Software $\sqrt{x}$ for the Proposed IEEE Floating-Point Standard

W. Kahan

University of California at Berkeley

August 25, 1980

Introduction:

Whether every implementor of the proposed standard must provide $\sqrt{x}$ correctly rounded has been a controversial question. There is a large market for inexpensive fast implementations designed for computations in which $\sqrt{x}$ never figures; why should these implementations be burdened by an unnecessary unwanted expensive feature? They shouldn't.

Part of the controversy arises because the word "implementation" is frequently misinterpreted as "hardware". Actually hardware and software may and usually must be used together to implement the standard fully, and the implementor is free to draw the boundary between the two as best suits his goals. For instance, he may design a chip to add, subtract, multiply and divide very fast, but omit square root to keep his chip's area (and hence cost) as low as possible. Provided he offers appropriate software with the chip, it can conform to the proposed standard in all respects. Of course, the software square root may well be slower than one implemented in hardware on the chip, but that is an aspect of the cost-performance trade-off that reflects the perceptions of the marketplace. To diminish the loss of speed, the chip's designer should include on-chip whatever extra features will promote faster software at negligible additional cost when the chip is designed. Paradoxically, a chip that includes those few extra features will cost the consumer less than a chip that lacks them, even if most such features are superfluous to most consumers' needs, provided each added feature enlarges the chip's market in a way that, by increasing the volume of production, cuts the chip's price below that of a simpler chip produced in smaller numbers. Customers who believe initially that $\sqrt{x}$ is superfluous will none the less prefer a chip that can calculate $\sqrt{x}$ quickly via software to one that cannot, other things being roughly equal; no prudent purchaser will disregard the value of an option which could rescue him when initial beliefs turn out later to be wrong.

The following algorithm provides just one of many ways to implement $\sqrt{x}$ in software. It supplies $\sqrt{x}$ correctly rounded at an acceptable cost provided division is fast, since the program involves three divisions. Also needed are the ability to chop sums and quotients instead of round them, and the INEXACT flag to indicate when a quotient leaves a remainder, all part of the standard. The ability to perform shift, add, subtract and logical OR operations upon 32-bit words is needed too, though not part of the standard.

$\sqrt{x}$ algorithm for the proposed IEEE floating-point standard

What follows is an algorithm to calculate $\sqrt{x}$ correctly rounded to SINGLE-precision using rational SINGLE-precision floating-point arithmetic and some fixed-point operations on 32-bit words. Included is a description of how the algorithm was tested.

The format for a normal SINGLE-precision floating-point number is:

$$x = (-1)^{\sigma} \cdot 2^{E-127} \cdot (1.F) \sim$$

implicit bit ↑

sign bit | Biased Exponent | Significand Fraction

provided $1 \leq E \leq 254$. The maximal exponent $E=255$ is reserved for

either $x = \pm\infty$ $(E=255, F=0)$
or $x = NaN$ (Not-a-Number: E = 255, F > 0).

The minimal exponent $E=0$ is used to encode denormalized numbers

$x = (-1)^{\sigma} \cdot 2^{1-127} \cdot (0.F)$ ...note implicit bit in exponent
or $x = \pm 0$ when $E=F=0$.

The algorithm below separates both special exponents $E=255$ and $E=0$ from the others in Step #0 which, for simplicity's sake, presumes that no traps exist.

We shall identify the floating-point number x with a fixed-point number X that is the value of the 32-bit word x above when interpreted as an unsigned integer $X = 2^{31} \cdot \sigma + (E.F) \cdot 2^{23}$. Similarly for y and Y, z and Z; arithmetic upon lower case x, y, z is SINGLE-precision floating-point but upon upper case X, Y, Z we perform integer arithmetic or logical operations.

$\sqrt{x}$ algorithm

Input:

32-bit word x ~

Current rounding mode: $R =$
- 00 for round-toward - zero
- 01 for round-toward - $-\infty$
- 10 for round-to-nearest
- 11 for round-toward - $+\infty$.

Current infinity mode: $w =$
- 0 for projective mode
- 1 for affine mode

Current underflow mode: $u = \begin{cases} 0 \text{ for warning (not normalizing) mode} \\ 1 \text{ for normalizing mode} \end{cases}$

Current INEXACT flag: I = *FALSE or TRUE*

Step 0 Filter out special operands:

If E=255 then (if F = 0& (w=0 or σ=1) then "Invalid" ...unsigned or negative ∞
else "Unchanged" ... + ∞ or *NaN*.)

If E = 0 then (if F = 0 then "Unchanged" ...±0
else if u=0 or σ=1 then "Invalid" ...denormalized warning
or negative x.
else "Normalize" ...positive
denormalized x.)
else if σ=1 then "Invalid" ... negative x.
else "Operate" ... calculate $\sqrt{x}$.

