4.56 * 10^0 = 127.56 * 10^0$$

We could have aligned the digits by either shifting one number right or another number left. Since either operation can result in the loss of digits and precision, it is the smaller number that is shifted. Therefore, any lost digits are of smaller significance. For each shift right the exponent is increased by one, and for each shift left the exponent is decreased by one.

3. Add or subtract the mantissas

4. Normalize the result. After performing the addition we may end up with a non-normalized number (e.g., $127.56 * 10^0$) which will need to be re-normalized and put back into the IEEE 754 format. We might also have the unfortunate case that the new value results in overflow or underflow of the exponent, as well.

Conceptually, multiplication is easier: add the exponents and multiply the mantissas.

For division, the exponents are subtracted and the mantissas are divided.

Both might require reorganizing the results into normal form and watching out for the signs and for overflow/underflow, and division by zero. Consult the Stallings textbook for detailed flowcharts on performing addition, multiplication, or division using floating point.

Other resources

For a comprehensive discussion about floating point representations, see Goldberg's article "What Every Computer Scientist Should Know About Floating Point Arithmetic" at http://cch.loria.fr/documentation/IEEE754/ACM/goldberg.pdf (warning, 3Mb large file).

Sun Microsystems also has a good discussion of underflow available at http://docs.sun.com/htmlcoll/coll.648.2/iso-8859-1/NUMCOMPGD/ncg_math.html

What Every Computer Scientist Should Know About Floating-Point Arithmetic

DAVID GOLDBERG

Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, California 94304

Floating-point arithmetic is considered an esotoric subject by many people. This is rather surprising, because floating-point is ubiquitous in computer systems: Almost every language has a floating-point datatype; computers from PCs to supercomputers have floating-point accelerators; most compilers will be called upon to compile floating-point algorithms from time to time; and virtually every operating system must respond to floating-point exceptions such as overflow This paper presents a tutorial on the aspects of floating-point that have a direct impact on designers of computer systems. It begins with background on floating-point representation and rounding error, continues with a discussion of the IEEE floating-point standard, and concludes with examples of how computer system builders can better support floating point.

Categories and Subject Descriptors: (Primary) C.0 [**Computer Systems Organization**]: General—*instruction set design*; D.3.4 [**Programming Languages**]: Processors—*compilers, optimization*; G.1.0 [**Numerical Analysis**]: General—*computer arithmetic, error analysis, numerical algorithms* (Secondary) D.2.1 [Software Engineering]: Requirements/Specifications—*languages*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*semantics* D.4.1 [**Operating Systems**]: Process Management—*synchronization*

General Terms: Algorithms, Design, Languages

Additional Key Words and Phrases: denormalized number, exception, floating-point, floating-point standard, gradual underflow, guard digit, NaN, overflow, relative error, rounding error, rounding mode, ulp, underflow

INTRODUCTION

Builders of computer systems often need information about floating-point arithmetic. There are however, remarkably few sources of detailed information about it. One of the few books on the subject, *Floating-Point Computation* by Pat Sterbenz, is long out of print. This paper is a tutorial on those aspects of floating-point arithmetic (*floating-point* hereafter) that have a direct connection to systems building. It consists of three loosely connected parts. The first (Section 1) discusses the implications of using different rounding strategies for the basic operations of addition, subtraction, multiplication, and division. It also contains background information on the two methods of measuring rounding error, *ulps* and *relative error*. The second part discusses the IEEE floating-point standard, which is becoming rapidly accepted by commercial hardware manufacturers. Included in the IEEE standard is the rounding method for basic operations; therefore, the discussion of the standard draws on the material in Section 1. The third part discusses the connections between floating point and the design of various aspects of computer systems. Topics include instruction set design, CONTENTS

optimizing compilers, and exception handling.

All the statements made about floating-point are provided with justifications, but those explanations not central to the main argument are in a section called *The Details* and can be skipped if desired. In particular, the proofs of many of the theorems appear in this section. The end of each proof is marked with the ■ symbol; when a proof is not included, the ■ appears immediately following the statement of the theorem.

1. ROUNDING ERROR

Squeezing infinitely many real numbers into a finite number of bits requires an approximate representation. Although there are infinitely many integers, in most programs the result of integer computations can be stored in 32 bits. In contrast, given any fixed number of bits, most calculations with real numbers will produce quantities that cannot be exactly represented using that many bits. Therefore, the result of a floating-point calculation must often be rounded in order to fit back into its finite representation. The resulting rounding error is the characteristic feature of floating-point computation. Section 1.2 describes how it is measured.

Since most floating-point calculations have rounding error anyway, does it matter if the basic arithmetic operations introduce a bit more rounding error than necessary? That question is a main theme throughout Section 1. Section 1.3 discusses *guard digits*, a means of reducing the error when subtracting two nearby numbers. Guard digits were considered sufficiently important by IBM that in 1968 it added a guard digit to the double precision format in the System/360 architecture (single precision already had a guard digit) and retrofitted all existing machines in the field. Two examples are given to illustrate the utility of guard digits.

The IEEE standard goes further than just requiring the use of a guard digit. It gives an algorithm for addition, subtraction, multiplication, division, and square root and requires that implementations produce the same result as that algorithm. Thus, when a program is moved from one machine to another, the results of the basic operations will be the same in every bit if both machines support the IEEE standard. This greatly simplifies the porting of programs. Other uses of this precise specification are given in Section 1.5.

2.1 Floating-Point Formats

Several different representations of real numbers have been proposed, but by far the most widely used is the floating-point representation.[1] Floating-point representations have a base β (which is always assumed to be even) and a precision p. If $\beta = 10$ and $p = 3$, the number 0.1 is represented as 1.00×10^{-1}. If $\beta = 2$ and $p = 24$, the decimal number 0.1 cannot

[1]Examples of other representatıons are *floatıng slash* and *signed logarıthm* [Matula and Kornerup 1985; Swartzlander and Alexopoulos 1975].

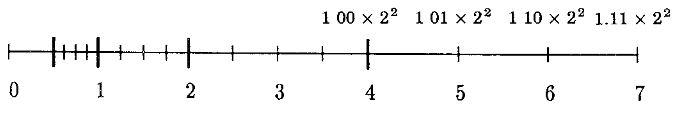


Figure 1. Normalized numbers when $\beta = 2$, $p = 3$, $e_{\min} = -1$, $e_{\max} = 2$.

be represented exactly but is approximately $1.10011001100110011001101 \times 2^{-4}$. In general, a floating-point number will be represented as $\pm d.dd \cdots d \times \beta^e$, where $d.dd \cdots d$ is called the *significand*[2] and has p digits. More precisely, $\pm d_0.d_1 d_2 \cdots d_{p-1} \times \beta^e$ represents the number

$$\pm\left(d_0 + d_1\beta^{-1} + \cdots + d_{p-1}\beta^{-(p-1)}\right)\beta^e, \qquad 0 < \underline{d}_i < \beta. \quad (1)$$

The term *floating-point number* will be used to mean a real number that can be exactly represented in the format under discussion. Two other parameters associated with floating-point representations are the largest and smallest allowable exponents, $e_{\max}$ and $e_{\min}$. Since there are β^p possible significands and $e_{\max} - e_{\min} + 1$ possible exponents, a floating-point number can be encoded in $\lceil \log_2(e_{\max} - e_{\min} + 1) \rceil + \lceil \log_2(\beta^p) \rceil + 1$ bits, where the final $+1$ is for the sign bit. The precise encoding is not important for now.

There are two reasons why a real number might not be exactly representable as a floating-point number. The most common situation is illustrated by the decimal number 0.1. Although it has a finite decimal representation, in binary it has an infinite repeating representation. Thus, when $\beta = 2$, the number 0.1 lies strictly between two floating-point numbers and is exactly representable by neither of them. A less common situation is that a real number is out of range; that is, its absolute value is larger than $\beta \times \beta^{e_{\max}}$ or smaller than $1.0 \times \beta^{e_{\min}}$. Most of this paper discusses issues due to the first reason. Numbers that are out of range will, however, be discussed in Sections 2.2.2 and 2.2.4.

Floating-point representations are not necessarily unique. For example, both 0.01×10^1 and 1.00×10^{-1} represent 0.1. If the leading digit is nonzero [$d_0 \neq 0$ in eq. (1)], the representation is said to be *normalized*. The floating-point number 1.00×10^{-1} is normalized, whereas 0.01×10^1 is not. When $\beta = 2$, $p = 3$, $e_{\min} = -1$, and $e_{\max} = 2$, there are 16 normalized floating-point numbers, as shown in Figure 1. The bold hash marks correspond to numbers whose significand is 1.00. Requiring that a floating-point representation be normalized makes the representation unique. Unfortunately, this restriction makes it impossible to represent zero! A natural way to represent 0 is with $1.0 \times \beta^{e_{\min}-1}$, since this preserves the fact that the numerical ordering of nonnegative real numbers corresponds to the lexicographical ordering of their floating-point representations.[3] When the exponent is stored in a k bit field, that means that only $2^k - 1$ values are available for use as exponents, since one must be reserved to represent 0.

Note that the $\times$ in a floating-point number is part of the notation and different from a floating-point multiply operation. The meaning of the $\times$ symbol should be clear from the context. For example, the expression $(2.5 \times 10^{-3}) \times (4.0 \times 10^2)$ involves only a single floating-point multiplication.

[2]This term was introduced by Forsythe and Moler [1967] and has generally replaced the older term *mantissa*.

[3]This assumes the usual arrangement where the exponent is stored to the left of the significand

1.2 Relative Error and Ulps

Since rounding error is inherent in floating-point computation, it is important to have a way to measure this error. Consider the floating-point format with $\beta = 10$ and $p = 3$, which will be used throughout this section. If the result of a floating-point computation is 3.12×10^{-2} and the answer when computed to infinite precision is .0314, it is clear that this is in error by 2 units in the last place. Similarly, if the real number .0314159 is represented as 3.14×10^{-2}, then it is in error by .159 units in the last place. In general, if the floating-point number $d.d \cdots d \times \beta^e$ is used to represent z, it is in error by $|d.d \cdots d - (z/\beta^e)| \beta^{p-1}$ units in the last place.[4] The term *ulps* will be used as shorthand for "units in the last place." If the result of a calculation is the floating-point number nearest to the correct result, it still might be in error by as much as 1/2 ulp.

Another way to measure the difference between a floating-point number and the real number it is approximating is *relative error*, which is the difference between the two numbers divided by the real number. For example, the relative error committed when approximating 3.14159 by 3.14×10^0 is .00159/3.14159 $\approx$ .0005.

To compute the relative error that corresponds to 1/2 ulp, observe that when a real number is approximated by the closest possible floating-point number $\overbrace{d\,dd \cdots dd}^{p} \times \beta^e$, the absolute error can be as large as $\overbrace{0\,00 \cdots 00}^{p}\beta' \times \beta^e$ where β' is the digit $\beta/2$. This error is $((\beta/2)\beta^{-p}) \times \beta^e$. Since numbers of the form $d.dd \cdots dd \times \beta^e$ all have this same absolute error but have values that range between β^e and $\beta \times \beta^e$, the relative error ranges between $((\beta/2)\beta^{-p}) \times \beta^e/\beta^e$ and $((\beta/2)\beta^{-p}) \times \beta^e/\beta^{e+1}$. That is,

$$\frac{1}{2}\beta^{-p} \le \frac{1}{2}\text{ulp} \le \frac{\beta}{2}\beta^{-p}. \qquad (2)$$

In particular, the relative error corresponding to 1/2 ulp can vary by a factor of β. This factor is called the *wobble*. Setting $\epsilon = (\beta/2)\beta^{-p}$ to the largest of the bounds in (2), we can say that when a real number is rounded to the closest floating-point number, the relative error is always bounded by ϵ, which is referred to as *machine epsilon*.

In the example above, the relative error was .00159/3.14159 $\approx$ 0005. To avoid such small numbers, the relative error is normally written as a factor times ϵ, which in this case is $\epsilon = (\beta/2)\beta^{-p} = 5(10)^{-3} = .005$. Thus, the relative error would be expressed as $((.00159/3.14159)/.005)\epsilon \approx 0.1\epsilon$.

To illustrate the difference between ulps and relative error, consider the real number $x = 12.35$. It is approximated by $\tilde{x} = 1.24 \times 10^1$. The error is 0.5 ulps; the relative error is 0.8ϵ. Next consider the computation $8x$. The exact value is $8x = 98.8$, whereas, the computed value is $8\tilde{x} = 9.92 \times 10^1$. The error is now 4.0 ulps, but the relative error is still 0.8ϵ. The error measured in ulps is eight times larger, even though the relative error is the same. In general, when the base is β, a fixed relative error expressed in ulps can wobble by a factor of up to β. Conversely, as eq. (2) shows, a fixed error of 1/2 ulps results in a relative error that can wobble by β.

The most natural way to measure rounding error is in ulps. For example, rounding to the nearest floating-point number corresponds to 1/2 ulp. When analyzing the rounding error caused by various formulas, however, relative error is a better measure. A good illustration of this is the analysis immediately following the proof of Theorem 10. Since ϵ can overestimate the effect of rounding to the nearest floating-point number by the wobble factor of β, error estimates of formulas will be tighter on machines with a small β.

[4]Unless the number z is larger than $\beta^{e_{\max}+1}$ or smaller than $\beta^{e_{\min}}$. Numbers that are out of range in this fashion will not be considered until further notice.

When only the order of magnitude of rounding error is of interest, ulps and ϵ may be used interchangeably since they differ by at most a factor of β. For example, when a floating-point number is in error by n ulps, that means the number of contaminated digits is $\log_\beta n$. If the relative error in a computation is $n\epsilon$, then

$$\text{contaminated digits} \approx \log_\beta n. \qquad (3)$$

1.3 Guard Digits

One method of computing the difference between two floating-point numbers is to compute the difference exactly, then round it to the nearest floating-point number. This is very expensive if the operands differ greatly in size. Assuming $p = 3$, $2.15 \times 10^{12} - 1.25 \times 10^{-5}$ would be calculated as

$$\begin{aligned} x &= 2.15 \times 10^{12} \\ y &= .0000000000000000125 \times 10^{12} \\ x - y &= 2.1499999999999999875 \times 10^{12}, \end{aligned}$$

which rounds to 2.15×10^{12}. Rather than using all these digits, floating-point hardware normally operates on a fixed number of digits. Suppose the number of digits kept is p and that when the smaller operand is shifted right, digits are simply discarded (as opposed to rounding). Then, $2.15 \times 10^{12} - 1.25 \times 10^{-5}$ becomes

$$\begin{aligned} x &= 2.15 \times 10^{12} \\ y &= 0.00 \times 10^{12} \\ x - y &= 2.15 \times 10^{12}. \end{aligned}$$

The answer is exactly the same as if the difference had been computed exactly then rounded. Take another example: $10.1 - 9.93$. This becomes

$$\begin{aligned} x &= 1.01 \times 10^{1} \\ y &= 0.99 \times 10^{1} \\ x - y &= .02 \times 10^{1}. \end{aligned}$$

The correct answer is .17, so the computed difference is off by 30 ulps and is wrong in every digit! How bad can the error be?

Theorem 1

Using a floating-point format with parameters β and p and computing differences using p digits, the relative error of the result can be as large as $\beta - 1$.

Proof. A relative error of $\beta - 1$ in the expression $x - y$ occurs when $x = 1.00 \cdots 0$ and $y = .\varrho\varrho \cdots \varrho$, where $\varrho = \beta - 1$. Here y has p digits (all equal to ϱ). The exact difference is $x - y = \beta^{-p}$. When computing the answer using only p digits, however, the rightmost digit of y gets shifted off, so the computed difference is β^{-p+1}. Thus, the error is $\beta^{-p} - \beta^{-p+1} = \beta^{-p}(\beta - 1)$, and the relative error is $\beta^{-p}(\beta - 1)/\beta^{-p} = \beta - 1$. ■

When $\beta = 2$, the absolute error can be as large as the result, and when $\beta = 10$, it can be nine times larger. To put it another way, when $\beta = 2$, (3) shows that the number of contaminated digits is $\log_2(1/\epsilon) = \log_2(2^p) = p$. That is, all of the p digits in the result are wrong!

Suppose one extra digit is added to guard against this situation (a *guard digit*). That is, the smaller number is truncated to $p + 1$ digits, then the result of the subtraction is rounded to p digits. With a guard digit, the previous example becomes

$$\begin{aligned} x &= 1.010 \times 10^{1} \\ y &= 0.993 \times 10^{1} \\ x - y &= .017 \times 10^{1}, \end{aligned}$$

and the answer is exact. With a single guard digit, the relative error of the result may be greater than ϵ, as in $110 - 8.59$:

$$\begin{aligned} x &= 1.10 \times 10^{2} \\ y &= .085 \times 10^{2} \\ x - y &= 1.015 \times 10^{2} \end{aligned}$$

This rounds to 102, compared with the correct answer of 101.41, for a relative error of .006, which is greater than

$\epsilon = .005$. In general, the relative error of the result can be only slightly larger than ϵ. More precisely, we have Theorem 2.

Theorem 2

If x and y are floating-point numbers in a format with β and p and if subtraction is done with $p + 1$ digits (i.e., one guard digit), then the relative rounding error in the result is less than 2ϵ.

This theorem will be proven in Section 4.1. Addition is included in the above theorem since x and y can be positive or negative.

1.4 Cancellation

Section 1.3 can be summarized by saying that without a guard digit, the relative error committed when subtracting two nearby quantities can be very large. In other words, the evaluation of any expression containing a subtraction (or an addition of quantities with opposite signs) could result in a relative error so large that *all* the digits are meaningless (Theorem 1). When subtracting nearby quantities, the most significant digits in the operands match and cancel each other. There are two kinds of cancellation: catastrophic and benign.

Catastrophic cancellation occurs when the operands are subject to rounding errors. For example, in the quadratic formula, the expression $b^2 - 4ac$ occurs. The quantities b^2 and $4ac$ are subject to rounding errors since they are the results of floating-point multiplications. Suppose they are rounded to the nearest floating-point number and so are accurate to within 1/2 ulp. When they are subtracted, cancellation can cause many of the accurate digits to disappear, leaving behind mainly digits contaminated by rounding error. Hence the difference might have an error of many ulps. For example, consider $b = 3.34$, $a = 1.22$, and $c = 2.28$. The exact value of $b^2 - 4ac$ is .0292. But b^2 rounds to 11.2 and $4ac$ rounds to 11.1, hence the final answer is .1, which is an error by 70 ulps even though $11.2 - 11.1$ is exactly equal to .1. The subtraction did not introduce any error but rather exposed the error introduced in the earlier multiplications.

Benign cancellation occurs when subtracting exactly known quantities. If x and y have no rounding error, then by Theorem 2 if the subtraction is done with a guard digit, the difference $x - y$ has a very small relative error (less than 2ϵ).

A formula that exhibits catastrophic cancellation can sometimes be rearranged to eliminate the problem. Again consider the quadratic formula

$$r_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a},$$

$$r_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}. \qquad (4)$$

When $b^2 \gg ac$, then $b^2 - 4ac$ does not involve a cancellation and $\sqrt{b^2 - 4ac} \approx |b|$. But the other addition (subtraction) in one of the formulas will have a catastrophic cancellation. To avoid this, multiply the numerator and denominator of r_1 by $-b - \sqrt{b^2 - 4ac}$ (and similarly for r_2) to obtain

$$r_1 = \frac{2c}{-b - \sqrt{b^2 - 4ac}},$$

$$r_2 = \frac{2c}{-b + \sqrt{b^2 - 4ac}}. \qquad (5)$$

If $b^2 \gg ac$ and $b > 0$, then computing r_1 using formula (4) will involve a cancellation. Therefore, use (5) for computing r_1 and (4) for r_2. On the other hand, if $b < 0$, use (4) for computing r_1 and (5) for r_2.

