Computer System Support for Scientific and Engineering Computation

Lecture 5, May 17, 1988 (revision date June 24, 1988)

1 Conventional Floating-Point Formats

Let's remind ourselves what we have to deal with. We have to deal with radix β, which has to be drawn from a set of numbers

$$\text{Radix} \quad \beta \in \{2, 4, 8, 10, 16\}$$

to correspond to machines actually in existence ($\beta = 4$ may be an exception, but we might just include it in although there are no machines that I know of today that use radix 4). I also talked about digits d_j

$$\text{Digits} \quad d_j \in \{0, 1, 2, \ldots, \rho - 1, \rho\} \quad \text{where } \rho = \beta - 1.$$

Now, conventional floating-point representation has to pass somehow three fields:

Sign	$\pm$
Significand, Mantissa, or Coefficient	$[d_1 d_2 \cdots d_{p-1} d_p]$, p "significant" digits of Precision
Exponent	e, stored as e+Bias, an integer.

The *sign* field normally takes up one bit in binary but there are other representations that may take up a whole character. The second field has a number of names: you could call it *Coefficient*, or *Mantissa* ("Mantissa" is a very bad word: originally it represented the fraction part of a logarithm), or *Significand*. It seems that "Significand" is widely used now. Finally the *Exponent* field is simply an integer, often stored with bias (will be explained in a moment). Once you got these three fields, the interpreted value is as follows.

$\pm[\underbrace{d_1 d_2 \cdots d_{p-1} d_p}_{\text{Integer Coefficient}}] \times \beta^{e_p}$	CDC Cyber 17× Burroughs B65××
$\pm[.d_1 d_2 \cdots d_{p-1} d_p] \times \beta^{e_0}$	CDC Cyber 18× IBM 370 DEC VAX "C" Language
$\pm[d_1 . d_2 \cdots d_{p-1} d_p] \times \beta^{e_1}$	"Scientific" notation Calculators IEEE 754/854
... others ...	...

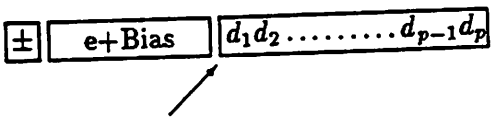
All values are the same if exponents are correlated:

$$e_0 = e_p + p = e_1 + 1.$$

Thus if you like to put the β-point in another place, you can accommodate that simply by changing the bias. What it really means is that the position in which you put the point has very little to do with the hardware (hardware is going to use integer arithmetic anyway). The trouble is that the representation is non-unique:

$$0.0123 = 0.123 \times 10^{-1} = 1.23 \times 10^{-2}$$

For hardware reasons it is a good idea to have a unique representation. So that is the purpose of normalization. A representation of a number will be called normalized if its first digit $d_1 \neq 0$.



$d_1 \neq 0$ for *normalized nonzero* number

An *unnormalized* nonzero number has $d_1 = 0$ but $e >$*minimum*. It is just another way to represent the value of a normalized number. Now whether zero is a *normalized* number or not is uninteresting and I won't argue about that, except that there are some unnormalized zeros:

Normalized zero : $d_1 = \ldots = d_p = 0, e =$ standard value, usually minimum.
Unnormalized zero : $d_1 = \ldots = d_p = 0, e \neq$ standard value.

There are reasons for unnormalized zeros on some machines. They can be used for converting a number to an integer represented in floating-point (cf. Fortran's AINT()). Finally there are something call subnormal:

Subnormal(denormalized)number: $d_1 = 0$, $e =$ minimum value.

Note that because of the minimum exponent, subnormal numbers are *unique* representations for their numerical values, so they are *not unnormalized* numbers.

Examples: on a calculator with format $\pm d.ddd \times 10^{\pm ee}$

0.0123×10^{0} is unnormalized
1.230×10^{-2} is normalized
0.012×10^{-99} is subnormal
0.000×10^{0} is normal zero by convention.

Note that the zero exponent in the normal zero is for display purposes. Internally it could be 0.000×10^{-99}. A more humane way is simply to display a single digit 0. That way you know you really got a zero rather than something that's hidden in the right hand side of the display window.

2 Why are most floating-point formats intended to be normalized?

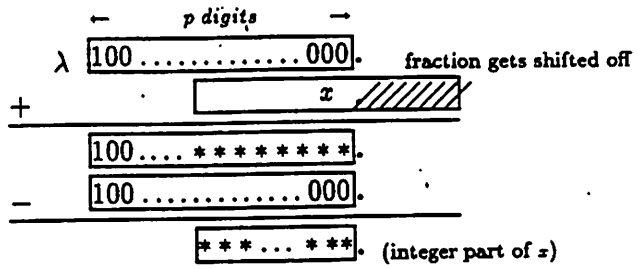
The reason for the normalization is to make the hardware simpler to design, especially for addition and subtraction.

Example: Add/Subtract $3.0 + 0.012 \Rightarrow 3.01$ to 3 significant decimals.

3.00×10^0		0.03×10^2	0.03×10^2
$+ 1.20 \times 10^{-2}$		$+ 1.20 \times 10^{-2}$	$+ 1.20 \times 10^{-2}$
3.00×10^0		0.03×10^2	0.03×10^2
0.012×10^0		0.00012×10^2	0.00012×10^2
3.012×10^0		0.03012×10^2	0.03012×10^2
↓		↓	normalization ↙
3.01×10^0	$\neq$	0.03×10^2	3.012×10^0
			↓
As usual		just 3. !!	3.01
		in Unnormalized arithmetic IBM 370 ...	on Burroughs B65xx using double-width accumulator. (cf ALGOL)

In the first column, when adding 0.012 and 3.0, the hardware would first discover that the operands' exponents are different, so it would initially do a pre-alignment. The last digit "2" has to go away. An interesting question is *when* should it go, and machines differ. Some machines will throw away the digit "2" before the addition, which doesn't do too much damage to the sum here, but you will see later that it can do a great deal of damage to subtraction. The point is, registers don't normally hold enough digits to keep everything that has been shifted right.

Now, the second column demonstrates that if an unnormal representation of 3.00 is allowed, the hardware will need four extra positions to hold all the digits. That seems to require a lot of extra digits. Most machines have floating-point add built in such a way that they only carry one or two extra digits. If that is the case the sum would end up with an incorrect result. That is why people say that unnormalized arithmetic is less accurate. It is not the unnormalized number that is less accurate; rather the hardware is designed in such a way that if you do arithmetic with unnormalized operands, you may get inaccurate results. That could be intentional because the shifting process provide a way to take the integer part of a number. Recall (see earlier notes for ceil and floor) that to get the integer part of a number, you add it to $\lambda = \beta^{p-1}$ and then subtract λ from the sum. The fraction part of x will then be rounded off.



That is a pain in the neck to get the integer part this way. There are people who say as long as you allow unnormalized zero in hardware, the same thing can be done faster by simply adding x to an aptly chosen unnormalized zero. The picture is the same as above except that the leading 1 of λ now is 0, and you only do the addition once. However, one should be careful about the exponent of the unnormalized zero (not necessarily p). For the hardware may carry extra digits (i.e., the fraction part of x may not be thrown away entirely if there are only p zeros ahead of the point) and you have to compensate by putting more digits in the unnormalized zero. I suppose it would make more sense to have an explicit instruction that converts to integer. That is what the IEEE Standard specified.