The expression $x^2 - y^2$ is another formula that exhibits catastrophic cancellation. It is more accurate to evaluate it as $(x - y)(x + y)$.[5] Unlike the quadratic

[5]Although the expression $(x - y)(x + y)$ does not cause a catastrophic cancellation, it is slightly less accurate than $x^2 - y^2$ if $x \gg y$ or $x \ll y$. In this case, $(x - y)(x + y)$ has three rounding errors, but $x^2 - y^2$ has only two since the rounding error committed when computing the smaller of x^2 and y^2 does not affect the final subtraction.

formula, this improved form still has a subtraction, but it is a benign cancellation of quantities without rounding error, not a catastrophic one. By Theorem 2, the relative error in $x - y$ is at most 2ϵ. The same is true of $x + y$. Multiplying two quantities with a small relative error results in a product with a small relative error (see Section 4.1).

To avoid confusion between exact and computed values, the following notation is used. Whereas $x - y$ denotes the exact difference of x and y, $x \oslash y$ denotes the computed difference (i.e., with rounding error). Similarly $\oplus$, $\otimes$, and $\oslash$ denote computed addition, multiplication, and division, respectively. All caps indicate the computed value of a function, as in LN(x) or SQRT(x). Lowercase functions and traditional mathematical notation denote their exact values as in $\ln(x)$ and $\sqrt{x}$.

Although $(x \oslash y) \otimes (x \oplus y)$ is an excellent approximation of $x^2 - y^2$, the floating-point numbers x and y might themselves be approximations to some true quantities $\hat{x}$ and $\hat{y}$. For example, $\hat{x}$ and $\hat{y}$ might be exactly known decimal numbers that cannot be expressed exactly in binary. In this case, even though $x \ominus y$ is a good approximation to $x - y$, it can have a huge relative error compared to the true expression $\hat{x} - \hat{y}$, and so the advantage of $(x + y)(x - y)$ over $x^2 - y^2$ is not as dramatic. Since computing $(x + y)(x - y)$ is about the same amount of work as computing $x^2 - y^2$, it is clearly the preferred form in this case. In general, however, replacing a catastrophic cancellation by a benign one is not worthwhile if the expense is large because the input is often (but not always) an approximation. But eliminating a cancellation entirely (as in the quadratic formula) is worthwhile even if the data are not exact. Throughout this paper, it will be assumed that the floating-point inputs to an algorithm are exact and that the results are computed as accurately as possible.

The expression $x^2 - y^2$ is more accurate when rewritten as $(x - y)(x + y)$ because a catastrophic cancellation is replaced with a benign one. We next present more interesting examples of formulas exhibiting catastrophic cancellation that can be rewritten to exhibit only benign cancellation.

The area of a triangle can be expressed directly in terms of the lengths of its sides a, b, and c as

$$A = \sqrt{s(s-a)(s-b)(s-c)}\,, \quad \text{where } s = \frac{a+b+c}{2}. \tag{6}$$

Suppose the triangle is very flat; that is, $a \approx b + c$. Then $s \approx a$, and the term $(s - a)$ in eq. (6) subtracts two nearby numbers, one of which may have rounding error. For example, if $a = 9.0$, $b = c = 4.53$, then the correct value of s is 9.03 and A is 2.34. Even though the computed value of s (9.05) is in error by only 2 ulps, the computed value of A is 3.04, an error of 60 ulps.

There is a way to rewrite formula (6) so that it will return accurate results even for flat triangles [Kahan 1986]. It is

$$A = \big[(1a + (b + c))(c - (a - b)) \times (c + (a - b))(a + (b - c))\big]^{1/2}/4, \quad a \ge b \ge c. \tag{7}$$

If a, b, and c do not satisfy $a \ge b \ge c$, simply rename them before applying (7). It is straightforward to check that the right-hand sides of (6) and (7) are algebraically identical. Using the values of a, b, and c above gives a computed area of 2.35, which is 1 ulp in error and much more accurate than the first formula.

Although formula (7) is much more accurate than (6) for this example, it would be nice to know how well (7) performs in general.

Theorem 3

The rounding error incurred when using (7) to compute the area of a triangle is at most 11ϵ, provided subtraction is performed with a guard digit, $\epsilon \le .005$, and square roots are computed to within 1/2 ulp.

The condition that $\epsilon \leq .005$ is met in virtually every actual floating-point system. For example, when $\beta = 2$, $p \geq 8$ ensures that $\epsilon < .005$, and when $\beta = 10$, $p \geq 3$ is enough.

In statements like Theorem 3 that discuss the relative error of an expression, it is understood that the expression is computed using floating-point arithmetic. In particular, the relative error is actually of the expression

$$\begin{aligned}&\big(\mathrm{SQRT}(a \oplus (b \oplus c)) \otimes (c \oslash (a \oslash b))\\ &\quad \otimes (c \oplus (a \oslash b)) \otimes (a \oplus (b \oslash c))\big)\\ &\quad \oslash 4. \qquad (8)\end{aligned}$$

Because of the cumbersome nature of (8), in the statement of theorems we will usually say *the computed value of E* rather than writing out E with circle notation.

Error bounds are usually too pessimistic. In the numerical example given above, the computed value of (7) is 2.35, compared with a true value of 2.34216 for a relative error of 0.7ϵ, which is much less than 11ϵ. The main reason for computing error bounds is not to get precise bounds but rather to verify that the formula does not contain numerical problems.

A final example of an expression that can be rewritten to use benign cancellation is $(1 + x)^n$, where $x \ll 1$. This expression arises in financial calculations. Consider depositing \$100 every day into a bank account that earns an annual interest rate of 6%, compounded daily. If $n = 365$ and $i = .06$, the amount of money accumulated at the end of one year is $100[(1 + i/n)^n - 1]/(i/n)$ dollars. If this is computed using $\beta = 2$ and $p = 24$, the result is \$37615.45 compared to the exact answer of \$37614.05, a discrepancy of \$1.40. The reason for the problem is easy to see. The expression $1 + i/n$ involves adding 1 to .0001643836, so the low order bits of i/n are lost. This rounding error is amplified when $1 + i/n$ is raised to the nth power.

The troublesome expression $(1 + i/n)^n$ can be rewritten as $\exp[n \ln(1 + i/n)]$, where now the problem is to compute $\ln(1 + x)$ for small x. One approach is to use the approximation $\ln(1 + x) \approx x$, in which case the payment becomes \$37617.26, which is off by \$3.21 and even less accurate than the obvious formula. But there is a way to compute $\ln(1 + x)$ accurately, as Theorem 4 shows [Hewlett-Packard 1982]. This formula yields \$37614.07, accurate to within 2 cents!

Theorem 4 assumes that $\mathrm{LN}(x)$ approximates $\ln(x)$ to within 1/2 ulp. The problem it solves is that when x is small, $\mathrm{LN}(1 \oplus x)$ is not close to $\ln(1 + x)$ because $1 \oplus x$ has lost the information in the low order bits of x. That is, the computed value of $\ln(1 + x)$ is not close to its actual value when $x \ll 1$.

Theorem 4

If $\ln(1 - x)$ *is computed using the formula*

$$\ln(1 + x) = \begin{cases} x & \text{for } 1 \oplus x = 1 \\ \dfrac{x \ln(1 + x)}{(1 + x) - 1} & \text{for } 1 \oplus x \neq 1 \end{cases}$$

the relative error is at most 5ϵ *when* $0 \leq x < 3/4$, *provided subtraction is performed with a guard digit,* $\epsilon < 0.1$, *and* ln *is computed to within* 1/2 *ulp.*

This formula will work for any value of x but is only interesting for $x \ll 1$, which is where catastrophic cancellation occurs in the naive formula $\ln(1 + x)$. Although the formula may seem mysterious, there is a simple explanation for why it works. Write $\ln(1 + x)$ as $x[\ln(1 + x)/x] = x\mu(x)$. The left-hand factor can be computed exactly, but the right-hand factor $\mu(x) = \ln(1 + x)/x$ will suffer a large rounding error when adding 1 to x. However, μ is almost constant, since $\ln(1 + x) \approx x$. So changing x slightly will not introduce much error. In other words, if $\tilde{x} \approx x$, computing $x\mu(\tilde{x})$ will be a good approximation to $x\mu(x) = \ln(1 + x)$. Is there a value for $\tilde{x}$ for which $\tilde{x}$ and $\tilde{x} + 1$ can be computed accurately? There is; namely, $\tilde{x} = (1 \oplus x) \ominus 1$, because then $1 + \tilde{x}$ is exactly equal to $1 \oplus x$.

The results of this section can be summarized by saying that a guard digit guarantees accuracy when nearby precisely known quantities are subtracted (benign cancellation). Sometimes a formula that gives inaccurate results can be rewritten to have much higher numerical accuracy by using benign cancellation; however, the procedure only works if subtraction is performed using a guard digit. The price of a guard digit is not high because is merely requires making the adder 1 bit wider. For a 54 bit double precision adder, the additional cost is less than 2%. For this price, you gain the ability to run many algorithms such as formula (6) for computing the area of a triangle and the expression in Theorem 4 for computing $\ln(1 + x)$. Although most modern computers have a guard digit, there are a few (such as Crays) that do not.

1.5 Exactly Rounded Operations

When floating-point operations are done with a guard digit, they are not as accurate as if they were computed exactly then rounded to the nearest floating-point number. Operations performed in this manner will be called *exactly rounded*. The example immediately preceding Theorem 2 shows that a single guard digit will not always give exactly rounded results. Section 1.4 gave several examples of algorithms that require a guard digit in order to work properly. This section gives examples of algorithms that require exact rounding.

So far, the definition of rounding has not been given. Rounding is straightforward, with the exception of how to round halfway cases; for example, should 12.5 round to 12 or 13? One school of thought divides the 10 digits in half, letting $\{0,1,2,3,4\}$ round down and $\{5,6,7,8,9\}$ round up; thus 12.5 would round to 13. This is how rounding works on Digital Equipment Corporation's VAX[6] computers. Another school of thought says that since numbers ending in 5 are halfway between two possible roundings, they should round down half the time and round up the other half. One way of obtaining this 50% behavior is to require that the rounded result have its least significant digit be even. Thus 12.5 rounds to 12 rather than 13 because 2 is even. Which of these methods is best, round up or round to even? Reiser and Knuth [1975] offer the following reason for preferring round to even.

Theorem 5

Let x and y be floating-point numbers, and define $x_0 = x$, $x_1 = (x_0 \ominus y) \oplus y, \ldots, x_n = (x_{n-1} \ominus y) \oplus y$. If $\oplus$ and $\ominus$ are exactly rounded using round to even, then either $x_n = x$ for all n or $x_n = x_1$ for all $n \geq 1$. ■

To clarify this result, consider $\beta = 10$, $p = 3$ and let $x = 1.00$, $y = -.555$. When rounding up, the sequence becomes $x_0 \ominus y = 1.56$, $x_1 = 1.56 \ominus .555 = 1.01$, $x_1 \ominus y = 1.01 \oplus .555 = 1.57$, and each successive value of x_n increases by .01. Under round to even, x_n is always 1.00. This example suggests that when using the round up rule, computations can gradually drift upward, whereas when using round to even the theorem says this cannot happen. Throughout the rest of this paper, round to even will be used.

One application of exact rounding occurs in multiple precision arithmetic. There are two basic approaches to higher precision. One approach represents floating-point numbers using a very large significand, which is stored in an array of words, and codes the routines for manipulating these numbers in assembly language. The second approach represents higher precision floating-point numbers as an array of ordinary floating-point

[6]VAX is a trademark of Digital Equipment Corporation.

numbers, where adding the elements of the array in infinite precision recovers the high precision floating-point number. It is this second approach that will be discussed here. The advantage of using an array of floating-point numbers is that it can be coded portably in a high-level language, but it requires exactly rounded arithmetic.

The key to multiplication in this system is representing a product xy as a sum, where each summand has the same precision as x and y. This can be done by splitting x and y. Writing $x = x_h + x_l$ and $y = y_h + y_l$, the exact product is $xy = x_h y_h + x_h y_l + x_l y_h + x_l y_l$. If x and y have p bit significands, the summands will also have p bit significands, provided x_l, x_h, y_h, y_l can be represented using $\lfloor p/2 \rfloor$ bits. When p is even, it is easy to find a splitting. The number $x_0.x_1 \cdots x_{p-1}$ can be written as the sum of $x_0.x_1 \cdots x_{p/2-1}$ and $0.0 \cdots 0x_{p/2} \cdots x_{p-1}$. When p is odd, this simple splitting method will not work. An extra bit can, however, be gained by using negative numbers. For example, if $\beta = 2$, $p = 5$, and $x = .10111$, x can be split as $x_h = .11$ and $x_l = -.00001$. There is more than one way to split a number. A splitting method that is easy to compute is due to Dekker [1971], but it requires more than a single guard digit.

Theorem 6

Let p be the floating-point precision, with the restriction that p is even when $\beta > 2$, and assume that floating-point operations are exactly rounded. Then if $k = \lceil p/2 \rceil$ is half the precision (rounded up) and $m = \beta^k + 1$, x can be split as $x = x_h + x_l$, where $x_h = (m \otimes x) \ominus (m \otimes x \ominus x)$, $x_l = x \ominus x_h$, and each x_i is representable using $\lfloor p/2 \rfloor$ bits of precision.

To see how this theorem works in an example, let $\beta = 10$, $p = 4$, $b = 3.476$, $a = 3.463$, and $c = 3.479$. Then $b^2 - ac$ rounded to the nearest floating-point number is .03480, while $b \otimes b = 12.08$, $a \otimes c = 12.05$, and so the computed value of $b^2 - ac$ is .03. This is an error of 480 ulps. Using Theorem 6 to write $b = 3.5 - .024$, $a = 3.5 - .037$, and $c = 3.5 - .021$, b^2 becomes $3.5^2 - 2 \times 3.5 \times .024 + .024^2$. Each summand is exact, so $b^2 = 12.25 - .168 + .000576$, where the sum is left unevaluated at this point. Similarly,

$$\begin{aligned} ac &= 3.5^2 - (3.5 \times .037 + 3.5 \times .021) \\ &\quad + .037 \times .021 \\ &= 12.25 - .2030 + .000777. \end{aligned}$$

Finally, subtracting these two series term by term gives an estimate for $b^2 - ac$ of $0 \oplus .0350 \ominus .04685 = .03480$, which is identical to the exactly rounded result. To show that Theorem 6 really requires exact rounding, consider $p = 3$, $\beta = 2$, and $x = 7$. Then $m = 5$, $mx = 35$, and $m \otimes x = 32$. If subtraction is performed with a single guard digit, then $(m \otimes x) \ominus x = 28$. Therefore, $x_h = 4$ and $x_l = 3$, hence x_l not representable with $\lfloor p/2 \rfloor = 1$ bit.

As a final example of exact rounding, consider dividing m by 10. The result is a floating-point number that will in general not be equal to $m/10$. When $\beta = 2$, however, multiplying $m \oslash 10$ by 10 will miraculously restore m, provided exact rounding is being used. Actually, a more general fact (due to Kahan) is true. The proof is ingenious, but readers not interested in such details can skip ahead to Section 2.

Theorem 7

When $\beta = 2$, if m and n are integers with $|m| < 2^{p-1}$ and n has the special form $n = 2^i + 2^j$, then $(m \oslash n) \otimes n = m$, provided floating-point operations are exactly rounded.

Proof. Scaling by a power of 2 is harmless, since it changes only the exponent not the significand. If $q = m/n$, then scale n so that $2^{p-1} \leq n < 2^p$ and scale m so that $1/2 < q < 1$. Thus, $2^{p-2} < m < 2^p$. Since m has p significant bits, it has at most 1 bit to the right of the binary point. Changing the sign of m is harmless, so assume $q > 0$.

If $\bar{q} = m \oslash n$, to prove the theorem requires showing that

$$|n\bar{q} - m| \le \frac{1}{4}. \qquad (9)$$

That is because m has at most 1 bit right of the binary point, so $n\bar{q}$ will round to m. To deal with the halfway case when $|n\bar{q} - m| = 1/4$, note that since the initial unscaled m had $|m| < 2^{p-1}$, its low-order bit was 0, so the low-order bit of the scaled m is also 0. Thus, halfway cases will round to m.

Suppose $q = .q_1 q_2 \cdots$, and let $\hat{q} = .q_1 q_2 \cdots q_p 1$. To estimate $|n\bar{q} - m|$, first compute $|\hat{q} - q| = |N/2^{p+1} - m/n|$, where N is an odd integer. Since $n = 2^i + 2^j$ and $2^{p-1} \le n < 2^p$, it must be that $n = 2^{p-1} + 2^k$ for some $k \le p - 2$, and thus

$$|\hat{q} - q| = \left| \frac{nN - 2^{p+1}m}{n2^{p+1}} \right| = \left| \frac{(2^{p-1-k} + 1)N - 2^{p+1-k}m}{n2^{p+1-k}} \right|.$$

The numerator is an integer, and since N is odd, it is in fact an odd integer. Thus, $|\hat{q} - q| \ge 1/(n2^{p+1-k})$. Assume $q < \hat{q}$ (the case $q > \hat{q}$ is similar). Then $n\bar{q} < m$, and

$$|m - n\bar{q}| = m - n\bar{q} = n(q - \bar{q})$$
$$= n(q - (\hat{q} - 2^{-p-1}))$$
$$\le n\left(2^{-p-1} - \frac{1}{n2^{p+1-k}}\right)$$
$$= (2^{p-1} + 2^k)2^{-p-1} + 2^{-p-1+k} = \frac{1}{4}.$$

This establishes (9) and proves the theorem. ■

The theorem holds true for any base β, as long as $2^i + 2^j$ is replaced by $\beta^i + \beta^j$. As β gets larger, however, there are fewer and fewer denominators of the form $\beta^i + \beta^j$.

We are now in a position to answer the question, Does it matter if the basic arithmetic operations introduce a little more rounding error than necessary? The answer is that it does matter, because accurate basic operations enable us to prove that formulas are "correct" in the sense they have a small relative error. Section 1.4 discussed several algorithms that require guard digits to produce correct results in this sense. If the input to those formulas are numbers representing imprecise measurements, however, the bounds of Theorems 3 and 4 become less interesting. The reason is that the benign cancellation $x - y$ can become catastrophic if x and y are only approximations to some measured quantity. But accurate operations are useful even in the face of inexact data, because they enable us to establish exact relationships like those discussed in Theorems 6 and 7. These are useful even if every floating-point variable is only an approximation to some actual value.

2. IEEE STANDARD

There are two different IEEE standards for floating-point computation. IEEE 754 is a binary standard that requires $\beta = 2$, $p = 24$ for single precision and $p = 53$ for double precision [IEEE 1987]. It also specifies the precise layout of bits in a single and double precision. IEEE 854 allows either $\beta = 2$ or $\beta = 10$ and unlike 754, does not specify how floating-point numbers are encoded into bits [Cody et al. 1984]. It does not require a particular value for p, but instead it specifies constraints on the allowable values of p for single and double precision. The term *IEEE Standard* will be used when discussing properties common to both standards.

This section provides a tour of the IEEE standard. Each subsection discusses one aspect of the standard and why it was included. It is not the purpose of this paper to argue that the IEEE standard is the best possible floating-point standard but rather to accept the standard as given and provide an introduction to its use. For full details consult the standards [Cody et al. 1984; Cody 1988; IEEE 1987].

2.1 Formats and Operations

2.1.1 Base

It is clear why IEEE 854 allows $\beta = 10$. Base 10 is how humans exchange and think about numbers. Using $\beta = 10$ is especially appropriate for calculators, where the result of each operation is displayed by the calculator in decimal.