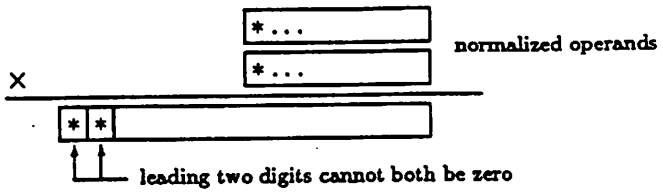
Finally, the third column on the previous page is what the Burroughs B65×× machine does. Burroughs originally designed its B5000 machine to suit the language ALGOL. One of the characteristic of ALGOL is that integer operations may result in floating-point numbers. This avoids the conundrum in Fortran which would return 0 when computing 1/3 because Fortran forces the result to be an integer. So ALGOL has both integer and floating-point "REAL" numbers in the sense that you cannot tell them apart. This was interpreted unnecessarily by the hardware people to mean that integers should look like floating-point numbers, and that's what they did. They encoded the floating-point numbers in such a way that the floating-point encoding for at least an unnormalized representation of an integral value would coincide bitwise with the encoding for an integer. Thus an unnormalized number could be as legitimate as others and in principle, any arithmetic you do with an unnormalized number should be the same as with its normalized counterpart. Burroughs accomplished that by keeping all the digits that other folks might throw away. In fact, the arithmetic was done in a double-width register (plus a digit), and then the result would be shifted to the left until either the left hand register filled or the bottom register is cleared. That explains why on Burroughs machines it is perfectly reasonable to have unnormalized numbers.

There have been various policies toward unnormalized operands. To summarize,

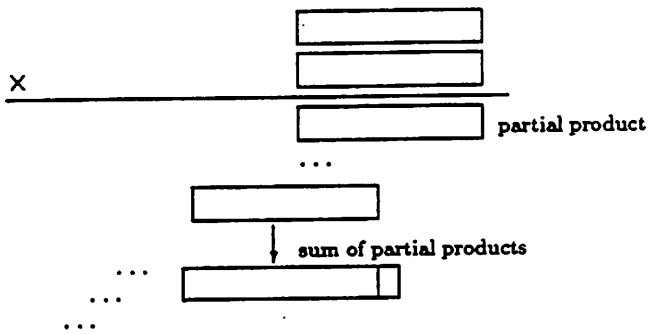
- IBM 370 Prenormalizes operands before $\times, \div$; not before $\pm$ (for the sake of AINT(x)).
- DEC VAX has no unnormalized operands
- IEEE 754/854 allows no unnormalized operands, but includes subnormal operands for gradual underflow.

Remark 1. Prenormalization before $\times$ and $\div$ is built in IBM 370 hardware. It is done that way to avoid unnecessary loss of accuracy: otherwise you would end up with a zero product if half of the leading bits in both operands were zeros. Or, if you decide to normalize the result, then you have to figure out in advance the amount of post-shifting, and you may end up with something that is harder to build. (But CYDOME's CYDRA5 does work this way.)

Remark 2. Another reason for prenormalization is cheaper hardware. It is possible to do the multiplication with a register that is just a little bit wider than the leading word if all you want is the leading word of the product. All you need are normalized numbers: for the product of two normalized numbers (length p) must have length either $2p$ or $2p-1$; consequently you wouldn't have to carry more than one extra digit (called the *guard* digit) in order to know that there might be a post-shift.



Thus it's enough to use a register with some extra bits to hold the sum of the partial products:



3 Exponent Bias and Range Asymmetry

Why is there a Bias for the exponent? Part of the reason is to compensate for where you may have put the "point" in the documentation. However, the bias does turn up when you ask how balanced or symmetrical is the range of numbers. If you multiply the biggest number with the smallest normal number, then you get

$$\text{(Max Normal)}\times\text{(Min Normal nonzero)} \approx (\beta \times \beta^{e_{max}})(1 \times \beta^{e_{min}}) = \beta^{1+e_{max}+e_{min}}$$

If this product is too far from 1, the range is regarded as too asymmetrical, and causes troubles with expressions like

$$q := (\text{small integer})/x \qquad \ldots\text{can over/underflow too easily ?}$$

In other words, the reciprocal of the smallest number may be much bigger than the biggest number (or the other way around). In that case over/underflow may come to you as a surprise. Here is another problem:

$$q := x \cdot y/z \qquad \ldots\text{what order avoids spurious over/underflow ?}$$

We'll come back to the above problem later. Now let's see how to choose a bias. Recall if

$$X : \boxed{\pm}\boxed{e + Bias}\boxed{d_1 d_2 \ldots\ldots d_{p-1} d_p}$$

then the interpreted value is

$$x = \pm[d_1 \cdot d_2 \ldots d_{p-1} d_p] \times \beta^e, \quad e_{min} \le e \le e_{max}. \tag{1}$$

Normally the exponent field $e + Bias$ would be stored in a field l bits wide; thus

$$0 \le e + Bias \le 2^l - 1.$$

It follows that

$$e_{min} = -Bias, \quad e_{max} = 2^l - 1 - Bias.$$

Therefore $1 + e_{max} + e_{min} = 2^l - 2 \cdot Bias$. That explain how the choice of $Bias$ can influence the symmetry of the exponent range. If you want not to have too asymmetrical a range, you should choose $Bias$ near 2^{l-1}. If overflow is more to be avoided than underflow (e.g. if underflow is gradual), choose $Bias = 2^{l-1} - 1$. (DEC VAX[1]: $Bias = 2^{l-1} + 1$ predisposes to overflow!)

Range Asymmetry is Severe on Some Machines:

- CDC Cyber 17x:

$$\begin{aligned} \text{Max. magnitude} &\approx 2^{1022+48} \\ \text{Min. normal magnitude} &= 2^{-1022+47} \end{aligned}$$

$$\boxed{(\text{Max}) \cdot (\text{Min}) \approx 2^{95}}$$

Hence 2^{94}/Max. *underflows* !

- Similar trouble on Burroughs B65××.

IEEE 754, single precision:

$$\begin{aligned} \text{Max.} &\approx 2^{128} \\ \text{Min. Normal} &= 2^{-126} \\ \text{Min. Subnormal} &= 2^{-149} \end{aligned}$$

$$\boxed{\begin{array}{c} (\text{Max}) \cdot (\text{Min. Normal}) \approx 4 \\ (\text{Max}) \cdot (\text{Min. Subnormal}) \approx 2^{-21} \end{array}}$$

Hence 2^{-20}/(Min. Subnormal) *overflows*.

Remark. Subnormal numbers are intended to cope with certain situations in addition and subtraction (see Lecture 7b, section 4), not multiplication and division. Dividing a number by a subnormal number and getting an overflow simply tells the user the bad news of the earlier loss of significant digits. On machines that flush subnormal numbers to zero you will encounter division by zero anyway. So we choose our bias by looking at (Max)·(Min. Normal) rather than (Max)·(Min. Subnormal).

[1] VAX's manual said $Bias = 2^{l-1}$, but here we interpreted x's value as in (1), whereas VAX puts the "point" before d_1.

Exercise: Exhibit a program that starts from any three given positive numbers x, y, z and computes $p := x \cdot y \cdot z$ in some order that avoids undeserved over/underflow. Do likewise for $q := x \cdot y/z$.

The first one is easy. Here is how you do it: sort x, y, z so that, say,

$$x \geq y \geq z > 0,$$

and then compute

$$p := (x \cdot z) \cdot y.$$

That is, you multiply two extreme numbers first. You should be able to verify, if p is computed that way, over/underflow occurs only if it has to. The second one is not that easy. We could sort x, y, and $1/z$ except $1/z$ might over/underflow.

Can you find a way?

I will leave to you the indistinct possibility that you might be able to find a portable program that will work on all machines. This is a problem that does arise in "real life" from time to time; cf. program SVD2 × 2.

4 Lexicographic Order

The format we have chosen for floating-point numbers preserves "lexicographic order". That is the reason we chose a biased exponent rather that an exponent with an independent sign (like Burroughs). Consider the string X

$$String \quad X: \quad \boxed{\pm}\ \underbrace{\boxed{e + Bias}}_{l\ bits}\ \boxed{d_1 d_2 \ldots\ldots\ldots d_{p-1} d_p}$$

with its floating-point value

$$x = \pm[d_1 \cdot d_2 \ldots d_{p-1} d_p] \times \beta^e, \quad 0 \leq e + Bias \leq 2^l - 1.$$

Assume X is Normalized (i.e., either $d_1 > 0$ or $(e + Bias) = [d_1 d_2 \ldots d_{p-1} d_p] = 0$). Then the order of x's as floating-point values is the same as the order of X's as integers with the same sign-convention. This permits *quick* floating-point comparison without doing floating-point subtraction. Note that we must have the correct sign-convention in order to make the lexicographic ordering work. Let's examine three sign-conventions that come to mind:

X	+	$e + Bias$	$d_1 d_2 \ldots\ldots\ldots d_{p-1} d_p$
$-X$	−	What ...	... goes in here?

1. Sign-magnitude, used by IBM, DEC, IEEE 754. Only the sign bit changes.
2. 1's complement, used by CDC Cyber 17×. The exponent and significand are 1's-complemented as well as the sign.
3. 2's complement, used by HP-3000, GE/Honeywell. The entire word is 2's-complemented.

Here are the crucial considerations:

- Sign-Magnitude ≡ 1's complement has ± 0, allows $\pm\infty$.
- 2's complement has one *unsigned* 0, must have a negative value that either *overflows* when negated, or is an outlaw (e.g. unsigned ∞).

It turns out that Sign-Magnitude is best because it allows ± 0. There are valuable uses for a signed zero in complex arithmetic, especially in conformal maps of slitted domains (cf. Powell&Iserles(ed.) (1987) "The state of the Art in Numerical Analysis" ch.7 Oxford U.P.).

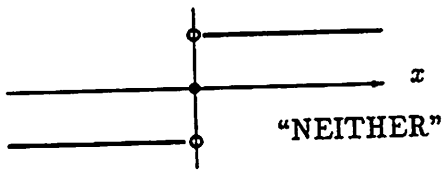
People may get uncomfortable about signed zero because it is very difficult to understand how to distinguish a $+0$ and a -0. In fact on a well designed machine they are equal:

$$+0 = -0.$$

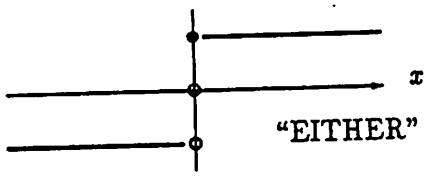
But, if $f(x)$ is *discontinuous* at $x = 0$, then it *may* distinguish $f(+0)$ from $f(-0)$. Consider to which side of a discontinuity the discontinuity itself should be attached:

e.g.

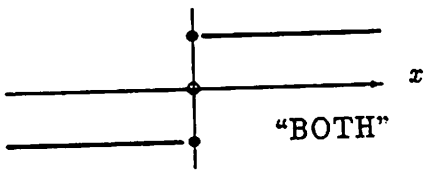
SIGNUM(x)
SIGNUM(± 0) = ± 0



Fortran SIGN(1.0,x)
SIGN(1.0, ± 0) = $+1$



IEEE COPYSIGN(1.0,x)
COPYSIGN(1.0, ± 0) = ± 1



The discontinuity of SIGNUM function is attached to neither of the continuous pieces. The Fortran SIGN function attaches the discontinuity to one of its two continuous components. IEEE COPYSIGN function attaches the discontinuity at both sides, making the graph of the function two *closed* components.

So the ambiguity of signed zero has to be resolved only when the function is actually discontinuous at zero; otherwise you can't care. The reciprocal function is an example: if you take the reciprocal of zero, its sign shows up

$$\frac{1.0}{\pm 0.0} = \pm\infty \quad \Rightarrow \quad \frac{1}{1/x} \approx x \text{ even for } x = \pm\infty.$$

It's true that most poeple don't care about these uses, and therefore it is vitally important that signed zeros be implemented in such a way that for every operation that is continuous at zero, they cannot be told apart. That is true in the IEEE Standards: $+0 = -0$ unless you have a discontinuous operation in mind. The signed zero propagates properly in the IEEE Standards, e.g., $3.0 \times (-0.0) = -0.0$ (on an IBM 370, $3.0 \times (-0.0) = +0.0$).

There are some conundrums about signed zero. For example, thanks to minus zero, we have to ask compiler writers not to replace $a - b$ by $-(b - a)$ because they have opposite signs when they are zero. It OK to do $(-b) + a$ though. So minus zero does pose some technical problems in the handling of details. But those problems in all situations that I know of do not introduce any performance disadvantage. That's why IEEE Standards have minus zero: it is useful, and doesn't cost anything except to the implementors.

5 Precision = Resolution (not Accuracy)

Precision is a measure of the resolving power of the arithmetic; accuracy is something that tells how right or wrong the result is. If I write $\pi = 3.14159023487628374$ then the precision is the number of digits (18) I wrote but the accuracy is only six figures. I can utter my inaccurate approximation of π to a *very high precision.* The distinction is necessary because in general people believe that if they carry more precision they will get more accuracy. That is usually true but beautiful examples (which we will come to) show that carrying more precision doesn't improve accuracy in every instance.

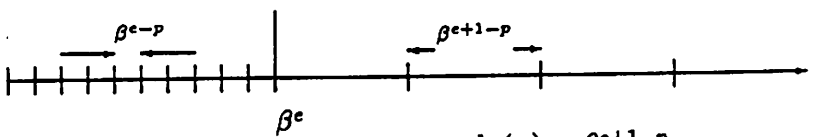
The *Relative Precision* to which you can specify a number x is the following quotient:

$$\frac{\text{ulp}(x)}{|x|}.$$

Here ulp stands for unit in the last place and if $x = \pm[d_1 \cdot d_2 \ldots d_{p-1} d_p] \times \beta^e$, then $\text{ulp}(x) = [0 \cdot 0 \ldots 01] \times \beta^e$. In general, if $\beta^e < x < \beta^{e+1}$ then $\text{ulp}(x) = \beta^{e+1-p}$. But there is ambiguity:

$$\text{ulp}(\beta^e) = \beta^{e+1-p} \text{ or } \beta^{e-p}$$

depending on whether you look left or right.



$\text{ulp}(x) = \beta^{e-p}$ $\qquad$ $\text{ulp}(x) = \beta^{e+1-p}$

Every time you cross a power of the radix the unit in the last place jumps by a factor of β. This is called *wobbling precision.* The larger the radix, the worse the wobble.

$$\underbrace{\beta^{-p} \leq \frac{\text{ulp}(x)}{|x|} \leq \beta^{1-p}}_{\text{"Wobble" by factor } \beta}$$

6 Error in Floating-point Operations

Let $\circledast \in \{+, -, \times, /\}$. Let A, B, C be the corresponding names of variables whose values are a, b, c. I'll use this notation is to distinguish between the assignment statement in a program and the corresponding mathematically exact algebraic relation.