There are several reasons why IEEE 854 requires that if the base is not 10, it must be 2. Section 1.2 mentioned one reason: The results of error analyses are much tighter when β is 2 because a rounding error of 1/2 ulp wobbles by a factor of β when computed as a relative error, and error analyses are almost always simpler when based on relative error. A related reason has to do with the effective precision for large bases. Consider $\beta = 16$, $p = 1$ compared to $\beta = 2$, $p = 4$. Both systems have 4 bits of significand. Consider the computation of 15/8. When $\beta = 2$, 15 is represented as 1.111×2^3 and 15/8 as 1.111×2^0. So 15/8 is exact. When $\beta = 16$, however, 15 is represented as $F \times 16^0$, where F is the hexadecimal digit for 15. But 15/8 is represented as 1×16^0, which has only 1 bit correct. In general, base 16 can lose up to 3 bits, so a precision of p can have an effective precision as low as $4p - 3$ rather than $4p$.

Since large values of β have these problems, why did IBM choose $\beta = 16$ for its system/370? Only IBM knows for sure, but there are two possible reasons. The first is increased exponent range. Single precision on the system/370 has $\beta = 16$, $p = 6$. Hence the significand requires 24 bits. Since this must fit into 32 bits, this leaves 7 bits for the exponent and 1 for the sign bit. Thus, the magnitude of representable numbers ranges from about 16^{-2^6} to about $16^{2^6} = 2^{2^8}$. To get a similar exponent range when $\beta = 2$ would require 9 bits of exponent, leaving only 22 bits for the significand. It was just pointed out, however, that when $\beta = 16$, the effective precision can be as low as $4p - 3 = 21$ bits. Even worse, when $\beta = 2$ it is possible to gain an extra bit of precision (as explained later in this section), so the $\beta = 2$ machine has 23 bits of precision to compare with a range of 21–24 bits for the $\beta = 16$ machine.

Another possible explanation for choosing $\beta = 16$ bits has to do with shifting. When adding two floating-point numbers, if their exponents are different, one of the significands will have to be shifted to make the radix points line up, slowing down the operation. In the $\beta = 16$, $p = 1$ system, all the numbers between 1 and 15 have the same exponent, so no shifting is required when adding any of the $\binom{15}{2} = 105$ possible pairs of distinct numbers from this set. In the $\beta = 2$, $p = 4$ system, however, these numbers have exponents ranging from 0 to 3, and shifting is required for 70 of the 105 pairs.

In most modern hardware, the performance gained by avoiding a shift for a subset of operands is negligible, so the small wobble of $\beta = 2$ makes it the preferable base. Another advantage of using $\beta = 2$ is that there is a way to gain an extra bit of significance.[7] Since floating-point numbers are always normalized, the most significant bit of the significand is always 1, and there is no reason to waste a bit of storage representing it. Formats that use this trick are said to have a *hidden bit*. It was pointed out in Section 1.1 that this requires a special convention for 0. The method given there was that an exponent of $e_{\min} - 1$ and a significand of all zeros represent not $1.0 \times 2^{e_{\min} - 1}$ but rather 0.

IEEE 754 single precision is encoded in 32 bits using 1 bit for the sign, 8 bits for the exponent, and 23 bits for the significand. It uses a hidden bit, however, so the significand is 24 bits ($p = 24$), even though it is encoded using only 23 bits.

[7]This appears to have first been published by Goldberg [1967], although Knuth [1981 page 211] attributes this idea to Konrad Zuse.

2.1.2 Precision

The IEEE standard defines four different precisions: single, double, single extended, and double extended. In 754, single and double precision correspond roughly to what most floating-point hardware provides. Single precision occupies a single 32 bit word, double precision two consecutive 32 bit words. Extended precision is a format that offers just a little extra precision and exponent range (Table 1). The IEEE standard only specifies a lower bound on how many extra bits extended precision provides. The minimum allowable double-extended format is sometimes referred to as 80-*bit format*, even though the table shows it using 79 bits. The reason is that hardware implementations of extended precision normally do not use a hidden bit and so would use 80 rather than 79 bits.[8]

The standard puts the most emphasis on extended precision, making no recommendation concerning double precision but strongly recommending that

> Implementations should support the extended format corresponding to the widest basic format supported, . . .

One motivation for extended precision comes from calculators, which will often display 10 digits but use 13 digits internally. By displaying only 10 of the 13 digits, the calculator appears to the user ε ; a black box that computes exponentials, cosines, and so on, to 10 digits of accuracy. For the calculator to compute functions like exp, log, and cos to within 10 digits with reasonable efficiency, however, it needs a few extra digits with which to work. It is not hard to find a simple rational expression that approximates log with an error of 500 units in the last place. Thus, computing with 13 digits gives an answer correct to 10 digits. By keeping these extra 3 digits hidden, the calculator presents a simple model to the operator.

Extended precision in the IEEE standard serves a similar function. It enables libraries to compute quantities to within about 1/2 ulp in single (or double) precision efficiently, giving the user of those libraries a simple model, namely, that each primitive operation, be it a simple multiply or an invocation of log, returns a value accurate to within about 1/2 ulp. When using extended precision, however, it is important to make sure that its use is transparent to the user. For example, on a calculator, if the internal representation of a displayed value is not rounded to the same precision as the display, the result of further operations will depend on the hidden digits and appear unpredictable to the user.

To illustrate extended precision further, consider the problem of converting between IEEE 754 single precision and decimal. Ideally, single precision numbers will be printed with enough digits so that when the decimal number is read back in, the single precision number can be recovered. It turns out that 9 decimal digits are enough to recover a single precision binary number (see Section 4.2). When converting a decimal number back to its unique binary representation, a rounding error as small as 1 ulp is fatal because it will give the wrong answer. Here is a situation where extended precision is vital for an efficient algorithm. When single extended is available, a straightforward method exists for converting a decimal number to a single precision binary one. First, read in the 9 decimal digits as an integer N, ignoring the decimal point. From Table 1, $p \geq 32$, and since $10^9 < 2^{32} \approx 4.3 \times 10^9$, N can be represented exactly in single extended. Next, find the appropriate power 10^P necessary to scale N. This will be a combination of the exponent of the decimal number, and the position of the (up until now) ignored decimal point. Compute $10^{|P|}$. If $|P| \leq 13$, this is also represented exactly, because $10^{13} = 2^{13}5^{13}$ and $5^{13} < 2^{32}$. Finally, multiply (or divide if $P < 0$) N and $10^{|P|}$. If this

[8]According to Kahan, extended precision has 64 bits of significand because that was the widest precision across which carry propagation could be done on the Intel 8087 without increasing the cycle time [Kahan 1988].

Table 1. IEEE 754 Format Parameters

Parameter	Format			
	Single	Single Extended	Double	Double Extended
p	24	≥ 32	53	≥ 64
$e_{\max}$	$+127$	$\geq +1023$	$+1023$	$> +16383$
$e_{\min}$	-126	≤ -1022	-1022	≤ -16382
Exponent width in bits	8	≥ 11	11	≥ 15
Format width in bits	32	≥ 43	64	≥ 79

last operation is done exactly, the closest binary number is recovered. Section 4.2 shows how to do the last multiply (or divide) exactly. Thus, for $|P| \leq 13$, the use of the single-extended format enables 9 digit decimal numbers to be converted to the closest binary number (i.e., exactly rounded). If $|P| > 13$, single-extended is not enough for the above algorithm to compute the exactly rounded binary equivalent always, but Coonen [1984] shows that it is enough to guarantee that the conversion of binary to decimal and back will recover the original binary number.

If double precision is supported, the algorithm above would run in double precision rather than single-extended, but to convert double precision to a 17 digit decimal number and back would require the double-extended format.

2.1.3 Exponent

Since the exponent can be positive or negative, some method must be chosen to represent its sign. Two common methods of representing signed numbers are sign/magnitude and two's complement. Sign/magnitude is the system used for the sign of the significand in the IEEE formats: 1 bit is used to hold the sign; the rest of the bits represent the magnitude of the number. The two's complement representation is often used in integer arithmetic. In this scheme, a number is represented by the smallest nonnegative number that is congruent to it modulo 2^p.

The IEEE binary standard does not use either of these methods to represent the exponent but instead uses a *biased* representation. In the case of single precision, where the exponent is stored in 8 bits, the bias is 127 (for double precision it is 1023). What this means is that if $\bar{k}$ is the value of the exponent bits interpreted as an unsigned integer, then the exponent of the floating-point number is $\bar{k} - 127$. This is often called the *biased exponent* to distinguish from the unbiased exponent $\bar{k}$. An advantage of biased representation is that nonnegative flouting-point numbers can be treated as integers for comparison purposes.

Referring to Table 1, single precision has $e_{\max} = 127$ and $e_{\min} = -126$. The reason for having $|e_{\min}| < e_{\max}$ is so that the reciprocal of the smallest number $(1/2^{e_{\min}})$ will not overflow. Although it is true that the reciprocal of the largest number will underflow, underflow is usually less serious than overflow. Section 2.1.1 explained that $e_{\min} - 1$ is used for representing 0, and Section 2.2 will introduce a use for $e_{\max} + 1$. In IEEE single precision, this means that the biased exponents range between $e_{\min} - 1 = -127$ and $e_{\max} + 1 = 128$ whereas the unbiased exponents range between 0 and 255, which are exactly the nonnegative numbers that can be represented using 8 bits.

2.1.4 Operations

The IEEE standard requires that the result of addition, subtraction, multiplication, and division be exactly rounded. That is, the result must be computed exactly then rounded to the nearest floating-point number (using round to even). Section 1.3 pointed out that computing the exact difference or sum of two floating-point numbers can be very expensive when their exponents are substantially different. That section introduced guard digits, which provide a practical way of computing differences while guaranteeing that the relative error is small. Computing with a single guard digit, however, will not always give the same answer as computing the exact result then rounding. By introducing a second guard digit and a third *sticky* bit, differences can be computed at only a little more cost than with a single guard digit, but the result is the same as if the difference were computed exactly then rounded [Goldberg 1990]. Thus, the standard can be implemented efficiently.

One reason for completely specifying the results of arithmetic operations is to improve the portability of software. When a program is moved between two machines and both support IEEE arithmetic, if any intermediate result differs, it must be because of software bugs not differences in arithmetic. Another advantage of precise specification is that it makes it easier to reason about floating point. Proofs about floating point are hard enough without having to deal with multiple cases arising from multiple kinds of arithmetic. Just as integer programs can be proven to be correct, so can floating-point programs, although what is proven in that case is that the rounding error of the result satisfies certain bounds. Theorem 4 is an example of such a proof. These proofs are made much easier when the operations being reasoned about are precisely specified. Once an algorithm is proven to be correct for IEEE arithmetic, it will work correctly on any machine supporting the IEEE standard.

Brown [1981] has proposed axioms for floating point that include most of the existing floating-point hardware. Proofs in this system cannot, however, verify the algorithms of Sections 1.4 and 1.5, which require features not present on all hardware. Furthermore, Brown's axioms are more complex than simply defining operations to be performed exactly then rounded. Thus, proving theorems from Brown's axioms is usually more difficult than proving them assuming operations are exactly rounded.

There is not complete agreement on what operations a floating-point standard should cover. In addition to the basic operations $+$, $-$, $\times$, and $/$, the IEEE standard also specifies that square root, remainder, and conversion between integer and floating point be correctly rounded. It also requires that conversion between internal formats and decimal be correctly rounded (except for very large numbers). Kulisch and Miranker [1986] have proposed adding inner product to the list of operations that are precisely specified. They note that when inner products are computed in IEEE arithmetic, the final answer can be quite wrong. For example, sums are a special case of inner products, and the sum $((2 \times 10^{-30} + 10^{30}) - 10^{-30}) - 10^{30}$ is exactly equal to 10^{-30}, but on a machine with IEEE arithmetic the computed result will be -10^{-30}. It is possible to compute inner products to within 1 ulp with less hardware than it takes to implement a fast multiplier [Kirchner and Kulisch 1987].[9]

All the operations mentioned in the standard, except conversion between decimal and binary, are required to be exactly rounded. The reason is that efficient algorithms for exactly rounding all the operations, except conversion, are known. For conversion, the best known efficient algorithms produce results that are slightly worse than exactly rounded ones [Coonen 1984].

The IEEE standard does not require transcendental functions to be exactly rounded because of the *table maker's dilemma*. To illustrate, suppose you are making a table of the exponential function to four places. Then exp(1.626) = 5.0835. Should this be rounded to 5.083 or 5.084? If exp(1.626) is computed more carefully, it becomes 5.08350, then

[9]Some arguments against including inner product as one of the basic operations are presented by Kahan and LeBlanc [1985].

5.083500, then 5.0835000. Since exp is transcendental, this could go on arbitrarily long before distinguishing whether exp(1.626) is $5.083500 \cdots 0ddd$ or $5.0834999 \cdots 9ddd$. Thus, it is not practical to specify that the precision of transcendental functions be the same as if the functions were computed to infinite precision then rounded. Another approach would be to specify transcendental functions algorithmically. But there does not appear to be a single algorithm that works well across all hardware architectures. Rational approximation, CORDIC,[10] and large tables are three different techniques used for computing transcendentals on contemporary machines. Each is appropriate for a different class of hardware, and at present no single algorithm works acceptably over the wide range of current hardware.

2.2 Special Quantities

On some floating-point hardware every bit pattern represents a valid floating-point number. The IBM System/370 is an example of this. On the other hand, the VAX reserves some bit patterns to represent special numbers called *reserved operands*. This idea goes back to the CDC 6600, which had bit patterns for the special quantities INDEFINITE and INFINITY.

The IEEE standard continues in this tradition and has NaNs (Not a Number, pronounced to rhyme with plan) and infinities. Without special quantities, there is no good way to handle exceptional situations like taking the square root of a negative number other than aborting computation. Under IBM System/370 FORTRAN, the default action in response to computing the square root of a negative number like -4 results in the printing of an error message. Since every bit pattern represents a valid number, the return value of square root must be some floating-point number. In the case of System/370 FORTRAN, $\sqrt{|-4|} = 2$ is returned. In IEEE arithmetic, an NaN is returned in this situation.

Table 2. IEEE 754 Special Values

Exponent	Fraction	Represents
$e = e_{\min} - 1$	$f = 0$	± 0
$e = e_{\min} - 1$	$f \neq 0$	$0.f \times 2^{e_{\min}}$
$e_{\min} \le e \le e_{\max}$	—	$1.f \times 2^{e}$
$e = e_{\max} + 1$	$f = 0$	$\pm \infty$
$e = e_{\max} + 1$	$f \neq 0$	NaN

The IEEE standard specifies the following special values (see Table 2): ± 0, denormalized numbers, $\pm\infty$ and NaNs (there is more than one NaN, as explained in the next section). These special values are all encoded with exponents of either $e_{\max} + 1$ or $e_{\min} - 1$ (it was already pointed out that 0 has an exponent of $e_{\min} - 1$).

2.2.1 NaNs

Traditionally, the computation of 0/0 or $\sqrt{-1}$ has been treated as an unrecoverable error that causes a computation to halt. There are, however, examples for which it makes sense for a computation to continue in such a situation. Consider a subroutine that finds the zeros of a function f, say **zero(f)**. Traditionally, zero finders require the user to input an interval $[a,b]$ on which the function is defined and over which the zero finder will search. That is, the subroutine is called as **zero(f, a, b)**. A more useful zero finder would not require the user to input this extra information. This more general zero finder is especially appropriate for calculators, where it is natural to key in a function and awkward to then have to specify the domain. It is easy, however, to see why most zero finders require a domain. The zero finder does its work by probing the function **f** at various values. If it probed for a value outside the domain of **f**, the code for **f**

[10]CORDIC is an acronym for Coordinate Rotation Digital Computer and is a method of computing transcendental functions that uses mostly shifts and adds (i.e., very few multiplications and divisions) [Walther 1971]. It is the method used on both the Intel 8087 and the Motorola 68881.

Table 3. Operations that Produce an NaN

Operation	NaN Produced by
$+$	$\infty + (-\infty)$
$\times$	$0 \times \infty$
$/$	$0/0, \infty/\infty$
REM	x REM 0, ∞ REM y
$\sqrt{\ }$	$\sqrt{x}$ (when $x < 0$)

might well compute $0/0$ or $\sqrt{-1}$, and the computation would halt, unnecessarily aborting the zero finding process.

This problem can be avoided by introducing a special value called NaN and specifying that the computation of expressions like $0/0$ and $\sqrt{-1}$ produce NaN rather than halting. (A list of some of the situations that can cause a NaN is given in Table 3.) Then, when **zero(f)** probes outside the domain of **f**, the code for **f** will return NaN and the zero finder can continue. That is, **zero(f)** is not "punished" for making an incorrect guess. With this example in mind, it is easy to see what the result of combining a NaN with an ordinary floating-point number should be. Suppose the final statement of **f** is **return(− b + sqrt(d))/(2 ∗ *a*)**. If $d < 0$, then **f** should return a NaN. Since $d < 0$, **sqrt(d)** is an NaN, and **−b + sqrt(d)** will be a NaN if the sum of an NaN and any other number is a NaN. Similarly, if one operand of a division operation is an NaN, the quotient should be a NaN. In general, whenever a NaN participates in a floating-point operation, the result is another NaN.

Another approach to writing a zero solver that does not require the user to input a domain is to use signals. The zero finder could install a signal handler for floating-point exceptions. Then if **f** were evaluated outside its domain and raised an exception, control would be returned to the zero solver. The problem with this approach is that every language has a different method of handling signals (if it has a method at all), and so it has no hope of portability.

In IEEE 754, NaNs are represented as floating-point numbers with the exponent $e_{\max} + 1$ and nonzero significands. Implementations are free to put system-dependent information into the significand. Thus, there is not a unique NaN but rather a whole family of NaNs. When an NaN and an ordinary floating-point number are combined, the result should be the same as the NaN operand. Thus, if the result of a long computation is an NaN, the system-dependent information in the significand will be the information generated when the first NaN in the computation was generated. Actually, there is a caveat to the last statement. If both operands are NaNs, the result will be one of those NaNs but it might not be the NaN that was generated first.

2.2.2 Infinity

Just as NaNs provide a way to continue a computation when expressions like $0/0$ or $\sqrt{-1}$ are encountered, infinities provide a way to continue when an overflow occurs. This is much safer than simply returning to the largest representable number. As an example, consider computing $\sqrt{x^2 + y^2}$, when $\beta = 10$, $p = 3$, and $e_{\max} = 98$. If $x = 3 \times 10^{70}$ and $y = 4 \times 10^{70}$, then x^2 will overflow and be replaced by 9.99×10^{98}. Similarly y^2 and $x^2 + y^2$ will each overflow in turn and be replaced by 9.99×10^{98}. So the final result will be $(9.99 \times 10^{98})^{1/2} = 3.16 \times 10^{49}$, which is drastically wrong. The correct answer is 5×10^{70}. In IEEE arithmetic, the result of x^2 is ∞, as is y^2, $x^2 + y^2$, and $\sqrt{x^2 + y^2}$. So the final result is ∞, which is safer than returning an ordinary floating-point number that is nowhere near the correct answer.[11]

The division of 0 by 0 results in an NaN. A nonzero number divided by 0, however, returns infinity: $1/0 = \infty$, $-1/0 = -\infty$. The reason for the distinction is this: If $f(x) \to 0$ and $g(x) \to 0$ as

[11]Fine point: Although the default in IEEE arithmetic is to round overflowed numbers to ∞, it is possible to change the default (see Section 2.3.2).

x approaches some limit, then $f(x)/g(x)$ could have any value. For example, when $f(x) = \sin x$ and $g(x) = x$, then $f(x)/g(x) \to 1$ as $x \to 0$. But when $f(x) = 1 - \cos x$, $f(x)/g(x) \to 0$. When thinking of 0/0 as the limiting situation of a quotient of two very small numbers, 0/0 could represent anything. Thus, in the IEEE standard, 0/0 results in an NaN. But when $c > 0$ and $f(x) \to c$, $g(x) \to 0$, then $f(x)/g(x) \to \pm\infty$ for any analytic functions f and g. If $g(x) < 0$ for small x, then $f(x)/g(x) \to -\infty$; otherwise the limit is $+\infty$. So the IEEE standard defines $c/0 = \pm\infty$ as long as $c \neq 0$. The sign of ∞ depends on the signs of c and 0 in the usual way, so $-10/0 = -\infty$ and $-10/-0 = +\infty$. You can distinguish between getting ∞ because of overflow and getting ∞ because of division by 0 by checking the status flags (which will be discussed in detail in Section 2.3.3). The overflow flag will be set in the first case, the division by 0 flag in the second.