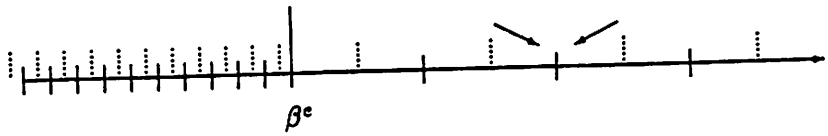
Program statement	"A	:=	$B \circledast C$"
produces value	a	=	$b \circledast c$ *Rounded*

The discrepancy between the exact result and the rounded one is easy to figure out if rounding is done in the best possible way: you will lose only half of the unit in the last place.

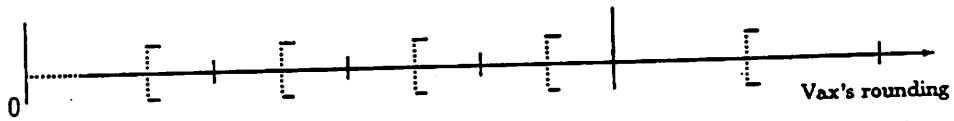
$$b \circledast c = \pm x.xxx\ldots\ldots xxx\ xxxx\ldots$$
$$a = \pm x.xxx\ldots\ldots xxx$$
$$\longleftarrow\ p\ \longrightarrow$$

$$\textit{correct}\ \text{rounding} \Rightarrow |a - b \circledast c| \leq \frac{1}{2}\text{ulp}(b \circledast c)$$

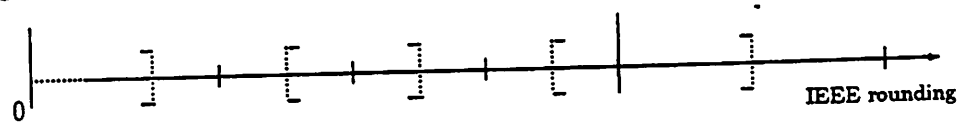
There are two special cases: (i) $a = b \circledast c$ if it is representable exactly, and (ii) $a =$ *something else* if over/underflow occurs. So values that fall between two dotted lines will be rounded to the center number (heavily marked):



The only ambiguity now is what if $b \circledast c$ falls midway between two adjacent representable values (i.e., the value lies on the dotted line)? This ambiguity gets resolved in a variety of ways, of which only two deserve close attention. These two roundings are both called *correct rounding* because their error doesn't exceed half of an ulp. But one is slightly more correct than the other. First let's look at DEC VAX. DEC VAX rounds midway case *away* from zero, i.e., *up* in magnitude:



Here we use a dotted left bracket to indicate that the value at the dotted line will be rounded to the right. IEEE 754/854 rounds midway cases to *nearest even*:

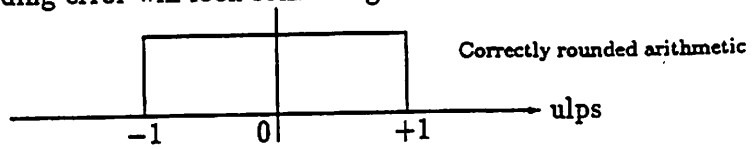


Here the representable number inside each pair of dotted brackets has its least significant bit equal to zero. We could have rounded to nearest odd but that would be bad news. Rounding to nearest odd would have a propensity to convert nearly integers to definitely not integers. Rounding to nearest even is the other way around; it is predisposed towards small integers and other *simple* values.

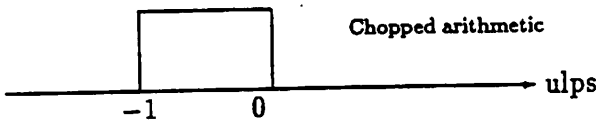
e.g. to 4 significant decimals

	IEEE854 (HP-71B)	like DEC VAX (HP-15C)
$3.1415 \hookrightarrow$	3.142	3.142
$3.1405 \hookrightarrow$	3.140	3.141
$3.1395 \hookrightarrow$	3.140	3.140
$3.1385 \hookrightarrow$	3.138	3.149
$3.1375 \hookrightarrow$	3.138	3.148

A fair question is why do we prefer correct rounding? The reason has been described in terms of the law of averages which the bias in rounding is smaller if you do round correctly. Here is a brief argument (by the way this is *not* the best argument): imagine that the values you round vary a lot and their digits behave randomly. For correct rounding, it looks as if as many numbers are being rounded up as are being rounded down. Roughly speakly the distribution of rounding error will look something like

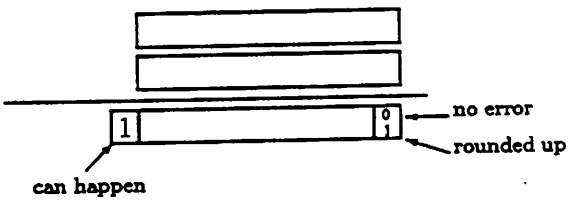


Here I am plotting the magnitude of rounding error in ulps. What this tells us is that we can calculate means and standard deviations. If all the rounding errors contribute their effect independently, then the means look like 0. That's a lot better than chopping (IBM 370, Cray, Cyber 170). The error for chopped arithmetic is biased (probabilistically). On the average it is $-\frac{1}{2}$ulp:

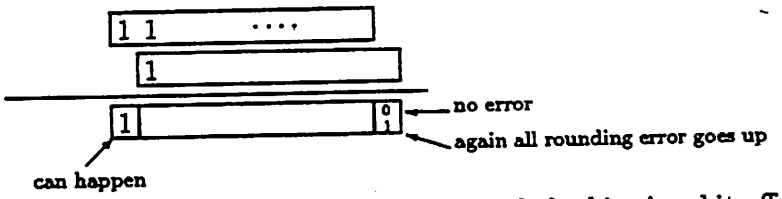


This bias shows up in many computation. For many years people have decried IBM for this fault because you can do computations (e.g. a long string of multiplies and adds) where you see your numbers steadly and systematically drifting downwards as you keep on computing with them.

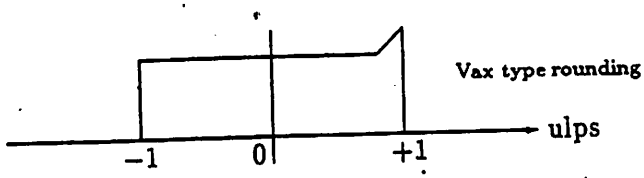
The VAX's type of rounding seems to have bias equal to 0. But that's not quite true. Here is the reason. A typical add might not involve any pre-shifting. It is often found when you do a floating-point add that the exponent of the operands are the same. The result may have a carry on the left, which means you have to shift one digit off:



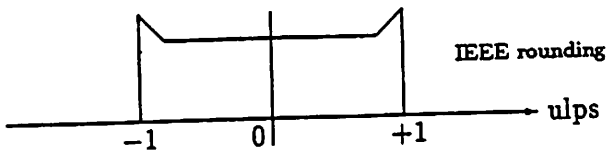
Consequently, all such rounding errors go up on a VAX. Another similar situation is a one-bit shifted subtract:



So there is a spike near +1 caused by the midway round up. And the bias is a bit off zero.



On the other hand, the IEEE rounds midway cases to nearest even, which means the rounding may go down as well as go up, depending on the least significant bit. So there are two spikes: one on -1 and one on $+1$. And the bias would again very near zero.



So that's the argument (not a very good one) for IEEE rounding in term of the law of averages. Unfortunately the law of averages is not something you should not depend on too much. Because when you get hurt by floating-point computation (which is rare), the pain is caused by the worst case, not the average. A much better argument comes about because certain computations are repetitive:

```
loop:  y = x+z
       ...
       ...      (y,z unchanged inside the loop)
       ...
       x = y-z
       goto loop
```

This may occur in solving a large system of coupled equations, when some unknowns converge faster than the others. Now the question is what does the loop do to the value of x? A remarkable theorem says, if the value of x changes it will change once and never again, provided you use the IEEE rounding. If you round as the Vax does, you run the risk that any time you go around this cycle, the value of x may keep on drifting along one direction. That is the reason that we really want to round to nearest even; it prevents certain kinds of cyclical operations from drifting. This is a very general result and is often seen under the name of Harry Diamond's theorem[2]. What the theorem says is roughly the following: if you compute the monotonic convex function $y = f(x)$ and then its inverse $x = f^{-1}(y)$, repeatedly, rounding off each time, then if you round the way IEEE Standard does, you could only make at most two changes in x.

7 Range/Precision Tradeoff for Radices $\beta = 2^k$

There are machines with various radices of the form $\beta = 2^k$:

[2]Harry Diamond didn't prove this theorem, but he proved something so near that it's worth attaching his name to it.

Name	β	k	Who?
Binary	2	1	IEEE 754, DEC VAX, CDC, CRAY, ...
Quaternary	4	2	...no more ...
Octal	8	3	Burroughs B65××
Hexadecimal	16	4	IBM 370, Amdahl, ...

The question is which radix is *best*? To answer this question, let's work out the wordsize, the range, and the precision of radix $\beta = 2^k$. Consider the floating-point word:

$$X = \overbrace{\boxed{\pm}}^{1}\ \overbrace{\boxed{e + Bias}}^{\leftarrow l \text{ bits} \rightarrow}\ \overbrace{\boxed{d_1 d_2 \ldots\ldots\ldots\ldots\ldots\ldots d_{p-1} d_p}}^{\leftarrow \quad p \text{ sig. digits} = p \cdot k \text{ bits} \quad \rightarrow}$$

Here for convenience the interpreted value has no "point" after d_1:

$$x = \pm\beta^e \times [d_1 d_2 \ldots d_{p-1} d_p]$$

where $\beta = 2^k$ and

$$\begin{aligned} 0 &\leq e + Bias \leq 2^{l-1} \\ 0 &\leq [d_1 d_2 \ldots d_{p-1} d_p] \leq \beta^p - 1 \end{aligned}$$

Using the above notation, we have

$$\boxed{\text{Total wordsize: } w = 1 + l + p \cdot k \text{ bits}}$$

Let $\rho = \beta - 1$ so $[00\ldots00] \leq [d_1 d_2 \ldots d_{p-1} d_p] \leq [\rho\rho\ldots\rho\rho]$. Normally X is normalized, i.e., $d_1 \geq 1$ unless $x = 0$. Now the range over which X varies has

$$\begin{aligned} \frac{\text{Max. } x}{\text{Min. } x > 0} &= \frac{\beta^{2^l - 1 - Bias} \times [\rho\rho\ldots\rho\rho]}{\beta^{0 - Bias} \times [10\ldots00]} \\ &= \frac{\beta^{2^l - 1} \times (\beta^p - 1)}{\beta^{p-1}} \\ &\approx \beta^{2^l} = 2^{k \cdot 2^l}. \end{aligned}$$

Thus

$$\boxed{\text{Range: } 2^{k \cdot 2^l}}$$

Finally the worst-case precision is

$$\max_{x>0} \frac{(successor\ of\ x) - x}{x} = \frac{[100\ldots001] - [100\ldots000]}{[100\ldots000]} = \frac{1}{\beta^{p-1}} = 2^{k \cdot (1-p)}.$$

Thus

$$\boxed{\text{Worst-Case Precision: } 2^{k \cdot (1-p)}}$$

Now we ask what is the word size of a binary format that has the same *range* and *worst-case precision* as $\beta = 2^k$? The result is interesting: you end up with smaller wordsize if you use binary. Here is the analysis. Let's say we use l' exponent bits and p' significant bits for binary. To achieve the same range and precision, we must have

$$2^{2^{l'}} = 2^{k \cdot 2^l}, \quad 2^{1-p'} = 2^{k \cdot (1-p)}.$$

Thus $l' = l + \log_2 k$ and $p' = 1 + k \cdot (p-1)$, and the wordsize is $w' = 1 + l + \log_2 k + k \cdot (p-1) = w - (k - log_2 k - 1)$. Hence $w - w' = k - log_2 k - 1 \geq 0$ for all $k \geq 1$. The following table shows how many digits get lost for $k = 1, 2, 3, 4$:

Name	k	lost bits $(k - log_2 k - 1)$
Binary	1	0 (−1 for Hidden bit!)
Quaternary	2	0
Octal	3	$(2 - log_2 3) \approx 0.415$
Hexadecimal	4	1

Note that it seems binary and quaternary are the same. However in binary with normalized numbers the leading bit is 1 and you don't have to store it. You can't do that for other radices. So with a hidden bit,

Binary beats	Quaternary	by	1 bit
	Octal	by	1.415 bits
	Hex.	by	2 bits

That is an argument that has been used to prove that binary is best; the argument is not that one can save a few bits using binary radix, but that for a given memory wordsize it has greater precision than other radices. not much of an argument since memory is cheap. But a stronger argument is *WOBBLING PRECISION* (see section 5). In order to understand the importance of wobbling precision, you should read my note "Roundoff in Polynomial Evaluation, 18 October 1986". On the other hand the advantage of decimal is that people can understand it. So to conclude: use decimal when your machines will produce numbers that you expect people to read (people like tax preparers and collectors). Because they think decimal, run your machine in decimal so you do just what they naively imagine you do. Otherwise, use binary (for engineers, scientists, etc).

CONVENTIONAL FLOATING-POINT FORMATS

Radix $\beta = 2, 4, 8, 10$ or 16.

Digits $d_j \in \{0, 1, 2, \ldots, \beta-1, \beta\}$ where $p = \beta - 1$.

Three FIELDS:

Sign ... $\pm$

{Significand / Mantissa / Coefficient} ... $[d_1 d_2 \cdots d_{p-1} d_p]$ ... p "significant" digits of Precision

Exponent ... e, stored as $e + \text{Bias}$, an integer.

Interpreted VALUE:

$\pm [d_1 d_2 \cdots d_{p-1} d_p] \times \beta^{e_p}$ ... CDC Cyber 17x, Burroughs B65xx

Integer Coefficient

OR $\pm [\bullet d_1 d_2 \cdots d_{p-1} d_p] \times \beta^{e_0}$ ... IBM 370, DEC VAX, CDC Cyber 18x, "C" Language

OR $\pm [d_1 \bullet d_2 d_3 \cdots d_{p-1} d_p] \bullet \beta^{e_1}$ ... "Scientific" notation, Calculators, IEEE 754/854

OR --- others ----

All values are the same if exponents are correlated:

$$e_0 = e_p + p = e_1 + 1.$$

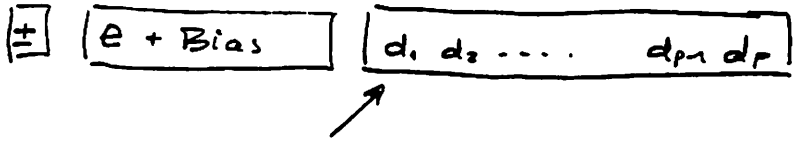
Shift of β-point without change in value $\equiv$ change in Bias.