The rule for determining the result of an operation that has infinity as an operand is simple: Replace infinity with a finite number x and take the limit as $x \to \infty$. Thus, $3/\infty = 0$, because $\lim_{x \to \infty} 3/x = 0$. Similarly $4 - \infty = -\infty$ and $\sqrt{\infty} = \infty$. When the limit does not exist, the result is an NaN, so ∞/∞ will be an NaN (Table 3 has additional examples). This agrees with the reasoning used to conclude that 0/0 should be an NaN.

When a subexpression evaluates to a NaN, the value of the entire expression is also a NaN. In the case of $\pm\infty$, however, the value of the expression might be an ordinary floating-point number because of rules like $1/\infty = 0$. Here is a practical example that makes use of the rules for infinity arithmetic. Consider computing the function $x/(x^2 + 1)$. This is a bad formula, because not only will it overflow when x is larger than $\sqrt{\beta}\beta^{e_{\max}/2}$, but infinity arithmetic will give the wrong answer because it will yield 0 rather than a number near $1/x$. However, $x/(x^2 + 1)$ can be rewritten as $1/(x + x^{-1})$. This improved expression will not overflow prematurely and because of infinity arithmetic will have the correct value when $x = 0$: $1/(0 + 0^{-1}) = 1/(0 + \infty) = 1/\infty = 0$. Without infinity arithmetic, the expression $1/(x + x^{-1})$ requires a test for $x = 0$, which not only adds extra instructions but may also disrupt a pipeline. This example illustrates a general fact; namely, that infinity arithmetic often avoids the need for special case checking; however, formulas need to be carefully inspected to make sure they do not have spurious behavior at infinity [as $x/(x^2 + 1)$ did].

2.2.3 Signed Zero

Zero is represented by the exponent $e_{\min} - 1$ and a zero significand. Since the sign bit can take on two different values, there are two zeros, $+0$ and -0. If a distinction were made when comparing $+0$ and -0, simple tests like **if (x = 0)** would have unpredictable behavior, depending on the sign of **x**. Thus, the IEEE standard defines comparison so that $+0 = -0$ rather than $-0 < +0$. Although it would be possible always to ignore the sign of zero, the IEEE standard does not do so. When a multiplication or division involves a signed zero, the usual sign rules apply in computing the sign of the answer. Thus, $3(+0) = +0$ and $+0/-3 = -0$. If zero did not have a sign, the relation $1/(1/x) = x$ would fail to hold when $x = \pm\infty$. The reason is that $1/-\infty$ and $1/+\infty$ both result in 0, and 1/0 results in $+\infty$, the sign information having been lost. One way to restore the identity $1/(1/x) = x$ is to have only one kind of infinity; however, that would result in the disastrous consequence of losing the sign of an overflowed quantity.

Another example of the use of signed zero concerns underflow and functions that have a discontinuity at zero such as log. In IEEE arithmetic, it is natural to define $\log 0 = -\infty$ and $\log x$ to be an NaN when $x < 0$. Suppose x represents a small negative number that has underflowed to zero. Thanks to signed zero, x will be negative so log can return an NaN. If there were no signed zero, however, the log function could not distinguish an underflowed negative number from 0 and would therefore have to return $-\infty$. Another example of a function with a discontinuity at zero is the signum function, which returns the sign of a number.

Probably the most interesting use of signed zero occurs in complex arithmetic. As an example, consider the equation $\sqrt{1/z} = 1/\sqrt{z}$. This is certainly true when $z \geq 0$. If $z = -1$, the obvious computation gives $\sqrt{1/-1} = \sqrt{-1} = i$ and $1/\sqrt{-1} = 1/i = -i$. Thus, $\sqrt{1/z} \neq 1/\sqrt{z}$! The problem can be traced to the fact that square root is multivalued, and there is no way to select the values so they are continuous in the entire complex plane. Square root is continuous, however, if a *branch cut* consisting of all negative real numbers is excluded from consideration. This leaves the problem of what to do for the negative real numbers, which are of the form $-x + i0$, where $x > 0$. Signed zero provides a perfect way to resolve this problem. Numbers of the form $-x + i(+0)$ have a square root of $i\sqrt{x}$, and numbers of the form $-x + i(-0)$ on the other side of the branch cut have a square root with the other sign $(-i\sqrt{x})$. In fact, the natural formulas for computing $\sqrt{}$ will give these results.

Let us return to $\sqrt{1/z} = 1/\sqrt{z}$. If $z = -1 = -1 + i0$, then

$$\begin{aligned} 1/z &= 1/(-1 + i0) \\ &= \frac{1(-1 - i0)}{(-1 + i0)(-1 - i0)} \\ &= (-1 - i0)/((-1)^2 - 0^2) \\ &= -1 + i(-0), \end{aligned}$$

so $\sqrt{1/z} = \sqrt{-1 + i(-0)} = -i$, while $1/\sqrt{z} = 1/i = -i$. Thus, IEEE arithmetic preserves this identity for all z. Some more sophisticated examples are given by Kahan [1987]. Although distinguishing between $+0$ and -0 has advantages, it can occasionally be confusing. For example, signed zero destroys the relation $x = y \Leftrightarrow 1/x = 1/y$, which is false when $x = +0$ and $y = -0$. The IEEE committee decided, however, that the advantages of using signed zero outweighed the disadvantages.

2.2.4 *Denormalized Numbers*

Consider normalized floating-point numbers with $\beta = 10$, $p = 3$, and $e_{\min} = -98$. The numbers $x = 6.87 \times 10^{-97}$ and $y = 6.81 \times 10^{-97}$ appear to be perfectly ordinary floating-point numbers, which are more than a factor of 10 larger than the smallest floating-point number 1.00×10^{-98}. They have a strange property, however: $x \ominus y = 0$ even though $x \neq y$! The reason is that $x - y = .06 \times 10^{-97} = 6.0 \times 10^{-99}$ is too small to be represented as a normalized number and so must be flushed to zero.

How important is it to preserve the property

$$x = y \Leftrightarrow x - y = 0? \qquad (10)$$

It is very easy to imagine writing the code fragment **if (x ≠ y) then z = 1/(x − y)** and later having a program fail due to a spurious division by zero. Tracking down bugs like this is frustrating and time consuming. On a more philosophical level, computer science textbooks often point out that even though it is currently impractical to prove large programs correct, designing programs with the idea of proving them often results in better code. For example, introducing invariants is useful, even if they are not going to be used as part of a proof. Floating-point code is just like any other code: It helps to have provable facts on which to depend. For example, when analyzing formula (7), it will be helpful to know that $x/2 < y < 2x \Rightarrow x \ominus y = x - y$ (see Theorem 11). Similarly, knowing that (10) is true makes writing reliable floating-point code easier. If it is only true for most numbers, it cannot be used to prove anything.

The IEEE standard uses denormalized[12] numbers, which guarantee (10), as

[12]They are called *subnormal* in 854, *denormal* in 754.

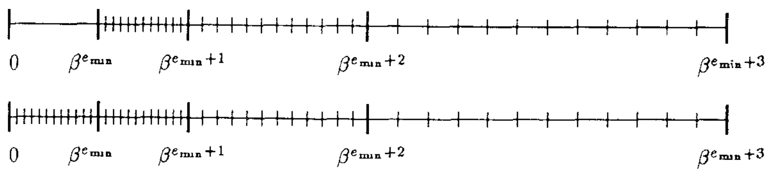


Figure 2. Flush to zero compared with gradual underflow.

well as other useful relations. They are the most controversial part of the standard and probably accounted for the long delay in getting 754 approved. Most high-performance hardware that claims to be IEEE compatible does not support denormalized numbers directly but rather traps when consuming or producing denormals, and leaves it to software to simulate the IEEE standard.[13] The idea behind denormalized numbers goes back to Goldberg [1967] and is simple. When the exponent is $e_{\min}$, the significand does not have to be normalized. For example, when $\beta = 10$, $p = 3$, and $e_{\min} = -98$, 1.00×10^{-98} is no longer the smallest floating-point number, because 0.98×10^{-98} is also a floating-point number.

There is a small snag when $\beta = 2$ and a hidden bit is being used, since a number with an exponent of $e_{\min}$ will always have a significand greater than or equal to 1.0 because of the implicit leading bit. The solution is similar to that used to represent 0 and is summarized in Table 2. The exponent $e_{\min} - 1$ is used to represent denormals. More formally, if the bits in the significant field are b_1, $b_2, \ldots, b_{p-1}$ and the value of the exponent is e, then when $e > e_{\min} - 1$, the number being represented is $1.b_1 b_2 \cdots b_{p-1} \times 2^e$, whereas when $e = e_{\min} - 1$, the number being represented is $0.b_1 b_2 \cdots b_{p-1} \times 2^{e+1}$. The $+1$ in the exponent is needed because denormals have an exponent of $e_{\min}$, not $e_{\min} - 1$.

[13]This is the cause of one of the most troublesome aspects of the standard. Programs that frequently underflow often run noticeably slower on hardware that uses software traps.

Recall the example $\beta = 10$, $p = 3$, $e_{\min} = -98$, $x = 6.87 \times 10^{-97}$, and $y = 6.81 \times 10^{-97}$ presented at the beginning of this section. With denormals, $x - y$ does not flush to zero but is instead represented by the denormalized number $.6 \times 10^{-98}$. This behavior is called *gradual underflow*. It is easy to verify that (10) always holds when using gradual underflow.

Figure 2 illustrates denormalized numbers. The top number line in the figure shows normalized floating-point numbers. Notice the gap between 0 and the smallest normalized number $1.0 \times \beta^{e_{\min}}$. If the result of a floating-point calculation falls into this gulf, it is flushed to zero. The bottom number line shows what happens when denormals are added to the set of floating-point numbers. The "gulf" is filled in; when the result of a calculation is less than $1.0 \times \beta^{e_{\min}}$, it is represented by the nearest denormal. When denormalized numbers are added to the number line, the spacing between adjacent floating-point numbers varies in a regular way: Adjacent spacings are either the same length or differ by a factor of β. Without denormals, the spacing abruptly changes from $\beta^{-p+1}\beta^{e_{\min}}$ to $\beta^{e_{\min}}$, which is a factor of β^{p-1}, rather than the orderly change by a factor of β. Because of this, many algorithms that can have large relative error for normalized numbers close to the underflow threshold are well behaved in this range when gradual underflow is used.

Without gradual underflow, the simple expression $x + y$ can have a very large relative error for normalized inputs, as was seen above for $x = 6.87 \times 10^{-97}$ and $y = 6.81 \times 10^{-97}$. Large relative errors can happen even without cancellation, as the following example shows [Demmel 1984]. Consider dividing two complex numbers, $a + ib$ and $c + id$. The obvious formula

$$\frac{a+ib}{c+id} = \frac{ac+bd}{c^2+d^2} + i\frac{bc-ad}{c^2+d^2}$$

suffers from the problem that if either component of the denominator $c + id$ is larger than $\sqrt{\beta}\,\beta^{e_{\max}/2}$, the formula will overflow even though the final result may be well within range. A better method of computing the quotients is to use Smith's formula:

$$\frac{a+ib}{c+id} = \begin{cases} \dfrac{a+b(d/c)}{c+d(d/c)} + i\dfrac{b-a(d/c)}{c+d(d/c)} \\ \quad \text{if } |d| < |c| \\ \dfrac{b+a(c/d)}{d+c(c/d)} + i\dfrac{-a+b(c/d)}{d+c(c/d)} \\ \quad \text{if } |d| \geq |c|. \end{cases} \tag{11}$$

Applying Smith's formula to

$$\frac{2\cdot 10^{-98} + i10^{-98}}{4\cdot 10^{-98} + i(2\cdot 10^{-98})}$$

gives the correct answer of 0.5 with gradual underflow. It yields 0.4 with flush to zero, an error of 100 ulps. It is typical for denormalized numbers to guarantee error bounds for arguments all the way down to $1.0 \times \beta^{e_{\min}}$.

2.3 Exceptions, Flags, and Trap Handlers

When an exceptional condition like division by zero or overflow occurs in IEEE arithmetic, the default is to deliver a result and continue. Typical of the default results are NaN for 0/0 and $\sqrt{-1}$ and ∞ for 1/0 and overflow. The preceding sections gave examples where proceeding from an exception with these default values was the reasonable thing to do. When any exception occurs, a status flag is also set. Implementations of the IEEE standard are required to provide users with a way to read and write the status flags. The flags are "sticky" in that once set, they remain set until explicitly cleared. Testing the flags is the only way to distinguish 1/0, which is a genuine infinity from an overflow.

Sometimes continuing execution in the face of exception conditions is not appropriate. Section 2.2.2 gave the example of $x/(x^2 + 1)$. When $x > \sqrt{\beta}\,\beta^{e_{\max}/2}$, the denominator is infinite, resulting in a final answer of 0, which is totally wrong. Although for this formula the problem can be solved by rewriting it as $1/(x + x^{-1})$, rewriting may not always solve the problem. The IEEE standard strongly recommends that implementations allow *trap handlers* to be installed. Then when an exception occurs, the trap handler is called instead of setting the flag. The value returned by the trap handler will be used as the result of the operation. It is the responsibility of the trap handler to either clear or set the status flag; otherwise, the value of the flag is allowed to be undefined.

The IEEE standard divides exceptions into five classes: overflow, underflow, division by zero, invalid operation, and inexact. There is a separate status flag for each class of exception. The meaning of the first three exceptions is self-evident. Invalid operation covers the situations listed in Table 3. The default result of an operation that causes an invalid exception is to return an NaN, but the converse is not true. When one of the operands to an operation is an NaN, the result is an NaN, but an invalid exception is not raised unless the operation also satisfies one of the conditions in Table 3.

The inexact exception is raised when the result of a floating-point operation is not exact. In the $\beta = 10$, $p = 3$ system, $3.5 \otimes 4.2 = 14.7$ is exact, but $3.5 \otimes 4.3 = 15.0$ is not exact (since $3.5 \cdot 4.3 = 15.05$) and raises an inexact exception. Section 4.2 discusses an algorithm that uses the inexact exception. A summary of the behavior of all five exceptions is given in Table 4.

There is an implementation issue connected with the fact that the inexact exception is raised so often. If floating-point

Table 4. Exceptions in IEEE 754[a]

Exception	Result When Traps Disabled	Argument to Trap Handler
Overflow	$\pm\infty$ or $\pm x_{\max}$	Round($x2^{-\alpha}$)
Underflow	0, $\pm 2^{e_{\min}}$ or denormal	Round($x2^{\alpha}$)
Divide by zero	$\pm\infty$	Operands
Invalid	NaN	Operands
Inexact	round(x)	round(x)

[a] x Is the exact result of the operation, $\alpha = 192$ for single precision, 1536 for double, and $x_{\max} = 1.11 \cdots 11 \times 2^{e_{\max}}$.

hardware does not have flags of its own but instead interrupts the operating system to signal a floating-point exception, the cost of inexact exceptions could be prohibitive. This cost can be avoided by having the status flags maintained by software. The first time an exception is raised, set the software flag for the appropriate class and tell the floating-point hardware to mask off that class of exceptions. Then all further exceptions will run without interrupting the operating system. When a user resets that status flag, the hardware mask is reenabled.

2.3.1 Trap Handlers

One obvious use for trap handlers is for backward compatibility. Old codes that expect to be aborted when exceptions occur can install a trap handler that aborts the process. This is especially useful for codes with a loop like **do S until (x > = 100)**. Since comparing a NaN to a number with $<$, $\le$, $>$, $\ge$, or $=$ (but not $\ne$) always returns false, this code will go into an infinite loop if **x** ever becomes an NaN.

There is a more interesting use for trap handlers that comes up when computing products such as $\Pi_{i=1}^{n} x_i$ that could potentially overflow. One solution is to use logarithms and compute $\exp(\sum \log x_i)$ instead. The problems with this approach are that it is less accurate and costs more than the simple expression Πx_i, even if there is no overflow. There is another solution using trap handlers called *over/underflow counting* that avoids both of these problems [Sterbenz 1974].

The idea is as follows: There is a global counter initialized to zero. Whenever the partial product $p_k = \Pi_{i=1}^{k} x_i$ overflows for some k, the trap handler increments the counter by 1 and returns the overflowed quantity with the exponent wrapped around. In IEEE 754 single precision, $e_{\max} = 127$, so if $p_k = 1.45 \times 2^{130}$, it will overflow and cause the trap handler to be called, which will wrap the exponent back into range, changing p_k to 1.45×2^{-62} (see below). Similarly, if p_k underflows, the counter would be decremented and the negative exponent would get wrapped around into a positive one. When all the multiplications are done, if the counter is zero, the final product is p_n. If the counter is positive, the product is overflowed; if the counter is negative, it underflowed. If none of the partial products is out of range, the trap handler is never called and the computation incurs no extra cost. Even if there are over/underflows, the calculation is more accurate than if it had been computed with logarithms, because each p_k was computed from p_{k-1} using a full-precision multiply. Barnett [1987] discusses a formula where the full accuracy of over/underflow counting turned up an error in earlier tables of that formula.

IEEE 754 specifies that when an overflow or underflow trap handler is called, it is passed the wrapped-around result as an argument. The definition of wrapped around for overflow is that the result is computed as if to infinite precision, then divided by 2^{α}, then rounded to the relevant precision. For underflow, the result is multiplied by 2^{α}. The exponent α is 192 for single precision and 1536 for double precision. This is why 1.45×2^{130} was transformed into 1.45×2^{-62} in the example above.

2.3.2 Rounding Modes

In the IEEE standard, rounding occurs whenever an operation has a result that is not exact, since (with the exception of binary decimal conversion) each operation is computed exactly then rounded. By default, rounding means round toward nearest. The standard requires that three other rounding modes be provided; namely, round toward 0, round toward $+\infty$, and round toward $-\infty$. When used with the convert to integer operation, round toward $-\infty$ causes the convert to become the floor function, whereas, round toward $+\infty$ is ceiling. The rounding mode affects overflow because when round toward 0 or round toward $-\infty$ is in effect, an overflow of positive magnitude causes the default result to be the largest representable number, not $+\infty$. Similarly, overflows of negative magnitude will produce the largest negative number when round toward $+\infty$ or round toward 0 is in effect.

One application of rounding modes occurs in interval arithmetic (another is mentioned in Section 4.2). When using interval arithmetic, the sum of two numbers x and y is an interval $[\underline{z}, \bar{z}]$, where $\underline{z}$ is $x \oplus y$ rounded toward $-\infty$ and $\bar{z}$ is $x \oplus y$ rounded toward $+\infty$. The exact result of the addition is contained within the interval $[\underline{z}, \bar{z}]$. Without rounding modes, interval arithmetic is usually implemented by computing $\underline{z} = (x \oplus y)(1 - \epsilon)$ and $\bar{z} = (x \oplus y)(1 + \epsilon)$, where ϵ is machine epsilon. This results in overestimates for the size of the intervals. Since the result of an operation in interval arithmetic is an interval, in general the input to an operation will also be an interval. If two intervals $[\underline{x}, \bar{x}]$ and $[\underline{y}, \bar{y}]$ are added, the result is $[\underline{z}, \bar{z}]$, where $\underline{z}$ is $\underline{x} \oplus \underline{y}$ with the rounding mode set to round toward $-\infty$, and $\bar{z}$ is $\bar{x} \oplus \bar{z}$ with the rounding mode set toward $+\infty$.

When a floating-point calculation is performed using interval arithmetic, the final answer is an interval that contains the exact result of the calculation. This is not very helpful if the interval turns out to be large (as it often does), since the correct answer could be anywhere in that interval. Interval arithmetic makes more sense when used in conjunction with a multiple precision floating-point package. The calculation is first performed with some precision p. If interval arithmetic suggests that the final answer may be inaccurate, the computation is redone with higher and higher precisions until the final interval is a reasonable size.