Kahan
17 May

NON-UNIQUENESS OF REPRESENTATION

$$0.0123 = 0.123 \times 10^{-1} = 1.23 \times 10^{-2}$$

NORMALIZATION $\Rightarrow$ UNIQUE REPRESENTATION



$d_1 \neq 0$ for NORMALIZED NONZERO No's.

UNNORMALIZED NONZERO NO. has $d_1 = 0$
but $e >$ minimum.

∴ UNNORMALIZED No. is another way to represent the value of a NORMALIZED No.

NORMALIZED ZERO: $d_1 = d_2 = \cdots = d_{p-1} = d_p = 0$
& e = standard value, usually minimum.

UNNORMALIZED ZERO: $d_1 = d_2 = \cdots = d_{p-1} = d_p = 0$
but $e \neq$ standard value.

SUBNORMAL (DENORMALIZED) NO. has $d_1 = 0$
and e = minimum value.

SUBNORMAL No. is UNIQUE REPRESENTATION of its value,
so NOT UNNORMALIZED.

e.g. calculator $\pm$ d.ddd $\times$ 10 $^{\pm ee}$

0.0123×10^{0} is UNNORMALIZED
1.230×10^{-2} is NORMALIZED
0.012×10^{-99} is SUBNORMAL
0.000×10^{0} is NORMAL ZERO by convention.

WHY ARE MOST FLOATING-POINT FORMATS INTENDED TO BE NORMALIZED ?

(NOT for better accuracy !)

Because it simplifies the hardware.

e.g. ADD/SUBTRACT 3.0 + 0.012 ⇒ 3.01 to 3 sig dec

3.00×10^0	0.03×10^2	0.03×10^2
$+\ 1.20 \times 10^{-2}$	$+\ 1.20 \times 10^{-2}$	1.20×10^{-2}
3.00×10^0	0.03×10^2	0.03×10^2
0.012×10^0	0.00012×10^2	0.00012×10^2
3.012×10^0	0.03012×10^2	0.03012×10^2
⇓	⇓	⇙
3.01×10^0	0.03×10^2	3.01×10^0
	just 3. !!	⇓
As usual	in UNNORMALIZED ARITHMETIC	3.01
	IBM 370, ...	on Burroughs B6500 using double-width accumulator. (cf. 4L60L)

≠

IBM '370 Prenormalizes operands before ×, ÷.
not before ±, ... AINT(X)

DEC VAX has No UNNORMALIZED OPERANDS

IEEE 754/854 allows no UNNORMALIZED OPERANDS,
but includes SUBNORMAL OPERANDS
for GRADUAL UNDERFLOW, q.v.

EXPONENT BIAS and RANGE (A)SYMMETRY

$$x = +[d_1 . d_2 d_3 \cdots d_{p-1} d_p] \times \beta^e$$

$$e_{min} \le e \le e_{max}$$

$$X = \boxed{+}\boxed{e + Bias}\boxed{d_1 d_2 d_3 \cdots d_{p-1} d_p}$$

(Max. Normal x) · (Min. Normal Nonzero x)

$$\doteq (\beta \times \beta^{e_{max}})(1 \times \beta^{e_{min}}) = \beta^{1 + e_{max} + e_{min}}$$

If this product is too far from 1, the range is regarded as too asymmetrical, and causes troubles with expressions like

... $q := (\text{small integer})/x$ ... can over/underflow too easily?

... $q := x \cdot y / z$ ... what order avoids spurious over/underflow?

If $0 \le e + Bias \le 2^\ell - 1$ for ℓ bits

then $e_{min} = -Bias$, $e_{max} = 2^\ell - 1 - Bias$,

$1 + e_{max} + e_{min} = 2^\ell - 2 \times Bias$ .... for balance, choose Bias near $2^{\ell-1}$.

If Overflow is more to be avoided than Underflow, (e.g. if Underflow is gradual), choose $Bias = 2^{\ell-1} - 1$.

(DEC VAX: $Bias = 2^{\ell-1} + 1$ predisposes to OVERFLOW!)

RANGE ASYMMETRY is SEVERE ON SOME MACHINES.

e.g. CDC Cyber 17x :

Max. magnitude $\doteq 2^{1022+48}$

Min. normal magnitude $= 2^{-1022+47}$

(Max.)·(Min.) $\doteq 2^{95}$

$\therefore$ 2^{94} / Max. UNDERFLOWS !

Similar trouble on Burroughs B 65xx.

IEEE 754, SINGLE PRECISION :

Max. $\doteq 2^{128}$

MIN. NORMAL $\doteq 2^{-126}$

MIN. SUBNORMAL $= 2^{-149}$

$\therefore$ (Max.)·(MIN. NORMAL) $= 4$

(Max.)·(MIN. SUBNORMAL) $\doteq 2^{-21}$

$\therefore$ 2^{-20} / (MIN. SUBNORMAL) OVERFLOWS.

PROBLEM: Exhibit a program that starts from any three given positive numbers x, y, z and computes $p := x \cdot y \cdot z$ in some order that avoids undeserved over/underflow. Do likewise for $q := x \cdot y / z$.

To compute p: Sort x, y, z so that, say, $x \geq y \geq z > 0$, and then compute $p := (x \cdot z) \cdot y$.

- -

To compute q, we could sort x, y and $1/z$ except $1/z$ might over/under flow.

CAN YOU FIND A WAY ?

This is a problem that does arise in "real life" from time to time; cf. program SVD 2x2.

LEXICOGRAPHIC ORDER

String X = [±] [exponent+Bias] [$d_1\, d_2 \cdots d_{p-1}\, d_p$] ($\leftarrow l$ bits $\rightarrow$)

Floating-pt value $x = \pm [d_1 . d_2 \cdots d_{p-1} d_p] \times \beta^{\text{exponent}}$

for $0 \leq$ exponent + Bias $\leq 2^l - 1$.

Assume X is Normalized;

i.e. either $d_1 > 0$

or (exponent + Bias) = $[d_1 d_2 \cdots d_{p-1} d_p] = 0$.

Then order of x's as floating-point values

is same as order of X's as integers

with the same sign-convention

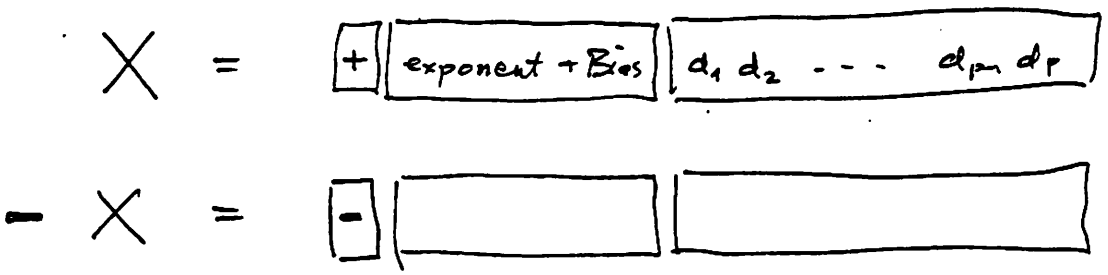
i.e. sign-magnitude, or

1's complement if $-X \equiv$ 1's compl. (X), or

2's complement $-X \equiv$ 2's compl. (X).