2.3.3 Flags

The IEEE standard has a number of flags and modes. As discussed above, there is one status flag for each of the five exceptions: underflow, overflow, division by zero, invalid operation, and inexact. There are four rounding modes: round toward nearest, round toward $+\infty$, round toward 0, and round toward $-\infty$. It is strongly recommended that there be an enable mode bit for each of the five exceptions. This section gives some examples of how these modes and flags can be put to good use. A more sophisticated example is discussed in Section 4.2.

Consider writing a subroutine to compute x^n, where n is an integer. When $n > 0$, a simple routine like

```
PositivePower(x,n) {
  while (n is even) {
    x = x*x
    n = n/2
  }
  u = x
  while (true) {
    n = n/2
    if (n == 0) return u
    x = x*x
    if (n is odd) u = u*x
  }
}
```

will compute x^n.

If $n < 0$, the most accurate way to compute x^n is not to call **PositivePower(1/x, −n)** but rather **1/PositivePower(x, −n)**, because the first expression multiplies n quantities, each of which has a rounding error from the division (i.e., $1/x$). In the second expression these are exact (i.e., x) and the final division commits just one additional rounding error. Unfortunately, there is a slight snag in this strategy. If **PositivePower(x, −n)** underflows, then either the underflow trap handler will be called or the underflow status flag will be set. This is incorrect, because if x^{-n} underflows, then x^n will either overflow or be in range.[14] But since the IEEE standard gives the user access to all the flags, the subroutine can easily correct for this. It turns off the overflow and underflow trap enable bits and saves the overflow and underflow status bits. It then computes **1/PositivePower(x, −n)**. If neither the overflow nor underflow status bit is set, it restores them together with the trap enable bits. If one of the status bits is set, it restores the flags and redoes the calculation using **PositivePower (1/x, −n)**, which causes the correct exceptions to occur.

Another example of the use of flags occurs when computing arccos via the formula

$$\arccos x = 2 \arctan \sqrt{\frac{1-x}{1+x}}\,.$$

If arctan(∞) evaluates to $\pi/2$, then arccos(−1) will correctly evaluate to $2\arctan(\infty) = \pi$ because of infinity arithmetic. There is a small snag, however, because the computation of $(1-x)/(1+x)$ will cause the divide by zero exception flag to be set, even though arccos(−1) is not exceptional. The solution to this problem is straightforward. Simply save the value of the divide by zero flag before computing arccos, then restore its old value after the computation.

3. SYSTEMS ASPECTS

The design of almost every aspect of a computer system requires knowledge about floating point. Computer architectures usually have floating-point instructions, compilers must generate those floating-point instructions, and the operating system must decide what to do when exception conditions are raised for those floating-point instructions. Computer system designers rarely get guidance from numerical analysis texts, which are typically aimed at users and writers of software not at computer designers.

As an example of how plausible design decisions can lead to unexpected behavior, consider the following BASIC program:

```
q = 3.0/7.0
if q = 3.0/7.0 then print "Equal":
  else print "Not Equal"
```

When compiled and run using Borland's Turbo Basic[15] on an IBM PC, the program prints **Not Equal!** This example will be analyzed in Section 3.2.1.

Incidentally, some people think that the solution to such anomalies is never to compare floating-point numbers for equality but instead to consider them equal if they are within some error bound E. This is hardly a cure all, because it raises as many questions as it answers. What should the value of E be? If $x < 0$ and $y > 0$ are within E, should they really be considered equal, even though they have different signs? Furthermore, the relation defined by this rule, $a \sim b \Leftrightarrow |a-b| < E$, is not an equivalence relation because $a \sim b$ and $b \sim c$ do not imply that $a \sim c$.

3.1 Instruction Sets

It is common for an algorithm to require a short burst of higher precision in order to produce accurate results. One example occurs in the quadratic formula $[-b \pm \sqrt{b^2 - 4ac}]/2a$. As discussed in Section 4.1, when $b^2 \approx 4ac$, rounding error can contaminate up to half the digits in the roots computed with the quadratic

[14]It can be in range because if $x < 1$, $n < 0$, and x^{-n} is just a tiny bit smaller than the underflow threshold $2^{e_{\min}}$, then $x^n \approx 2^{-e_{\min}} < 2^{e_{\max}}$ and so may not overflow, since in all IEEE precisions, $-e_{\min} < e_{\max}$.

[15]Turbo Basic is a registered trademark of Borland International, Inc.

formula. By performing the subcalculation of $b^2 - 4ac$ in double precision, half the double precision bits of the root are lost, which means that all the single precision bits are preserved.

The computation of $b^2 - 4ac$ in double precision when each of the quantities a, b, and c are in single precision is easy if there is a multiplication instruction that takes two single precision numbers and produces a double precision result. To produce the exactly rounded product of two p-digit numbers, a multiplier needs to generate the entire $2p$ bits of product, although it may throw bits away as it proceeds. Thus, hardware to compute a double-precision product from single-precision operands will normally be only a little more expensive than a single-precision multiplier and much less expensive than a double-precision multiplier. Despite this, modern instruction sets tend to provide only instructions that produce a result of the same precision as the operands.[16]

If an instruction that combines two single-precision operands to produce a double-precision product were only useful for the quadratic formula, it would not be worth adding to an instruction set. This instruction has many other uses, however. Consider the problem of solving a system of linear equations:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n, \end{aligned}$$

which can be written in matrix form as $Ax = b$, where

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}.$$

[16]This is probably because designers like "orthogonal" instruction sets, where the precisions of a floating-point instruction are independent of the actual operation. Making a special case for multiplication destroys this orthogonality.

Suppose a solution $x^{(1)}$ is computed by some method, perhaps Gaussian elimination. There is a simple way to improve the accuracy of the result called *iterative improvement*. First compute

$$\xi = Ax^{(1)} - b. \tag{12}$$

Then solve the system

$$Ay = \xi. \tag{13}$$

Note that if $x^{(1)}$ is an exact solution, then ξ is the zero vector, as is y. In general, the computation of ξ and y will incur rounding error, so $Ay \approx \xi \approx Ax^{(1)} - b = A(x^{(1)} - x)$, where x is the (unknown) true solution. Then $y \approx x^{(1)} - x$, so an improved estimate for the solution is

$$x^{(2)} = x^{(1)} - y. \tag{14}$$

The three steps (12), (13), and (14) can be repeated, replacing $x^{(1)}$ with $x^{(2)}$, and $x^{(2)}$ with $x^{(3)}$. This argument that $x^{(i+1)}$ is more accurate than $x^{(i)}$ is only informal. For more information, see Golub and Van Loan [1989].

When performing iterative improvement, ξ is a vector whose elements are the difference of nearby inexact floating-point numbers and so can suffer from catastrophic cancellation. Thus, iterative improvement is not very useful unless $\xi = Ax^{(1)} - b$ is computed in double precision. Once again, this is a case of computing the product of two single-precision numbers (A and $x^{(1)}$), where the full double-precision result is needed.

To summarize, instructions that multiply two floating-point numbers and return a product with twice the precision of the operands make a useful addition to a floating-point instruction set. Some of the implications of this for compilers are discussed in the next section.

3.2 Languages and Compilers

The interaction of compilers and floating point is discussed in Farnum [1988], and much of the discussion in this section is taken from that paper.

3.2.1 Ambiguity

Ideally, a language definition should define the semantics of the language precisely enough to prove statements about programs. Whereas this is usually true for the integer part of a language, language definitions often have a large gray area when it comes to floating point (modula-3 is an exception [Nelson 1991]). Perhaps this is due to the fact that many language designers believe that nothing can be proven about floating point, since it entails rounding error. If so, the previous sections have demonstrated the fallacy in this reasoning. This section discusses some common gray areas in language definitions and gives suggestions about how to deal with them.

Remarkably enough, some languages do not clearly specify that if **x** is a floating-point variable (with say a value of **3.0/10.0**), then every occurrence of (say) **10.0** $*$ **x** must have the same value. For example Ada,[17] which is based on Brown's model, seems to imply that floating-point arithmetic only has to satisfy Brown's axioms, and thus expressions can have one of many possible values. Thinking about floating point in this fuzzy way stands in sharp contrast to the IEEE model, where the result of each floating-point operation is precisely defined. In the IEEE model, we can prove that **(3.0/10.0)** $*$ **3.0** evaluates to 3 (Theorem 7). In Brown's model, we cannot.

Another ambiguity in most language definitions concerns what happens on overflow, underflow, and other exceptions. The IEEE standard precisely specifies the behavior of exceptions, so languages that use the standard as a model can avoid any ambiguity on this point.

Another gray area concerns the interpretation of parentheses. Due to roundoff errors, the associative laws of algebra do not necessarily hold for floating-point numbers. For example, the expression **(x + y) + z** has a totally different answer than **x + (y + z)** when $x = 10^{30}$, $y = -10^{30}$, and $z = 1$ (it is 1 in the former case, 0 in the latter). The importance of preserving parentheses cannot be overemphasized. The algorithms presented in Theorems 3, 4, and 6 all depend on it. For example, in Theorem 6, the formula $x_h = mx - (mx - x)$ would reduce to $x_h = x$ if it were not for parentheses, thereby destroying the entire algorithm. A language definition that does not require parentheses to be honored is useless for floating-point calculations.

Subexpression evaluation is imprecisely defined in many languages. Suppose **ds** is double precision, but **x** and **y** are single precision. Then in the expression **ds** + **x** $*$ **y**, is the product performed in single or double precision? Here is another example: In **x** + **m**/**n** where **m** and **n** are integers, is the division an integer operation or a floating-point one? There are two ways to deal with this problem, neither of which is completely satisfactory. The first is to require that all variables in an expression have the same type. This is the simplest solution but has some drawbacks. First, languages like Pascal that have subrange types allow mixing subrange variables with integer variables, so it is somewhat bizarre to prohibit mixing single- and double-precision variables. Another problem concerns constants. In the expression **0.1** $*$ **x**, most languages interpret 0.1 to be a single-precision constant. Now suppose the programmer decides to change the declaration of all the floating-point variables from single to double precision. If 0.1 is still treated as a single-precision constant, there will be a compile time error. The programmer will have to hunt down and change every floating-point constant.

The second approach is to allow mixed expressions, in which case rules for subexpression evaluation must be provided. There are a number of guiding examples. The original definition of C required that every floating-point expres-

[17]Ada is a registered trademark of the U S. Government Ada joint program office

sion be computed in double precision [Kernighan and Ritchie 1978]. This leads to anomalies like the example immediately proceeding Section 3.1. The expression **3.0/7.0** is computed in double precision, but if **q** is a single-precision variable, the quotient is rounded to single precision for storage. Since 3/7 is a repeating binary fraction, its computed value in double precision is different from its stored value in single precision. Thus, the comparison $q = 3/7$ fails. This suggests that computing every expression in the highest precision available is not a good rule.

Another guiding example is inner products. If the inner product has thousands of terms, the rounding error in the sum can become substantial. One way to reduce this rounding error is to accumulate the sums in double precision (this will be discussed in more detail in Section 3.2.3). If **d** is a double-precision variable, and **x[ ]** and **y[ ]** are single precision arrays, the inner product loop will look like **d = d + x[i] * y[i]**. If the multiplication is done in single precision, much of the advantage of double-precision accumulation is lost because the product is truncated to single precision just before being added to a double-precision variable.

A rule that covers the previous two examples is to compute an expression in the highest precision of any variable that occurs in that expression. Then **q = 3.0/7.0** will be computed entirely in single precision[18] and will have the Boolean value true, whereas **d = d + x[i] * y[i]** will be computed in double precision, gaining the full advantage of double-precision accumulation. This rule is too simplistic, however, to cover all cases cleanly. If **dx** and **dy** are double-precision variables, the expression **y = x + single(dx − dy)** contains a double-precision variable, but performing the sum in double precision would be pointless because both operands are single precision, as is the result.

A more sophisticated subexpression evaluation rule is as follows. First, assign each operation a tentative precision, which is the maximum of the precision of its operands. This assignment has to be carried out from the leaves to the root of the expression tree. Then, perform a second pass from the root to the leaves. In this pass, assign to each operation the maximum of the tentative precision and the precision expected by the parent. In the case of **q = 3.0/7.0**, every leaf is single precision, so all the operations are done in single precision. In the case of **d = d + x[i] * y[i]**, the tentative precision of the multiply operation is single precision, but in the second pass it gets promoted to double precision because its parent operation expects a double-precision operand. And in **y = x + single (dx − dy)**, the addition is done in single precision. Farnum [1988] presents evidence that this algorithm is not difficult to implement.

The disadvantage of this rule is that the evaluation of a subexpression depends on the expression in which it is embedded. This can have some annoying consequences. For example, suppose you are debugging a program and want to know the value of a subexpression. You cannot simply type the subexpression to the debugger and ask it to be evaluated because the value of the subexpression in the program depends on the expression in which it is embedded. A final comment on subexpression is that since converting decimal constants to binary is an operation, the evaluation rule also affects the interpretation of decimal constants. This is especially important for constants like 0.1, which are not exactly representable in binary.

Another potential gray area occurs when a language includes exponentiation as one of its built-in operations. Unlike the basic arithmetic operations, the value of exponentiation is not always obvious [Kahan and Coonen 1982]. If ** is the exponentiation operator, then **(−3)**3** certainly has the value −27.

[18]This assumes the common convention that **3.0** is a single-precision constant, whereas **3.0D0** is a double-precision constant.

However, $(-\mathbf{3.0}) * * \mathbf{3.0}$ is problematical. If the $**$ operator checks for integer powers, it would compute $(-\mathbf{3.0}) * * \mathbf{3.0}$ as $-3.0^3 = -27$. On the other hand, if the formula $x^y = e^{y \log x}$ is used to define $**$ for real arguments, then depending on the log function, the result could be a NaN (using the natural definition of $\log(x) = \text{NaN}$ when $x < 0$). If the FORTRAN **CLOG** function is used, however, the answer will be -27 because the ANSI FORTRAN standard defines **CLOG** $(-\mathbf{3.0})$ to be $i\pi \log 3$ [ANSI 1978]. The programming language Ada avoids this problem by only defining exponentiation for integer powers, while ANSI FORTRAN prohibits raising a negative number to a real power.

In fact, the FORTRAN standard says that

> Any arithmetic operation whose result is not mathematically defined is prohibited . . .

Unfortunately, with the introduction of $\pm\infty$ by the IEEE standard, the meaning of *not mathematically defined* is no longer totally clear cut. One definition might be to use the method of Section 2.2.2. For example, to determine the value of a^b, consider nonconstant analytic functions f and g with the property that $f(x) \to a$ and $g(x) \to b$ as $x \to 0$. If $f(x)^{g(x)}$ always approaches the same limit, this should be the value of a^b. This definition would set $2^\infty = \infty$, which seems quite reasonable. In the case of 1.0^∞, when $f(x) = 1$ and $g(x) = 1/x$ the limit approaches 1, but when $f(x) = 1 - x$ and $g(x) = 1/x$ the limit is e. So 1.0^∞ should be an NaN. In the case of 0^0, $f(x)^{g(x)} = e^{g(x) \log f(x)}$. Since f and g are analytical and take on the value of 0 at 0, $f(x) = a_1 x^1 + a_2 x^2 + \cdots$ and $g(x) = b_1 x^1 + b_2 x^2 + \cdots$. Thus,

$$\begin{aligned} \lim_{x \to 0} g(x) \log f(x) & \\ &= \lim_{x \to 0} x \log(x(a_1 + a_2 x + \cdots)) \\ &= \lim_{x \to 0} x \log(a_1 x) = 0. \end{aligned}$$

So $f(x)^{g(x)} \to e^0 = 1$ for all f and g, which means $0^0 = 1$.[19] Using this definition would unambiguously define the exponential function for all arguments and in particular would define $(-\mathbf{3.0}) * * \mathbf{3.0}$ to be -27.

3.2.2 IEEE Standard

Section 2 discussed many of the features of the IEEE standard. The IEEE standard, however, says nothing about how these features are to be accessed from a programming language. Thus, there is usually a mismatch between floating-point hardware that supports the standard and programming languages like C, Pascal, or FORTRAN. Some of the IEEE capabilities can be accessed through a library of subroutine calls. For example, the IEEE standard requires that square root be exactly rounded, and the square root function is often implemented directly in hardware. This functionality is easily accessed via a library square root routine. Other aspects of the standard, however, are not so easily implemented as subroutines. For example, most computer languages specify at most two floating-point types, whereas, the IEEE standard has four different precisions (although the recommended configurations are single plus single extended or single, double, and double extended). Infinity provides another example. Constants to represent $\pm\infty$ could be supplied by a subroutine. But that might make them unusable in places that require constant expressions, such as the initializer of a constant variable.

A more subtle situation is manipulating the state associated with a computation, where the state consists of the rounding modes, trap enable bits, trap handlers, and exception flags. One approach is to provide subroutines for reading and writing the state. In addition, a

[19]The conclusion that $0^0 = 1$ depends on the restriction f be nonconstant. If this restriction is removed, then letting f be the identically 0 function gives 0 as a possible value for $\lim_{x \to 0} f(x)^{g(x)}$, and so 0^0 would have to be defined to be a NaN.

single call that can atomically set a new value and return the old value is often useful. As the examples in Section 2.3.3 showed, a common pattern of modifying IEEE state is to change it only within the scope of a block or subroutine. Thus, the burden is on the programmer to find each exit from the block and make sure the state is restored. Language support for setting the state precisely in the scope of a block would be very useful here. Modula-3 is one language that implements this idea for trap handlers [Nelson 1991].

A number of minor points need to be considered when implementing the IEEE standard in a language. Since $x - x = +0$ for all x,[20] $(+0) - (+0) = +0$. However, $-(+0) = -0$, thus $-x$ should not be defined as $0 - x$. The introduction of NaNs can be confusing because an NaN is never equal to any other number (including another NaN), so $x = x$ is no longer always true. In fact, the expression $x \neq x$ is the simplest way to test for a NaN if the IEEE recommended function **Isnan** is not provided. Furthermore, NaNs are unordered with respect to all other numbers, so $x \leq y$ cannot be defined as **not** $x > y$. Since the introduction of NaNs causes floating-point numbers to become partially ordered, a **compare** function that returns one of $<$, $=$, $>$, or *unordered* can make it easier for the programmer to deal with comparisons.

Although the IEEE standard defines the basic floating-point operations to return a NaN if any operand is a NaN, this might not always be the best definition for compound operations. For example, when computing the appropriate scale factor to use in plotting a graph, the maximum of a set of values must be computed. In this case, it makes sense for the max operation simply to ignore NaNs.

Finally, rounding can be a problem. The IEEE standard defines rounding precisely, and it depends on the current value of the rounding modes. This sometimes conflicts with the definition of implicit rounding in type conversions or the explicit **round** function in languages. This means that programs that wish to use IEEE rounding cannot use the natural language primitives, and conversely the language primitives will be inefficient to implement on the ever-increasing number of IEEE machines.

[20]Unless the rounding mode is round toward $-\infty$, in which case $x - x = -0$.

3.2.3 Optimizers

Compiler texts tend to ignore the subject of floating point. For example, Aho et al. [1986] mentions replacing **x/2.0** with **x * 0.5**, leading the reader to assume that **x/10.0** should be replaced by **0.1 * x**. These two expressions do not, however, have the same semantics on a binary machine because 0.1 cannot be represented exactly in binary. This textbook also suggests replacing **x * y − x * z** by **x * (y − z)**, even though we have seen that these two expressions can have quite different values when $y \approx z$. Although it does qualify the statement that any algebraic identity can be used when optimizing code by noting that optimizers should not violate the language definition, it leaves the impression that floating-point semantics are not very important. Whether or not the language standard specifies that parenthesis must be honored, **(x + y) + z** can have a totally different answer than **x + (y + z)**, as discussed above.

There is a problem closely related to preserving parentheses that is illustrated by the following code:

```
eps = 1
do eps = 0.5 * eps while (eps + 1 > 1)
```

This code is designed to give an estimate for machine epsilon. If an optimizing compiler notices that $eps + 1 > 1 \Leftrightarrow eps > 0$, the program will be changed completely. Instead of computing the smallest number x such that $1 \oplus x$ is still greater than $x (x \approx \epsilon \approx \beta^{-p})$, it will compute the largest number x for which $x/2$ is rounded to 0 ($x \approx \beta^{e_{\min}}$). Avoiding this kind of "optimization" is so important that it is worth presenting one more useful algorithm that is totally ruined by it.

Many problems, such as numerical integration and the numerical solution of differential equations, involve computing sums with many terms. Because each addition can potentially introduce an error as large as 1/2 ulp, a sum involving thousands of terms can have quite a bit of rounding error. A simple way to correct for this is to store the partial summand in a double-precision variable and to perform each addition using double precision. If the calculation is being done in single precision, performing the sum in double precision is easy on most computer systems. If the calculation is already being done in double precision, however, doubling the precision is not so simple. One method that is sometimes advocated is to sort the numbers and add them from smallest to largest. There is a much more efficient method, however, that dramatically improves the accuracy of sums, namely Theorem 8.

Theorem 8 (Kahan Summation Formula)

Suppose $\sum_{j=1}^{N} x_j$ *is computed using the following algorithm*

```
S = X[1]
C = 0
for j = 2 to N {
  Y = X[j] - C
  T = S + Y
  C = (T - S) - Y
  S = T
}
```

Then the computed sum S is equal to $\sum x_j(1 + \delta_j) + O(N\epsilon^2)\sum |x_j|$, *where* $|\delta_j| \le 2\epsilon$.

Using the naive formula $\sum x_j$, the computed sum is equal to $\sum x_j(1 + \delta_j)$ where $|\delta_j| < (n - j)\epsilon$. Comparing this with the error in the Kahan summation formula shows a dramatic improvement. Each summand is perturbed by only 2ϵ instead of perturbations as large as $n\epsilon$ in the simple formula. Details are in Section 4.3.

An optimizer that believed floating-point arithmetic obeyed the laws of algebra would conclude that $C = [T - S] - Y = [(S + Y) - S] - Y = 0$, rendering the algorithm completely useless. These examples can be summarized by saying that optimizers should be extremely cautious when applying algebraic identities that hold for the mathematical real numbers to expressions involving floating-point variables.

Another way that optimizers can change the semantics of floating-point code involves constants. In the expression **1.0E-40** $*$ **x**, there is an implicit decimal to binary conversion operation that converts the decimal number to a binary constant. Because this constant cannot be represented exactly in binary, the inexact exception should be raised. In addition, the underflow flag should to be set if the expression is evaluated in single precision. Since the constant is inexact, its exact conversion to binary depends on the current value of the IEEE rounding modes. Thus, an optimizer that converts **1.0E-40** to binary at compile time would be changing the semantics of the program. Constants like 27.5, however, that are exactly representable in the smallest available precision can be safely converted at compile time, since they are always exact, cannot raise any exception, and are unaffected by the rounding modes. Constants that are intended to be converted at compile time should be done with a constant declaration such as **const pi = 3.14159265.**

Common subexpression elimination is another example of an optimization that can change floating-point semantics, as illustrated by the following code:

```
C = A * B;
RndMode = Up
D = A * B;
```

Although **A** $*$ **B** may appear to be a common subexpression, it is not because the rounding mode is different at the two evaluation sites. Three final examples are $x = x$ cannot be replaced by the Boolean constant **true**, because it fails when x is an NaN; $-x = 0 - x$ fails for $x = +0$; and $x < y$ is not the opposite of $x \geq y$, because NaNs are neither greater than nor less than ordinary floating-point numbers.

Despite these examples, there are useful optimizations that can be done on floating-point code. First, there are algebraic identities that are valid for floating-point numbers. Some examples in IEEE arithmetic are $x + y = y + x$, $2 \times x = x + x$, $1 \times x = x$, and $0.5 \times x = x/2$. Even these simple identities, however, can fail on a few machines such as CDC and Cray supercomputers. Instruction scheduling and inline procedure substitution are two other potentially useful optimizations.[21] As a final example, consider the expression **dx** = **x** $*$ **y**, where **x** and **y** are single precision variables and **dx** is double precision. On machines that have an instruction that multiplies two single-precision numbers to produce a double-precision number, **dx** = **x** $*$ **y** can get mapped to that instruction rather than compiled to a series of instructions that convert the operands to double then perform a double-to-double precision multiply.

Some compiler writers view restrictions that prohibit converting $(x + y) + z$ to $x + (y + z)$ as irrelevant, of interest only to programmers who use unportable tricks. Perhaps they have in mind that floating-point numbers model real numbers and should obey the same laws real numbers do. The problem with real number semantics is that they are extremely expensive to implement. Every time two n bit numbers are multiplied, the product will have $2n$ bits. Every time two n bit numbers with widely spaced exponents are added, the sum will have $2n$ bits. An algorithm that involves thousands of operations (such as solving a linear system) will soon be operating on huge numbers and be hopelessly slow.

[21]The VMS math libraries on the VAX use a weak form of inline procedure substitution in that they use the inexpensive jump to subroutine call rather than the slower **CALLS** and **CALLG** instructions.

The implementation of library functions such as sin and cos is even more difficult, because the value of these transcendental functions are not rational numbers. Exact integer arithmetic is often provided by Lisp systems and is handy for some problems. Exact floating-point arithmetic is, however, rarely useful.

The fact is there are useful algorithms (like the Kahan summation formula) that exploit $(x + y) + z \neq x + (y + z)$, and work whenever the bound

$$a \oplus b = (a + b)(1 + \delta)$$

holds (as well as similar bounds for $-$, $\times$, and $/$). Since these bounds hold for almost all commercial hardware not just machines with IEEE arithmetic, it would be foolish for numerical programmers to ignore such algorithms, and it would be irresponsible for compiler writers to destroy these algorithms by pretending that floating-point variables have real number semantics.

3.3 Exception Handling

The topics discussed up to now have primarily concerned systems implications of accuracy and precision. Trap handlers also raise some interesting systems issues. The IEEE standard strongly recommends that users be able to specify a trap handler for each of the five classes of exceptions, and Section 2.3.1 gave some applications of user defined trap handlers. In the case of invalid operation and division by zero exceptions, the handler should be provided with the operands, otherwise with the exactly rounded result. Depending on the programming language being used, the trap handler might be able to access other variables in the program as well. For all exceptions, the trap handler must be able to identify what operation was being performed and the precision of its destination.

The IEEE standard assumes that operations are conceptually serial and that when an interrupt occurs, it is possible to identify the operation and its operands.

On machines that have pipelining or multiple arithmetic units, when an exception occurs, it may not be enough simply to have the trap handler examine the program counter. Hardware support for identifying exactly which operation trapped may be necessary.

Another problem is illustrated by the following program fragment:

```
x = y*z
z = x*w
a = b+c
d = a/x
```

Suppose the second multiply raises an exception, and the trap handler wants to use the value of **a**. On hardware that can do an add and multiply in parallel, an optimizer would probably move the addition operation ahead of the second multiply, so that the add can proceed in parallel with the first multiply. Thus, when the second multiply traps, **a** = **b** + **c** has already been executed, potentially changing the result of **a**. It would not be reasonable for a compiler to avoid this kind of optimization because every floating-point operation can potentially trap, and thus virtually all instruction scheduling optimizations would be eliminated. This problem can be avoided by prohibiting trap handlers from accessing any variables of the program directly. Instead, the handler can be given the operands or result as an argument.

But there are still problems. In the fragment

```
x = y*z
z = a+b
```

the two instructions might well be executed in parallel. If the multiply traps, its argument **z** could already have been overwritten by the addition, especially since addition is usually faster than multiply. Computer systems that support trap handlers in the IEEE standard must provide some way to save the value of **z**, either in hardware or by having the compiler avoid such a situation in the first place.

Kahan has proposed using *presubstitution* instead of trap handlers to avoid these problems. In this method, the user specifies an exception and a value to be used as the result when the exception occurs. As an example, suppose that in code for computing sin x/x, the user decides that $x = 0$ is so rare that it would improve performance to avoid a test for $x = 0$ and instead handle this case when a $0/0$ trap occurs. Using IEEE trap handlers, the user would write a handler that returns a value of 1 and installs it before computing sin x/x. Using presubstitution, the user would specify that when an invalid operation occurs, the value of 1 should be used. Kahan calls this presubstitution because the value to be used must be specified before the exception occurs. When using trap handlers, the value to be returned can be computed when the trap occurs.

The advantage of presubstitution is that it has a straightforward hardware implementation. As soon as the type of exception has been determined, it can be used to index a table that contains the desired result of the operation. Although presubstitution has some attractive attributes, the widespread acceptance of the IEEE standard makes it unlikely to be widely implemented by hardware manufacturers.

4. DETAILS

Various claims have been made in this paper concerning properties of floating-point arithmetic. We now proceed to show that floating point is not black magic, but rather a straightforward subject whose claims can be verified mathematically.

This section is divided into three parts. The first part represents an introduction to error analysis and provides the details for Section 1. The second part explores binary-to-decimal conversion, filling in some gaps from Section 2. The third part discusses the Kahan summation formula, which was used as an example in Section 3.

4.1 Rounding Error

In the discussion of rounding error, it was stated that a single guard digit is enough to guarantee that addition and subtraction will always be accurate (Theorem 2). We now proceed to verify this fact. Theorem 2 has two parts, one for subtraction and one for addition. The part for subtraction is as follows:

Theorem 9

If x and y are positive floating-point numbers in a format with parameters β and p and if subtraction is done with $p + 1$ digits (i.e., one guard digit), then the relative rounding error in the result is less than $[(\beta/2) + 1]\beta^{-p} = [1 + (2/\beta)]\epsilon \le 2\epsilon$.

Proof. Interchange x and y is necessary so that $x > y$. It is also harmless to scale x and y so that x is represented by $x_0.x_1 \cdots x_{p-1} \times \beta^0$. If y is represented as $y_0.y_1 \cdots y_{p-1}$, then the difference is exact. If y is represented as $0.y_1 \cdots y_p$, then the guard digit ensures that the computed difference will be the exact difference rounded to a floating-point number, so the rounding error is at most ϵ. In general, let $y = 0.0 \cdots 0 y_{k+1} \cdots y_{k+p}$ and let $\bar{y}$ be y truncated to $p + 1$ digits. Then,

$$y - \bar{y} < (\beta - 1)(\beta^{-p-1} \cdots + \beta^{-p-k}). \tag{15}$$

From the definition of guard digit, the computed value of $x - y$ is $x - \bar{y}$ rounded to be a floating-point number; that is, $(x - \bar{y}) + \delta$, where the rounding error δ satisfies

$$|\delta| \le \left(\frac{\beta}{2}\right)\beta^{-p}. \tag{16}$$

The exact difference is $x - y$, so the error is $(x - y) - (x - \bar{y} + \delta) = \bar{y} - y + \delta$. There are three cases. If $x - y \ge 1$, the relative error is bounded by

$$\left|\frac{y - \bar{y} + \delta}{1}\right| \le \beta^{-p}\left[(\beta - 1)(\beta^{-1} + \cdots + \beta^{-k}) + \frac{\beta}{2}\right] < \beta^{-p}\left(1 + \frac{\beta}{2}\right). \tag{17}$$

Second, if $x - \bar{y} < 1$, then $\delta = 0$. Since the smallest that $x - y$ can be is

$$1.0 - 0.\overbrace{0 \cdots 0}^{k}\overbrace{\varrho \cdots \varrho}^{k} > (\beta - 1)(\beta^{-1} + \cdots + \beta^{-k})$$

(where $\varrho = \beta - 1$), in this case the relative error is bounded by

$$\frac{y - \bar{y} + \delta}{(\beta - 1)(\beta^{-1} + \cdots + \beta^{-k})} < \frac{(\beta - 1)\beta^{-p}(\beta^{-1} + \cdots + \beta^{-k})}{(\beta - 1)(\beta^{-1} + \cdots + \beta^{-k})} = \beta^{-p}. \tag{18}$$

The final case is when $x - y < 1$ but $x - \bar{y} \ge 1$. The only way this could happen is if $x - \bar{y} = 1$, in which case $\delta = 0$. But if $\delta = 0$, then (18) applies, so again the relative error is bounded by $\beta^{-p} < \beta^{-p}(1 + \beta/2)$. ■

When $\beta = 2$, the bound is exactly 2ϵ, and this bound is achieved for $x = 1 + 2^{2-p}$ and $y = 2^{1-p} - 2^{1-2p}$ in the limit as $p \to \infty$. When adding numbers of the same sign, a guard digit is not necessary to achieve good accuracy, as the following result shows.

Theorem 10

If $x \ge 0$ and $y \ge 0$, the relative error in computing $x + y$ is at most 2ϵ, even if no guard digits are used.

Proof. The algorithm for addition with k guard digits is similar to the algorithm for subtraction. If $x \ge y$, and shift y right until the radix points of x and y are aligned. Discard any digits shifted past the $p + k$ position. Compute the sum of these two $p + k$ digit numbers exactly. Then round to p digits.

We will verify the theorem when no guard digits are used; the general case is similar. There is no loss of generality in assuming that $x \ge y \ge 0$ and that x is scaled to be of the form $d.d \cdots d \times \beta^0$. First, assume there is no carry out. Then the digits shifted off the end of y have a value less than β^{-p+1} and the sum is at least 1, so the relative error is less than $\beta^{-p+1}/1 = 2\epsilon$. If there is a carry out, the error from shifting must be added to the rounding error of $(1/2)\beta^{-p+2}$. The sum is at least β, so the relative error is less than $(\beta^{-p+1} + (1/2)\beta^{-p+2})/\beta = (1 + \beta/2)\beta^{-p} \le 2\epsilon$. ■

It is obvious that combining these two theorems gives Theorem 2. Theorem 2 gives the relative error for performing one operation. Comparing the rounding error of $x^2 - y^2$ and $(x + y)(x - y)$ requires knowing the relative error of multiple operations. The relative error of $x \ominus y$ is $\delta_1 = [(x \ominus y) - (x - y)]/(x - y)$, which satisfies $|\delta_1| \le 2\epsilon$. Or to write it another way,

$$x \ominus y = (x - y)(1 + \delta_1), \qquad |\delta_1| \le 2\epsilon. \tag{19}$$

Similarly,

$$x \oplus y = (x + y)(1 + \delta_2), \qquad |\delta_2| \le 2\epsilon. \tag{20}$$

Assuming that multiplication is performed by computing the exact product then rounding, the relative error is at most 1/2 ulp, so

$$u \otimes v = uv(1 + \delta_3), \qquad |\delta_3| \le \epsilon \tag{21}$$

for any floating point numbers u and v. Putting these three equations together (letting $u = x \ominus y$ and $v = x \oplus y$) gives

$$\begin{aligned}(x \ominus y) \otimes (x \oplus y) &= (x - y)(1 + \delta_1) \\ &\quad \times (x + y)(1 + \delta_2)(1 + \delta_3).\end{aligned} \tag{22}$$

So the relative error incurred when computing $(x - y)(x + y)$ is

$$\begin{aligned}&\frac{(x \ominus y) \otimes (x \oplus y) - (x^2 - y^2)}{(x^2 - y^2)} \\ &\quad = (1 + \delta_1)(1 + \delta_2)(1 + \delta_3) - 1.\end{aligned} \tag{23}$$

This relative error is equal to $\delta_1 + \delta_2 + \delta_3 + \delta_1\delta_2 + \delta_1\delta_3 + \delta_2\delta_3$, which is bounded by $5\epsilon + 8\epsilon^2$. In other words, the maximum relative error is about five rounding errors (since ϵ is a small number, ϵ^2 is almost negligible).

A similar analysis of $(x \otimes x) \ominus (y \otimes y)$ cannot result in a small value for the relative error because when two nearby values of x and y are plugged into $x^2 - y^2$, the relative error will usually be quite large. Another way to see this is to try and duplicate the analysis that worked on $(x \ominus y) \otimes (x \oplus y)$, yielding

$$\begin{aligned}(x \otimes x) \ominus (y \otimes y) &= [x^2(1 + \delta_1) - y^2(1 + \delta_2)](1 + \delta_3) \\ &= ((x^2 - y^2)(1 + \delta_1) + (\delta_1 - \delta_2)y^2) \\ &\quad (1 + \delta_3).\end{aligned}$$

When x and y are nearby, the error term $(\delta_1 - \delta_2)y^2$ can be as large as the result $x^2 - y^2$. These computations formally justify our claim that $(x - y)(x + y)$ is more accurate than $x^2 - y^2$.

We next turn to an analysis of the formula for the area of a triangle. To estimate the maximum error that can occur when computing with (7), the following fact will be needed.

Theorem 11

If subtraction is performed with a guard digit and $y/2 \le x \le 2y$, then $x - y$ is computed exactly.

Proof. Note that if x and y have the same exponent, then certainly $x \ominus y$ is exact. Otherwise, from the condition of the theorem, the exponents can differ by at most 1. Scale and interchange x and y if necessary so $0 \le y \le x$ and x is represented as $x_0.x_1 \cdots x_{p-1}$ and y as $0.y_1 \cdots y_p$. Then the algorithm for computing $x \ominus y$ will compute $x - y$ exactly and round to a floating-point number but if the difference is of the form $0.d_1 \cdots d_p$, the difference will already be p digits long, and no rounding is necessary. Since

$x \le 2y$, $x - y \le y$, and since y is of the form $0.d_1 \cdots d_p$, so is $x - y$. ■

When $\beta > 2$, the hypothesis of Theorem 11 cannot be replaced by $y/\beta \le x \le \beta y$; the stronger condition $y/2 \le x \le 2y$ is still necessary. The analysis of the error in $(x - y)(x + y)$ in the previous section used the fact that the relative error in the basic operations of addition and subtraction is small [namely, eqs. (19) and (20)]. This is the most common kind of error analysis. Analyzing formula (7), however, requires something more; namely, Theorem 11, as the following proof will show.

Theorem 12

If subtraction uses a guard digit and if a, b, and c are the sides of a triangle, the relative error in computing $(a + (b + c))(c - (a - b))(c + (a - b))(a + (b - c))$ is at most 16ϵ, provided $\epsilon < .005$.