This can permit QUICK FLOATING-PT. COMPARISON

without floating-pt. subtraction



Sign-Magnitude ? IBM, DEC

1's-complement ? CDC Cyber 17x

2's complement ? HP, GE/Honeywell

Which is best ?

- Sign-Magnitude ≡ 1's complement, has ±0, allows ±∞.
- 2's complement has UNSIGNED 0, must have a negative value that either OVERFLOWS when negated, or is an outlaw (e.g. UNSIGNED ∞).

Sign-Magnitude is best because it allows ±0. ! Valuable for COMPLEX ARITHMETIC - CONFORMAL MAPS of SLITTED DOMAINS.

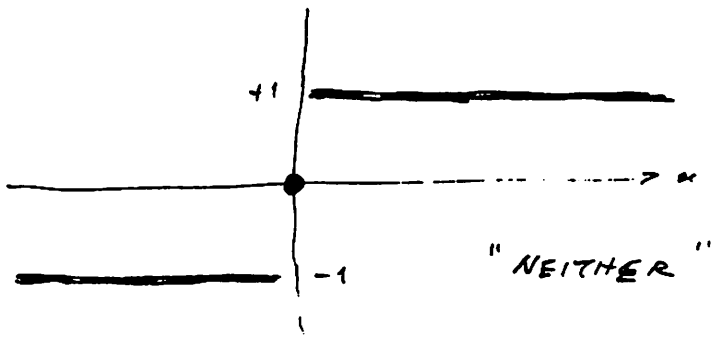
M. Powell & Iserles (ed.) (1987) "The State of the Art in Numerical Analysis" ch. 7 Oxford U. P.

$+0 = -0$

But, if $f(x)$ is DISCONTINUOUS at $x = 0$, then it may distinguish $f(+0)$ from $f(-0)$:

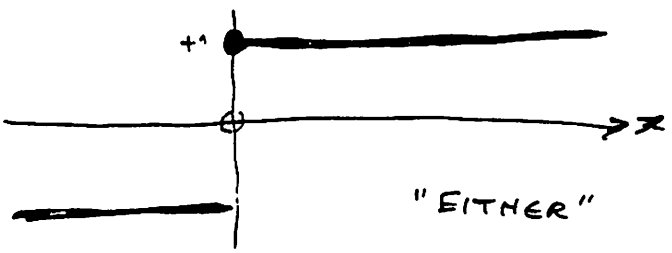
e.g.

SIGNUM (x)



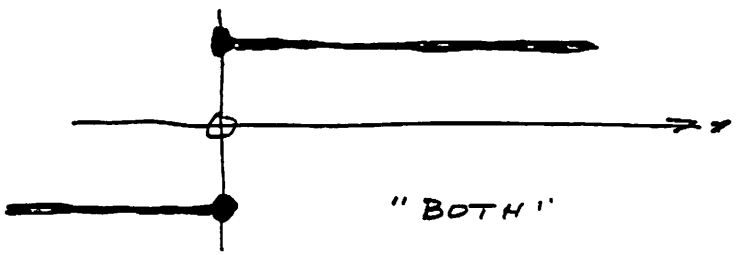
SIGNUM $(\pm 0) = \pm 0$

FORTRAN SIGN $(1.0, x)$



SIGN $(1.0, \pm 0.0) = +1.0$

IEEE COPYSIGN $(1.0, x)$



COPYSIGN $(1.0, \pm 0.0) = \pm 1.0$

$1.0 / \pm 0.0 = \pm\infty \Rightarrow 1/(1/x) \doteqdot x.$

PRECISION = RESOLUTION

(not ACCURACY)

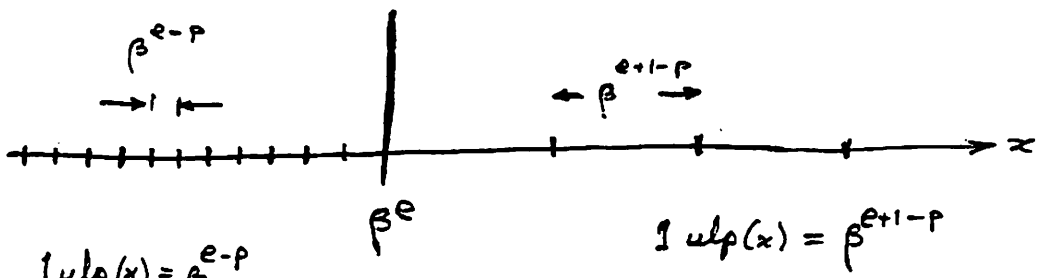
Relative Precision at x : $\dfrac{1\ ulp\ (x)}{|x|}$

where, if $x = \pm [d_1 . d_2 \cdots d_{p-1} d_p] \times \beta^e$

then $ulp(x) = [0.0 \cdots 0\ \ 1] \times \beta^e$

i.e. if $\beta^e < x < \beta^{e+1}$ then $ulp(x) = \beta^{e+1-p}$

but $ulp(\beta^e) = \beta^{e+1-p}$ or β^{e-p} ambiguously

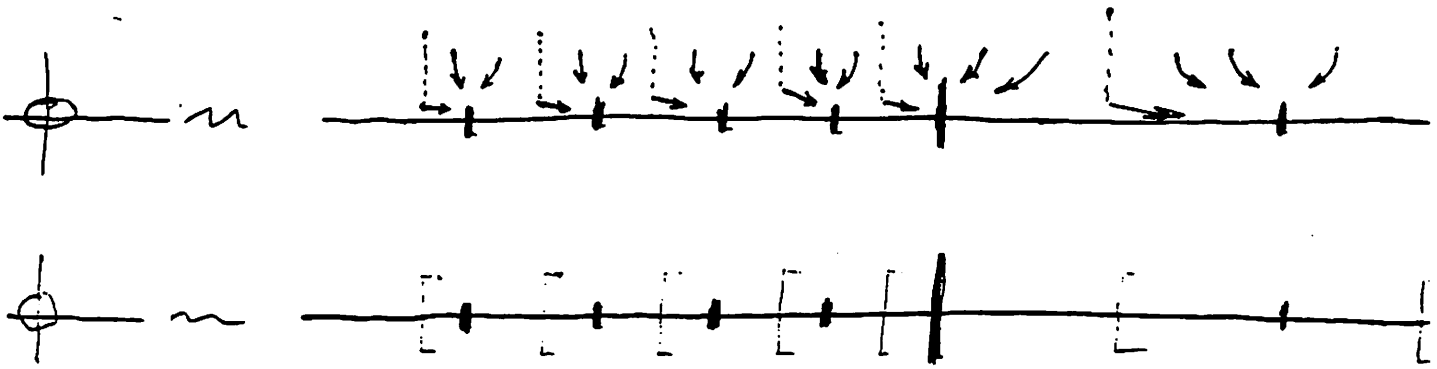


$$\therefore \quad \beta^{-p} \leq \frac{1\ ulp(x)}{|x|} \leq \beta^{1-p}$$

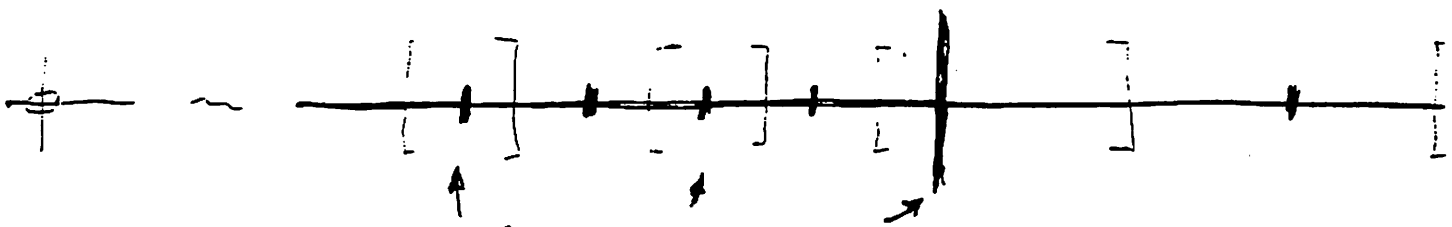
"WOBBLE" by factor β

"CORRECT" ROUNDING OF MIDWAY CASES

DEC VAX rounds MIDWAY CASES away from zero, i.e. UP in magnitude:



IEEE 754/854 rounds MIDWAY CASES to "NEAREST EVEN":



Least-sig.-bit is zero.

e.g. to 4 sig.dec.