Proof. Let us examine the factors one by one. From Theorem 10, $b \oplus c = (b + c)(1 + \delta_1)$, where δ_1 is the relative error and $|\delta_1| \le 2\epsilon$. Then the value of the first factor is $(a \oplus (b \oplus c)) = (a + (b \oplus c))(1 + \delta_2) = (a + (b + c)(1 + \delta_1)) \times (1 + \delta_2)$, and thus

$$\begin{aligned}(a + b + c)(1 - 2\epsilon)^2 &\le [a + (b + c)(1 - 2\epsilon)](1 - 2\epsilon)\\ &\le a \oplus (b \oplus c)\\ &\le [a + (b + c)(1 + 2\epsilon)](1 + 2\epsilon)\\ &\le (a + b + c)(1 + 2\epsilon)^2.\end{aligned}$$

This means that there is an η_1 so that

$$(a \oplus (b \oplus c)) = (a + b + c)(1 + \eta_1)^2, \quad |\eta_1| \le 2\epsilon. \quad (24)$$

The next term involves the potentially catastrophic subtraction of c and $a \ominus b$, because $a \ominus b$ may have rounding error. Because a, b, and c are the sides of a triangle, $a \le b + c$, and combining this with the ordering $c \le b \le a$ gives $a \le b + c \le 2b \le 2a$. So $a - b$ satisfies the conditions of Theorem 11. This means $a - b = a \ominus b$ is exact, and hence $c \ominus (a - b)$ is a harmless subtraction that can be estimated from Theorem 9 to be

$$(c \ominus (a \ominus b)) = (c - (a - b))(1 + \eta_2), \quad |\eta_2| \le 2\epsilon. \quad (25)$$

The third term is the sum of two exact positive quantities, so

$$(c \oplus (a \ominus b)) = (c + (a - b))(1 + \eta_3), \quad |\eta_3| \le 2\epsilon. \quad (26)$$

Finally, the last term is

$$(a \oplus (b \ominus c)) = (a + (b - c))(1 + \eta_4)^2, \quad |\eta_4| \le 2\epsilon, \quad (27)$$

using both Theorem 9 and Theorem 10. If multiplication is assumed to be exactly rounded so that $x \otimes y = xy(1 + \zeta)$ with $|\zeta| \le \epsilon$, then combining (24), (25), (26), and (27) gives

$$\begin{aligned}&(a \oplus (b \oplus c))(c \ominus (a \ominus b))\\ &\quad (c \oplus (a \ominus b))(a \oplus (b \ominus c))\\ &\le (a + (b + c))(c - (a - b))\\ &\quad (c + (a - b))\\ &\quad (a + (b - c))E,\end{aligned}$$

where

$$\begin{aligned}E = {}&(1 + \eta_1)^2(1 + \eta_2)(1 + \eta_3)(1 + \eta_4)^2\\ &(1 + \zeta_1)(1 + \zeta_2)(1 + \zeta_3).\end{aligned}$$

An upper bound for E is $(1 + 2\epsilon)^6(1 + \epsilon)^3$, which expands to $1 + 15\epsilon + O(\epsilon^2)$. Some writers simply ignore the $O(\epsilon^2)$ term, but it is easy to account for it. Writing $(1 + 2\epsilon)^6(1 + \epsilon)^3 = 1 + 15\epsilon + \epsilon R(\epsilon)$, $R(\epsilon)$ is a polynomial in ϵ with positive coefficients, so it is an increasing function of ϵ. Since $R(.005) = .505$, $R(\epsilon) < 1$ for all $\epsilon < .005$, and hence $E \le (1 + 2\epsilon)^6(1 + \epsilon)^3 < 1 + 16\epsilon$. To get a lower bound on E, note that $1 - 15\epsilon - \epsilon R(\epsilon) < E$; so when $\epsilon < .005$, $1 - 16\epsilon < (1 -$

$2\epsilon)^6(1-\epsilon)^3$. Combining these two bounds yields $1 - 16\epsilon < E < 1 + 16\epsilon$. Thus the relative error is at most 16ϵ. ■

Theorem 12 shows there is no catastrophic cancellation in formula (7). Therefore, although it is not necessary to show formula (7) is numerically stable, it is satisfying to have a bound for the entire formula, which is what Theorem 3 of Section 1.4 gives.

Proof Theorem 3. Let

$$q = \big(a + (b + c)\big)\big(c - (a - b)\big)\big(c + (a - b)\big)\big(a + (b - c)\big)$$

and

$$Q = \big(a \oplus (b \oplus c)\big) \otimes \big(c \ominus (a \ominus b)\big) \otimes \big(c \oplus (a \ominus b)\big) \otimes \big(a \oplus (b \ominus c)\big).$$

Then Theorem 12 shows that $Q = q(1 + \delta)$, with $\delta \le 16\epsilon$. It is easy to check that

$$1 - .52|\delta| \le \sqrt{1 - |\delta|} \le \sqrt{1 + |\delta|} \le 1 + .52|\delta| \qquad (28)$$

provided $\delta \le .04/(.52)^2 \approx .15$. Since $|\delta| \le 16\epsilon \le 16(.005) = .08$, δ does satisfy the condition. Thus, $\sqrt{Q} = [q(1 + \delta)]^{1/2} = \sqrt{q}(1 + \delta_1)$, with $|\delta_1| \le .52|\delta| \le 8.5\epsilon$. If square roots are computed to within 1/2 ulp, the error when computing $\sqrt{Q}$ is $(1 + \delta_1)(1 + \delta_2)$, with $|\delta_2| \le \epsilon$. If $\beta = 2$, there is no further error committed when dividing by 4. Otherwise, one more factor $1 + \delta_3$ with $|\delta_3| \le \epsilon$ is necessary for the division, and using the method in the proof of Theorem 12, the final error bound of $(1 + \delta_1)(1 + \delta_2)(1 + \delta_3)$ is dominated by $1 + \delta_4$, with $|\delta_4| \le 11\epsilon$. ■

To make the heuristic explanation immediately following the statement of Theorem 4 precise, the next theorem describes just how closely $\mu(x)$ approximates a constant.

Theorem 13

If $\mu(x) = \ln(1 + x)/x$, then for $0 \le x \le 3/4$, $1/2 \le \mu(x) \le 1$ and the derivative satisfies $|\mu'(x)| \le 1/2$.

Proof. Note that $\mu(x) = 1 - x/2 + x^2/3 - \cdots$ is an alternating series with decreasing terms, so for $x \le 1$, $\mu(x) \ge 1 - x/2 \ge 1/2$. It is even easier to see that because the series for μ is alternating, $\mu(x) \le 1$. The Taylor series of $\mu'(x)$ is also alternating, and if $x \le 3/4$ has decreasing terms, so $-1/2 \le \mu'(x) \le -1/2 + 2x/3$, or $-1/2 \le \mu'(x) \le 0$, thus $|\mu'(x)| \le 1/2$. ■

Proof Theorem 4. Since the Taylor series for ln,

$$\ln(1 + x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \cdots,$$

is an alternating series, $0 < x - \ln(1 + x) < x^2/2$. Therefore, the relative error incurred when approximating $\ln(1 + x)$ by x is bounded by $x/2$. If $1 \oplus x = 1$, then $|x| < \epsilon$, so the relative error is bounded by $\epsilon/2$.

When $1 \oplus x \ne 1$, define $\hat{x}$ via $1 \oplus x = 1 + \hat{x}$. Then since $0 \le x < 1$, $(1 \oplus x) \ominus 1 = \hat{x}$. If division and logarithms are computed to within 1/2 ulp, the computed value of the expression $\ln(1 + x)/((1 + x) - 1)$ is

$$\frac{\ln(1 \oplus x)}{(1 \oplus x) \ominus 1}(1 + \delta_1)(1 + \delta_2) = \frac{\ln(1 + \hat{x})}{\hat{x}}(1 + \delta_1)(1 + \delta_2) = \mu(\hat{x})(1 + \delta_1)(1 + \delta_2), \qquad (29)$$

where $|\delta_1| \le \epsilon$ and $|\delta_2| \le \epsilon$. To estimate $\mu(\hat{x})$, use the mean value theorem, which says that

$$\mu(\hat{x}) - \mu(x) = (\hat{x} - x)\mu'(\xi) \qquad (30)$$

for some ξ between x and $\hat{x}$. From the definition of $\hat{x}$, it follows that $|\hat{x} - x| \le \epsilon$. Combining this with Theorem 13 gives $|\mu(\hat{x}) - \mu(x)| \le \epsilon/2$ or $|\mu(\hat{x})/\mu(x) - 1| \le \epsilon/(2|\mu(x)|) \le \epsilon$, which means $\mu(\hat{x}) = \mu(x)(1 + \delta_3)$, with $|\delta_3| \le \epsilon$. Finally, multiplying by x introduces a final δ_4, so the computed value of $x\ln(1+x)/((1+x)-1)$ is

$$\frac{x\ln(1+x)}{((1+x)-1)}(1+\delta_1)(1+\delta_2) \times (1+\delta_3)(1+\delta_4), \qquad |\delta_i| \le \epsilon.$$

It is easy to check that if $\epsilon < 0.1$, then $(1+\delta_1)(1+\delta_2)(1+\delta_3)(1+\delta_4) = 1+\delta$, with $|\delta| \le 5\epsilon$. ■

An interesting example of error analysis using formulas (19), (20), and (21) occurs in the quadratic formula $[-b \pm \sqrt{b^2-4ac}]/2a$. Section 1.4 explained how rewriting the equation will eliminate the potential cancellation caused by the $\pm$ operation. But there is another potential cancellation that can occur when computing $d = b^2 - 4ac$. This one cannot be eliminated by a simple rearrangement of the formula. Roughly speaking, when $b^2 \approx 4ac$, rounding error can contaminate up to half the digits in the roots computed with the quadratic formula. Here is an informal proof (another approach to estimating the error in the quadratic formula appears in Kahan [1972]).

If $b^2 \approx 4ac$, rounding error can contaminate up to half the digits in the roots computed with the quadratic formula $[-b \pm \sqrt{b^2-4ac}]/2a$.

Proof. Write $(b \otimes b) \oslash (4a \otimes c) = (b^2(1+\delta_1) - 4ac(1+\delta_2))(1+\delta_3)$, where $|\delta_i| \le 2\epsilon$.[22] Using $d = b^2 - 4ac$, this can be rewritten as $(d(1+\delta_1) - 4ac(\delta_1 - \delta_2))(1+\delta_3)$. To get an estimate for the size of this error, ignore second-order terms in δi, in which the case of the absolute error is $d(\delta_1+\delta_3) - 4ac\delta_4$, where $|\delta_4| = |\delta_1 - \delta_2| \le 2\epsilon$. Since $d \ll 4ac$, the first term $d(\delta_1+\delta_3)$ can be ignored. To estimate the second term, use the fact that $ax^2+bx+c = a(x-r_1)(x-r_2)$, so $ar_1r_2 = c$. Since $b^2 \approx 4ac$, then $r_1 \approx r_2$, so the second error term is $4ac\delta_4 \approx 4a^2r_1^2\delta_4$. Thus, the computed value of $\sqrt{d}$ is $\sqrt{d + 4a^2r_1^2\delta_4}$. The inequality

$$p - q \le \sqrt{p^2-q^2} \le \sqrt{p^2+q^2} \le p + q$$

shows that $\sqrt{d+4a^2r_1^2\delta_4} = \sqrt{d} + E$, where $|E| \le \sqrt{4a^2r_1^2|\delta_4|}$, so the absolute error in $\sqrt{d}/2a$ is about $r_1\sqrt{\delta_4}$. Since $\delta_4 \approx \beta^{-p}$, $\sqrt{\delta_4} \approx \beta^{-p/2}$, and thus the absolute error of $r_1\sqrt{\delta_4}$ destroys the bottom half of the bits of the roots $r_1 \approx r_2$. In other words, since the calculation of the roots involves computing with $\sqrt{d}/2a$ and this expression does not have meaningful bits in the position corresponding to the lower order half of r_i, the lower order bits of r_i cannot be meaningful. ■

Finally, we turn to the proof of Theorem 6. It is based on the following fact in Theorem 14, which is proven in the Appendix.

Theorem 14

Let $0 < k < p$, and set $m = \beta^k + 1$, and assume floating-point operations are exactly rounded. Then $(m \otimes x) \ominus (m \otimes x \ominus x)$ is exactly equal to x rounded to $p - k$ significant digits. More precisely, x is rounded by taking the significand of x, imagining a radix point just left of the k least significant digits and rounding to an integer.

Proof Theorem 6. By Theorem 14, x_h is x rounded to $p - k = \lfloor p/2 \rfloor$ places. If there is no carry out, x_h can be represented with $\lfloor p/2 \rfloor$ significant digits. Suppose there is a carry out. If $x = x_0\ x_1\ x_{p-1} \times \beta^e$, rounding adds 1 to x_{p-k-1}, the only way there can be a carry out is if $x_{p-k-1} = \beta - 1$. In that case, however, the low-order digit of x_h is $1 + x_{p-k-1} = 0$, and so again x_h is representable in $\lfloor p/2 \rfloor$ digits.

[22]In this informal proof, assume $\beta = 2$ so multiplication by 4 is exact and does not require a δ_i.

To deal with x_l, scale x to be an integer satisfying $\beta^{p-1} \le x \le \beta^p - 1$. Let $x = \bar{x}_h + \bar{x}_l$, where $\bar{x}_h$ is the $p - k$ high-order digits of x and $\bar{x}_l$ is the k low-order digits. There are three cases to consider. If $\bar{x}_l < (\beta/2)\beta^{k-1}$, then rounding x to $p - k$ places is the same as chopping and $x_h = \bar{x}_h$, and $x_l = \bar{x}_l$. Since $\bar{x}_l$ has at most k digits, if p is even, then $\bar{x}_l$ has at most $k = \lceil p/2 \rceil = \lceil p/2 \rceil$ digits. Otherwise, $\beta = 2$ and $\bar{x}_l < 2^{k-1}$ is representable with $k - 1 \le \lceil p/2 \rceil$ significant bits. The second case is when $\bar{x}_l > (\beta/2)\beta^{k-1}$; then computing x_h involves rounding up, so $x_h = \bar{x}_h + \beta^k$ and $x_l = x - x_h = x - \bar{x}_h - \beta^k = \bar{x}_l - \beta^k$. Once again, $\bar{x}_l$ has at most k digits, so it is representable with $\lceil p/2 \rceil$ digits. Finally, if $\bar{x}_l = (\beta/2)\beta^{k-1}$, then $x_h = \bar{x}_h$ or $\bar{x}_h + \beta^k$ depending on whether there is a round up. Therefore, x_l is either $(\beta/2)\beta^{k-1}$ or $(\beta/2)\beta^{k-1} - \beta^k = -\beta^k/2$, both of which are represented with 1 digit. ■

Theorem 6 gives a way to express the product of two single-precision numbers exactly as a sum. There is a companion formula for expressing a sum exactly. If $|x| \ge |y|$, then $x + y = (x \oplus y) + (x \ominus (x \oplus y)) \oplus y$ [Dekker 1971; Knuth 1981, Theorem C in Section 4.2.2]. When using exactly rounded operations, however, this formula is only true for $\beta = 2$, not for $\beta = 10$ as the example $x = .99998$, $y = .99997$ shows.

4.2 Binary-to-Decimal Conversion

Since single precision has $p = 24$ and $2^{24} < 10^8$, we might expect that converting a binary number to eight decimal digits would be sufficient to recover the original binary number. This is not the case, however.

Theorem 15

When a binary IEEE single-precision number is converted to the closest eight digit decimal number, it is not always possible to recover the binary number uniquely from the decimal one. If nine decimal digits are used, however, then converting the decimal number to the closest binary number will recover the original floating-point number.

Proof. Binary single-precision numbers lying in the half-open interval $[10^3, 2^{10}) = [1000, 1024)$ have 10 bits to the left of the binary point and 14 bits to the right of the binary point. Thus, there are $(2^{10} - 10^3)2^{14} = 393{,}216$ different binary numbers in that interval. If decimal numbers are represented with eight digits, there are $(2^{10} - 10^3)10^4 = 240{,}000$ decimal numbers in the same interval. There is no way 240,000 decimal numbers could represent 393,216 different binary numbers. So eight decimal digits are not enough to represent each single-precision binary number uniquely.

To show that nine digits are sufficient, it is enough to show that the spacing between binary numbers is always greater than the spacing between decimal numbers. This will ensure that for each decimal number N, the interval $[N - 1/2 \text{ ulp}, N + 1/2 \text{ ulp}]$ contains at most one binary number. Thus, each binary number rounds to a unique decimal number, which in turn rounds to a unique binary number.

To show that the spacing between binary numbers is always greater than the spacing between decimal numbers, consider an interval $[10^n, 10^{n+1}]$. On this interval, the spacing between consecutive decimal numbers is $10^{(n+1)-9}$. On $[10^n, 2^m]$, where m is the smallest integer so that $10^n < 2^m$, the spacing of binary numbers is 2^{m-24} and the spacing gets larger further on in the interval. Thus, it is enough to check that $10^{(n+1)-9} < 2^{m-24}$. But, in fact, since $10^n < 2^m$, then $10^{(n+1)-9} = 10^n 10^{-8} < 2^m 10^{-8} < 2^m 2^{-24}$. ■

The same argument applied to double precision shows that 17 decimal digits are required to recover a double-precision number.

Binary-decimal conversion also provides another example of the use of flags. Recall from Section 2.1.2 that to recover a binary number from its decimal expansion, the decimal-to-binary conversion must be computed exactly. That conversion is performed by multiplying the quantities N and $10^{|P|}$ (which are both exact if $P < 13$) in single-extended precision and then rounding this to single precision (or dividing if $P < 0$; both cases are similar). The computation of $N \cdot 10^{|P|}$ cannot be exact; it is the combined operation round $(N \cdot 10^{|P|})$ that must be exact, where the rounding is from single extended to single precision. To see why it might fail to be exact, take the simple case of $\beta = 10$, $p = 2$ for single and $p = 3$ for single extended. If the product is to be 12.51, this would be rounded to 12.5 as part of the single-extended multiply operation. Rounding to single precision would give 12. But that answer is not correct, because rounding the product to single precision should give 13. The error is a result of double rounding.

By using the IEEE flags, the double rounding can be avoided as follows. Save the current value of the inexact flag, then reset it. Set the rounding mode to round to zero. Then perform the multiplication $N \cdot 10^{|P|}$. Store the new value of the inexact flag in **ixflag**, and restore the rounding mode and inexact flag. If **ixflag** is 0, then $N \cdot 10^{|P|}$ is exact, so **round** $(N \cdot 10^{|P|})$ will be correct down to the last bit. If **ixflag** is 1, then some digits were truncated, since round to zero always truncates. The significand of the product will look like $1.b_1 \cdots b_{22} b_{23} \cdots b_{31}$. A double-rounding error may occur if $b_{23} \cdots b_{31} = 10 \cdots 0$. A simple way to account for both cases is to perform a logical OR of **ixflag** with b_{31}. Then **round** $(N \cdot 10^{|P|})$ will be computed correctly in all cases.

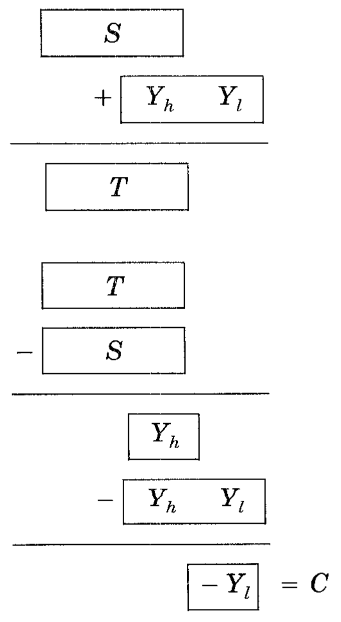
4.3 Errors in Summation

Section 3.2.3 mentioned the problem of accurately computing very long sums. The simplest approach to improving accuracy is to double the precision. To get a rough estimate of how much doubling the precision improves the accuracy of a sum, let $s_1 = x_1$, $s_2 = s_1 \oplus x_2, \ldots, s_i = s_{i-1} \oplus x_i$. Then $s_i = (1 + \delta_i)(s_{i-1} + x_i)$, where $|\delta_i| \le \epsilon$, and ignoring second-order terms in δi gives

$$s_n = \sum_{j=1}^{n} x_j \left(1 + \sum_{k=j}^{n} \delta_k\right) = \sum_{j=1}^{n} x_j + \sum_{j=1}^{n} x_j \left(\sum_{k=j}^{n} \delta_k\right). \quad (31)$$

The first equality of (31) shows that the computed value of $\sum x_j$ is the same as if an exact summation was performed on perturbed values of x_j. The first term x_1 is perturbed by $n\epsilon$, the last term x_n by only ϵ. The second equality in (31) shows that error term is bounded by $n\epsilon \sum |x_j|$. Doubling the precision has the effect of squaring ϵ. If the sum is being done in an IEEE double-precision format, $1/\epsilon \approx 10^{16}$, so that $n\epsilon \ll 1$ for any reasonable value of n. Thus, doubling the precision takes the maximum perturbation of $n\epsilon$ and changes it to $n\epsilon^2 \ll \epsilon$. Thus the 2ϵ error bound for the Kahan summation formula (Theorem 8) is not as good as using double precision, even though it is much better than single precision.

For an intuitive explanation of why the Kahan summation formula works, consider the following diagram of procedure:



Each time a summand is added, there is a correction factor C that will be applied on the next loop. So first subtract the correction C computed in the previous loop from X_j, giving the corrected summand Y. Then add this summand to the running sum S. The low-order bits of Y (namely, Y_l) are lost in the sum. Next, compute the high-order bits of Y by computing $T - S$. When Y is subtracted from this, the low-order bits of Y will be recovered. These are the bits that were lost in the first sum in the diagram. They become the correction factor for the next loop. A formal proof of Theorem 8, taken from Knuth [1981] page 572, appears in the Appendix.

5. SUMMARY

It is not uncommon for computer system designers to neglect the parts of a system related to floating point. This is probably due to the fact that floating point is given little, if any, attention in the computer science curriculum. This in turn has caused the apparently widespread belief that floating point is not a quantifiable subject, so there is little point in fussing over the details of hardware and software that deal with it.

This paper has demonstrated that it is possible to reason rigorously about floating point. For example, floating-point algorithms involving cancellation can be proven to have small relative errors if the underlying hardware has a guard digit and there is an efficient algorithm for binary-decimal conversion that can be proven to be invertible, provided extended precision is supported. The task of constructing reliable floating-point software is made easier when the underlying computer system is supportive of floating point. In addition to the two examples just mentioned (guard digits and extended precision), Section 3 of this paper has examples ranging from instruction set design to compiler optimization illustrating how to better support floating point.

The increasing acceptance of the IEEE floating-point standard means that codes that use features of the standard are becoming ever more portable. Section 2 gave numerous examples illustrating how the features of the IEEE standard can be used in writing practical floating-point codes.

APPENDIX

This Appendix contains two technical proofs omitted from the text.

Theorem 14

Let $0 < k < p$, set $m = \beta^k + 1$, and assume floating-point operations are exactly rounded. Then $(m \otimes x) \ominus (m \otimes x \ominus x)$ is exactly equal to x rounded to $p - k$ significant digits. More precisely, x is rounded by taking the significand of x, imagining a radix point just left of the k least-significant digits, and rounding to an integer.

Proof. The proof breaks up into two cases, depending on whether or not the computation of $mx = \beta^k x + x$ has a carry out or not.

Assume there is no carry out. It is harmless to scale x so that it is an integer. Then the computation of $mx = x + \beta^k x$ looks like this:

$$\begin{array}{rl} & \mathbf{aa} \cdots \mathbf{aabb} \cdots \mathbf{bb} \\ + & \underline{\mathbf{aa} \cdots \mathbf{aabb} \cdots \mathbf{bb} \qquad\qquad} \\ & \mathbf{zz} \cdots \mathbf{zzbb} \cdots \mathbf{bb} \end{array},$$

where x has been partitioned into two parts. The low-order k digits are marked **b** and the high-order $p - k$ digits are marked **a**. To compute $m \otimes x$ from mx involves rounding off the low-order k digits (the ones marked with **b**) so

$$m \otimes x = mx - x \bmod(\beta^k) + r\beta^k. \quad (32)$$

The value of r is 1 if $.\mathbf{bb} \cdots \mathbf{b}$ is greater than 1/2 and 0 otherwise. More precisely,

$$r = 1 \text{ if } \mathbf{a.bb} \cdots \mathbf{b} \text{ rounds to } a + 1, \qquad r = 0 \text{ otherwise}. \quad (33)$$

Next compute

$$m \otimes x - x = mx - x \bmod(\beta^k) + r\beta^k - x$$
$$= \beta^k(x + r) - x \bmod(\beta^k).$$

The picture below shows the computation of $m \otimes x - x$ rounded, that is, $(m \otimes x) \ominus x$. The top line is $\beta^k(x + r)$, where **B** is the digit that results from adding **r** to the lowest order digit **b**:

$$\begin{array}{l} \mathbf{aa \cdots aabb \cdots bB00 \cdots 00} \\ \underline{-\mathbf{bb \cdots bb} \qquad\qquad\qquad} \\ \mathbf{zz \cdots zzZ00 \cdots 00} \end{array}$$

If $\mathbf{.bb \cdots b} < 1/2$, then $r = 0$. Subtracting causes a borrow from the digit marked **B**, but the difference is rounded up, so the net effect is that the rounded difference equals the top line, which is $\beta^k x$. If $\mathbf{.bb \cdots b} > 1/2$, then $r = 1$, and 1 is subtracted from **B** because of the borrow. So again the result is $\beta^k x$. Finally, consider the case $\mathbf{.bb \cdots b} = 1/2$. If $r = 0$, then **B** is even, **Z** is odd, and the difference is rounded up giving $\beta^k x$. Similarly, when $r = 1$, **B** is odd, **Z** is even, the difference is rounded down, so again the difference is $\beta^k x$. To summarize,

$$(m \otimes x) \ominus x = \beta^k x. \qquad (34)$$

Combining eqs. (32) and (34) gives $(m \otimes x) - (m \otimes x \ominus x) = x - x \bmod(\beta^k) + r\beta^k$. The result of performing this computation is

$$\begin{array}{rl} & \mathbf{r00 \cdots 00} \\ & \mathbf{aa \cdots aabb \cdots bb} \\ + & \underline{-\mathbf{bb \cdots bb} \qquad\qquad} \\ & \mathbf{aa \cdots aa00 \cdots 00}. \end{array}$$

The rule for computing r, eq. (33), is the same as the rule for rounding $\mathbf{a \cdots ab \cdots b}$ to $p - k$ places. Thus, computing $mx - (mx - x)$ in floating-point arithmetic precision is exactly equal to rounding x to $p - k$ places, in the case when $x + \beta^k x$ does not carry out.

When $x + \beta^k x$ does carry out, $mx = \beta^k x + x$ looks like this:

$$\begin{array}{rl} & \mathbf{aa \cdots aabb \cdots bb} \\ + & \underline{\mathbf{aa \cdots aabb \cdots bb}} \\ & \mathbf{zzz \cdots zZbb \cdots bb} \end{array}.$$

Thus, $m \otimes x = mx - x \bmod(\beta^k) + w\beta^k$, where $w = -Z$ if $Z < \beta/2$, but the exact value of w in unimportant. Next $m \otimes x - x = \beta^k x - x \bmod(\beta^k) + w\beta^k$. In a picture

$$\begin{array}{l} \mathbf{aa \cdots aabb \cdots bb\,00 \cdots 00} \\ -\mathbf{bb \cdots bb} \\ \underline{+\mathbf{w} \qquad\qquad\qquad\qquad\quad} \end{array}.$$

Rounding gives $(m \otimes x) \ominus x = \beta^k x + w\beta^k - r\beta^k$, where $r = 1$ if $\mathbf{.bb \cdots b} > 1/2$ or if $\mathbf{.bb \cdots b} = 1/2$ and $b_0 = 1$. Finally,

$$(m \otimes x) - (m \otimes x \ominus x)$$
$$= mx - x \bmod(\beta^k) + w\beta^k$$
$$-(\beta^k x + w\beta^k - r\beta^k)$$
$$= x - x \bmod(\beta^k) + r\beta^k.$$

Once again, $r = 1$ exactly when rounding $\mathbf{a \cdots ab \cdots b}$ to $p - k$ places involves rounding up. Thus, Theorem 14 is proven in all cases. ■

Theorem 8 (Kahan Summation Formula)

Suppose $\sum_{j=1}^{N} x_j$ is computed using the following algorithm

```
S = X[1]
C = 0
for j = 2 to N {
  Y = X[j] − C
  T = S + Y
  C = (T − S) − Y
  S = T
}
```

Then the computed sum S is equal to $S = \sum x_j(1 + \delta_j) + 0(N\epsilon^2)\sum |x_j|$, where $|\delta_j| \le 2\epsilon$.

Proof. First recall how the error estimate for the simple formula $\sum x_i$ went. Introduce $s_1 = x_1$, $s_i = (1 + \delta_i)(s_{i-1} - 1 + x_i)$. Then the computed sum is s_n, which is a sum of terms, each of which is an x_i multiplied by an expression involving δ_j's. The exact coefficient of x_1 is $(1+\delta_2)(1+\delta_3)\cdots(1+\delta_n)$. Therefore by renumbering, the coefficient of x_2 must be $(1+\delta_3)(1+\delta_4)\cdots(1+\delta_n)$, and so on. The proof of Theorem 8 runs along exactly the same lines, only the coefficients of x_1 is more complicated. In detail $s_0 = c_0 = 0$ and

$$\begin{aligned}
y_k &= x_k \ominus c_{k-1} = (x_k - c_{k-1})(1+\eta_k)\\
s_k &= s_{k-1} \oplus y_k = (s_{k-1} + y_k)(1+\sigma_k)\\
c_k &= (s_k \ominus s_{k-1}) \ominus y_k\\
&= [(s_k - s_{k-1})(1+\gamma_k) - y_k](1+\delta_k)
\end{aligned}$$

where all the Greek letters are bounded by ϵ. Although the coefficient of x_1 in s_k is the ultimate expression of interest, it turns out to be easier to compute the coefficient of x_1 in $s_k - c_k$ and c_k. When $k = 1$,

$$\begin{aligned}
c_1 &= (s_1(1+\gamma_1) - y_1)(1+\delta_1)\\
&= y_1((1+\sigma_1)(1+\gamma_1) - 1)(1+\delta_1)\\
&= x_1(\sigma_1 + \gamma_1 + \sigma_1\gamma_1)\\
&\quad (1+\delta_1)(1+\eta_1)\\
s_1 - c_1 &= x_1[(1+\sigma_1) - (\sigma_1 + \gamma_1 + \sigma_1\gamma_1)\\
&\quad (1+\delta_1)](1+\eta_1)\\
&= x_1[1 - \gamma_1 - \sigma_1\delta_1 - \sigma_1\gamma_1\\
&\quad - \delta_1\gamma_1 - \sigma_1\gamma_1\delta_1](1+\eta_1).
\end{aligned}$$

Calling the coefficients of x_1 in these expressions C_k and S_k, respectively, then

$$\begin{aligned}
C_1 &= 2\epsilon + O(\epsilon^2)\\
S_1 &= 1 + \eta_1 - \gamma_1 + 4\epsilon^2 + O(\epsilon^3).
\end{aligned}$$

To get the general formula for S_k and C_k, expand the definitions of s_k and c_k, ignoring all terms involving x_i with $i > 1$. That gives

$$\begin{aligned}
s_k &= (s_{k-1} + y_k)(1+\sigma_k)\\
&= [s_{k-1} + (x_k - c_{k-1})(1+\eta_k)]\\
&\quad (1+\sigma_k)\\
&= [(s_{k-1} - c_{k-1}) - \eta_k c_{k-1}](1+\sigma_k)
\end{aligned}$$

$$\begin{aligned}
c_k &= [\{s_k - s_{k-1}\}(1+\gamma_k) - y_k](1+\delta_k)\\
&= [\{((s_{k-1} - c_{k-1}) - \eta_k c_{k-1})(1+\sigma_k)\\
&\quad - s_{k-1}\}(1+\gamma_k) + c_{k-1}(1+\eta_k)]\\
&\quad (1+\delta_k)\\
&= [\{(s_{k-1} - c_{k-1})\sigma_k - \eta_k c_{k-1}(1+\sigma_k)\\
&\quad - c_{k-1}\}(1+\gamma_k) + c_{k-1}(1+\eta_k)]\\
&\quad (1+\delta_k)\\
&= [(s_{k-1} - c_{k-1})\sigma_k(1+\gamma_k)\\
&\quad - c_{k-1}(\gamma_k + \eta_k(\sigma_k + \gamma_k + \sigma_k\gamma_k))]\\
&\quad (1+\delta_k)
\end{aligned}$$

$$\begin{aligned}
s_k &- c_k\\
&= ((s_{k-1} - c_{k-1}) - \eta_k c_{k-1})\\
&\quad (1+\sigma_k)\\
&\quad - [(s_{k-1} - c_{k-1})\sigma_k(1+\gamma_k)\\
&\quad - c_{k-1}(\gamma_k + \eta_k(\sigma_k + \gamma_k + \sigma_k\gamma_k))]\\
&\quad (1+\delta_k)\\
&= (s_{k-1} - c_{k-1})((1+\sigma_k)\\
&\quad - \sigma_k(1+\gamma_k)(1+\delta_k))\\
&\quad + c_{k-1}(-\eta_k(1+\sigma_k)\\
&\quad + (\gamma_k + \eta_k(\sigma_k + \gamma_k + \sigma_k\gamma_k))\\
&\quad (1+\delta_k))\\
&= (s_{k-1} - c_{k-1})\\
&\quad (1 - \sigma_k(\gamma_k + \delta_k + \gamma_k\delta_k))\\
&\quad + c_{k-1}[-\eta_k + \gamma_k\\
&\quad + \eta_k(\gamma_k + \sigma_k\gamma_k)\\
&\quad + (\gamma_k + \eta_k(\sigma_k + \gamma_k + \sigma_k\gamma_k))\delta_k]
\end{aligned}$$

Since S_k and C_k are only being computed up to order ϵ^2, these formulas can be simplified to

$$\begin{aligned}
C_k &= (\sigma_k + O(\epsilon^2))S_{k-1}\\
&\quad + (-\gamma_k + O(\epsilon^2))C_{k-1}\\
S_k &= ((1 + 2\epsilon^2 + O(\epsilon^3))S_{k-1}\\
&\quad + (2\epsilon + O(\epsilon^2))C_{k-1}.
\end{aligned}$$

Using these formulas gives

$$C_2 = \sigma_2 + O(\epsilon^2)$$

$$S_2 = 1 + \eta_1 - \gamma_1 + 10\epsilon^2 + O(\epsilon^3),$$

and, in general, it is easy to check by induction that

$$C_k = \sigma_k + O(\epsilon^2)$$

$$S_k = 1 + \eta_1 - \gamma_1 + (4k + 2)\epsilon^2 + O(\epsilon^3).$$

Finally, what is wanted is the coefficient of x_1 in s_k. To get this value, let $x_{n+1} = 0$, let all the Greek letters with subscripts of $n + 1$ equal 0 and compute s_{n+1}. Then $s_{n+1} = s_n - c_n$, and the coefficient of x_1 in s_n is less than the coefficient in s_{n+1}, which is $S_n = 1 + \eta_1 - \gamma_1 + (4n + 2)\epsilon^2 = 1 + 2\epsilon + O(n\epsilon^2)$. ■

ACKNOWLEDGMENTS

This article was inspired by a course given at Sun Microsystem by W. Kahan, ably organized by David Hough of Sun, from May through July 1988. Its aim is to enable others to learn about the interaction of floating point and computer systems without having to get up in time to attend 8:00 am lectures. Thanks are due to Kahan and many of my colleagues at Xerox PARC (especially John Gilbert) for reading drafts of this paper and providing many useful comments. Reviews from Paul Hilfinger and an anonymous referee also helped improve the presentation.

REFERENCES

AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1986. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Reading, Mass.

ANSI 1978. *American National Standard Programming Language FORTRAN, ANSI Standard X3.9-1978.* American National Standards Institute, New York.

BARNETT, D. 1987. A portable floating-point environment. Unpublished manuscript.

BROWN, W. S. 1981. A simple but realistic model of floating-point computation. *ACM Trans. Math. Softw. 7*, 4, 445-480.

CARDELLI, L., DONAHUE, J., GLASSMAN, L., JORDAN, M., KASLOW, B., AND NELSON, G. 1989. Modula-3 Report (revised). Digital Systems Research Center Report #52, Palo Alto, Calif.

CODY, W. J. et al. 1984. A proposed radix- and word-length-independent standard for floating-point arithmetic. *IEEE Micro 4*, 4, 86-100.

CODY, W. J. 1988. Floating-point standards—Theory and practice. In *Reliability in Computing: The Role of Interval Methods on Scientific Computing*, Ramon E. Moore, Ed. Academic Press, Boston, Mass., pp. 99-107.

COONEN, J. 1984. Contributions to a proposed standard for binary floating-point arithmetic. PhD dissertation, Univ. of California, Berkeley.

DEKKER, T. J. 1971. A floating-point technique for extending the available precision. *Numer. Math. 18*, 3, 224-242.

DEMMEL, J. 1984. Underflow and the reliability of numerical software. *SIAM J. Sci. Stat. Comput. 5*, 4, 887-919.

FARNUM, C. 1988. Compiler support for floating-point computation. *Softw. Pract. Experi. 18*, 7, 701-709.

FORSYTHE, G. E., AND MOLER, C. B. 1967. *Computer Solution of Linear Algebraic Systems.* Prentice-Hall, Englewood Cliffs, N.J.

GOLDBERG, I. B. 1967. 27 Bits are not enough for 8-digit accuracy. *Commum. ACM 10*, 2, 105-106.

GOLDBERG, D. 1990. Computer arithmetic. In *Computer Architecture: A Quantitative Approach*, David Patterson and John L. Hennessy, Eds. Morgan Kaufmann, Los Altos, Calif., Appendix A.

GOLUB, G. H., AND VAN LOAN, C. F. 1989. *Matrix Computations.* The Johns Hopkins University Press, Baltimore, MD.

HEWLETT PACKARD 1982. *HP-15C Advanced Functions Handbook.*

IEEE 1987. IEEE Standard 754-1985 for Binary Floating-Point Arithmetic, IEEE. Reprinted in *SIGPLAN 22*, 2, 9-25.

KAHAN, W. 1972. A Survey of Error Analysis. In *Information Processing 71*, (Ljubljana, Yugoslavia), North Holland, Amsterdam, vol. 2, pp. 1214-1239.

KAHAN, W. 1986. Calculating Area and Angle of a Needle-like Triangle. Unpublished manuscript.

KAHAN, W. 1987. Branch cuts for complex elementary functions. In *The State of the Art in Numerical Analysis*, M. J. D. Powell and A Iserles, Eds., Oxford University Press, N.Y., Chap. 7.

KAHAN, W. 1988. Unpublished lectures given at Sun Microsystems, Mountain View, Calif.

KAHAN, W., AND COONEN, J. T. 1982. The near orthogonality of syntax, semantics, and diagnostics in numerical programming environments. In *The Relationship between Numerical Computation and Programming Languages*, J. K. Reid, Ed. North-Holland, Amsterdam, pp 103-115.

KAHAN, W., AND LEBLANC, E. 1985. Anomalies in the IBM acrith package. In *Proceedings of the 7th IEEE Symposium on Computer Arithmetic* (Urbana, Ill.), pp. 322-331.

KERNIGHAN, B. W., AND RITCHIE, D. M. 1978. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, N.J.

KIRCHNER, R., AND KULISCH, U 1987 Arithmetic for vector processors. In *Proceedings of the 8th IEEE Symposium on Computer Arithmetic* (Como, Italy), pp. 256-269.

KNUTH, D. E. 1981. *The Art of Computer Programming* Addison-Wesley, Reading, Mass., vol. II, 2nd ed.

KULISH, U. W., AND MIRANKER W. L. 1986. The Arithmetic of the Digital Computer: A new approach. *SIAM Rev 28*, 1, 1-36.

MATULA, D. W., AND KORNERUP, P. 1985. Finite Precision Rational Arithmetic: Slash Number Systems. *IEEE Trans. Comput. C-34*, 1, 3-18.

REISER, J. F., AND KNUTH, D E. 1975. Evading the drift in floating-point addition. *Inf. Process. Lett 3*, 3, 84-87

STERBETZ, P. H. 1974. *Floating-Pomt Computation*. Prentice-Hall, Englewood Cliffs, N.J.

SWARTZLANDER, E. E , AND ALEXOPOULOS, G. 1975. The sign/logarithm number system. *IEEE Trans. Comput. C-24*, 12, 1238-1242

WALTHER, J. S. 1971. A unified algorithm for elementary functıons. *Proceedings of the AFIP Sprıng Joınt Computer Conference*, pp. 379-385.

Receıved December 1988; fınal revısıon accepted March 1990