IEEE 854 (HP-71B)	3.142 ←	3.1415	→ 3.142	like DEC VAX HP-15C
	3.140 ←	3.1405	→ 3.141	
	3.140 ←	3.1395	→ 3.140	
	3.138 ←	3.1385	→ 3.139	
	3.138 ←	3.1375	→ 3.138	

Error in floating-point operations

Let $\circledast \in \{ +, -, \times, / \}$

Let A be name of variable whose value is a,
B b,
C c.

Program statement "$A := B \circledast C$"

produces value $a = b \circledast c$ ROUNDED

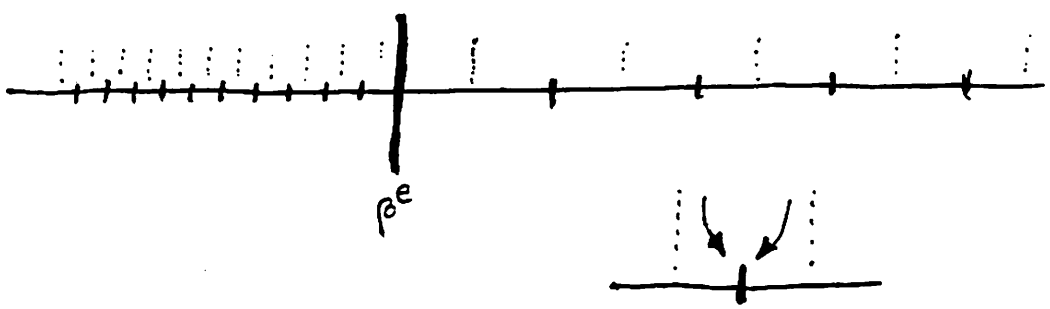
$$b \circledast c = \pm x.xxx\cdots xxx\ xxxx\cdots$$

$$a = \pm x.xxx\cdots xxx$$

$\longleftarrow p \longrightarrow$

"CORRECT" ROUNDING $|a - b \circledast c| \leq \frac{1}{2}\ \text{ulp}(b \circledast c)$

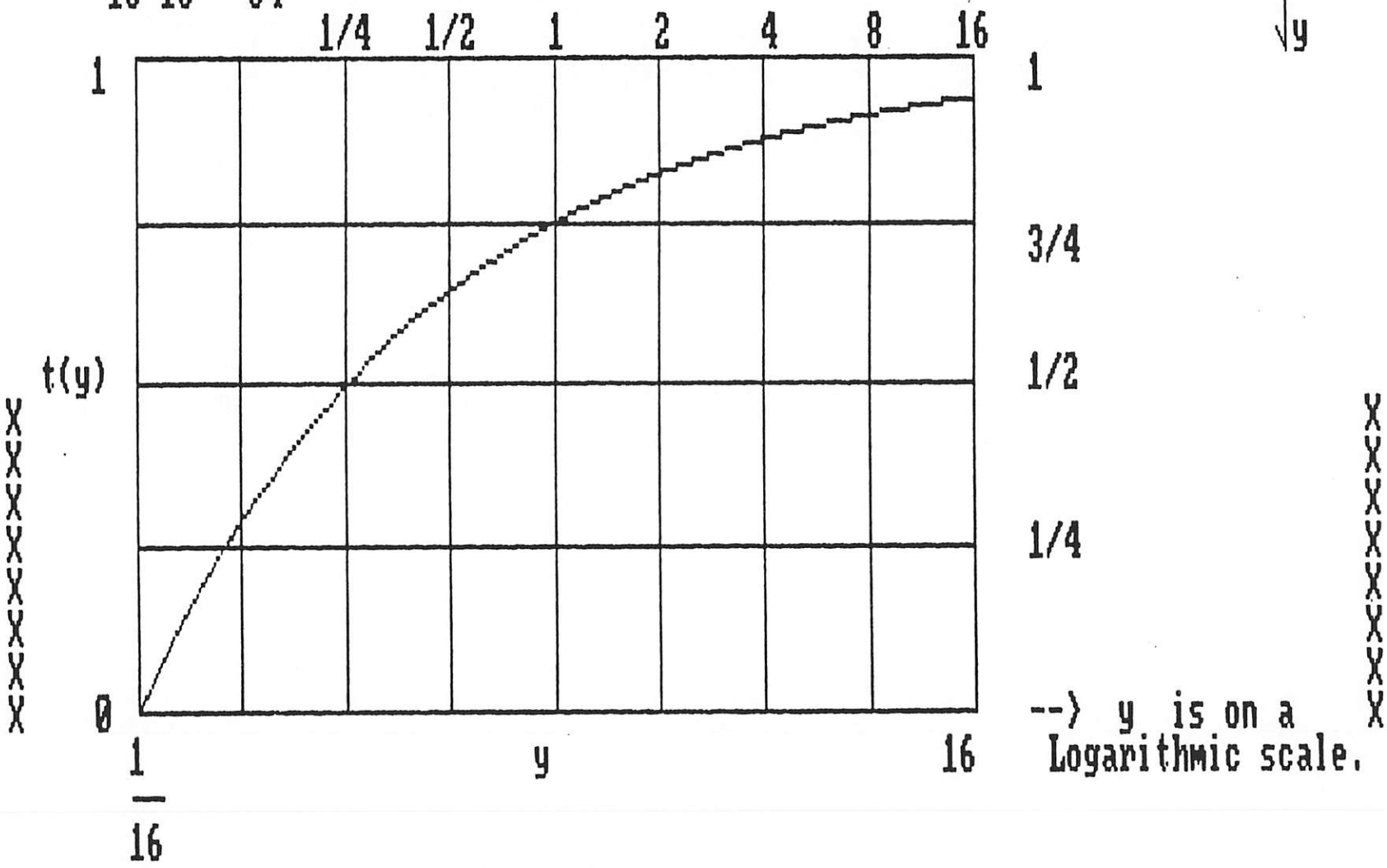
except $\begin{cases} a = b \circledast c & \text{if it is representable exactly,} \\ a = \text{something else} & \text{if over/underflow occurs.} \end{cases}$



What if $b \circledast c$ falls midway between two adjacent representable values?

$$y := \frac{1}{16}, \frac{1}{16} + \frac{1}{64} .. 16$$

$$t(y) := 1 - \frac{0.25}{\sqrt{y}}$$



--> y is on a Logarithmic scale.