Exits:

"Invalid": return $\sqrt{x}$:= *NaN* created for the occasion, and raise INVALID flag
"Unchanged": return $\sqrt{x} := x$ ... ±0 or +∞ or *NaN*.
"Normalize":
$Y := 193 \cdot 2^{23}$ ... | 0 | 193 | 0 | ... a constant
$X := X + Y$ ... | 0 | 193 | F |
$y := \sqrt{x} - y$ via "Operate" called as a subroutine.
$Z := Y - 96 \cdot 2^{23}$ ... $y / 2^{96}$
return $\sqrt{x} := z$ end "Normalize"

"Operate": continue to step 1 with finite normal positive x.

Step 1 Save, reset and initialize:

$i := I$ ... save INEXACT flag I.
$r := R$; $R := 0$ ... save rounding mode and reset to round-toward-zero.
$Y := \lfloor X/2 \rfloor + (127 \cdot 2^{22} - 320000)$ ... drop bit shifted off X.
... now magically y approximates $\sqrt{x}$ to almost 5 sig. bits.

Step 2 Heron's rule twice:

$z := y + x/y$ ... in chopped floating-point arithmetic
$Z := Z - (2^{23} + 3150)$ ... z approximates $\sqrt{x}$ to over 11 sig. bits.
$y := z + x/z$ ... in chopped floating-point arithmetic
$Y := Y - 2^{23}$ ... y approximates $\sqrt{x}$ to within 1 ulp.
... Most $\sqrt{x}$ programs do no better than this, but we must persevere to get $\sqrt{x}$ correctly rounded and, if r = 10 (round-to-nearest), accurate within 1/2 ulp.

Step 3 Twiddle last bit using integer add and shift:

I := *FALSE* ... reset INEXACT flag I.
$z := x/y$ ... chopped quotient, possibly inexact.
If not I then (if z = y then skip to Step 4 ... $\sqrt{x}$ is exact
else $Y := Y-1$ ... special rounding)
i := *TRUE* ... $\sqrt{x}$ is INEXACT.
If $r > 01$ then($Y := Y + 1$ ... round-to-nearest
if $r = 11$ then $Z := Z + 1$ ... round-towards $+\infty$)
$Y := \lfloor (Y + Z)/2 \rfloor$ ... logical right shift has left-most bit 0.

Step 4

$R := r$; $I := i$ ...restore rounding mode and INEXACT flag
return $\sqrt{x} := y$ end.

Alternative to Step 3

Step 3 Twiddle last bit using floating add:

I := *FALSE* ... reset INEXACT flag I.
$z := x/y$ ... chopped quotient, possibly inexact
If not I then (if $z = y$ then skip to Step 4 ... $\sqrt{x}$ is exact
else $Z := Z-1$ ... special rounding)
i := *TRUE* ... $\sqrt{x}$ is inexact.
If $r > 01$ then ($Z := Z + 1$ ... round-to-nearest
If $r = 11$ then $Y := Y + 1$ ... round-toward + ∞.)
$y := y + z$ ... chopped sum
$Y := Y - 2^{23}$ ... $y := (y + z)/2$ is correctly rounded.

Although stated in terms of SINGLE precision, Step 3 is the way to produce $y = \sqrt{x}$ correctly rounded to DOUBLE, or any other working precision, provided the previous approximation y differs from $\sqrt{x}$ by less than 1 ulp (one unit in the last place carried) of the desired precision. The proof that Step 3 works correctly is an exercise in elementary inequalities with integers.

Testing

The DEC PDP-11 and VAX single-precision floating-point formats so closely resemble the proposed IEEE format that the foregoing algorithm could be tested on those machines. However certain significant differences demand attention:

i) DEC's exponent bias is 129 instead of 127 for normal numbers; therefore the constant $(127 \cdot 2^{22} - 320000)$ in Step 1 must be replaced by $(129 \cdot 2^{22} - 320000)$.

ii) DEC reserves only the exponent $E = 0$ for $x = 0$ or *NaN*, omitting denormalized numbers and -0; also $E = 255$ is a normal exponent so there is no ∞. Hence Step 0 must be shortened by omission of the statements that test for $E = 255$ or $u = 0$.

iii) The VAX swaps the first and last 16 bits of floating-point numbers when they are loaded into registers, so a compensatory swap must be inserted before and after integer and logical operations that affect the F-field. Also, the VAX lacks the PDP-11's chopped floating-point arithmetic, so the foregoing algorithm's floating-point operations must be effected in double-precision and then chopped back to single, thereby squandering the algorithm's speed.

iv) The DEC machines lack the INEXACT flag I used in Step 3 to decide how to increment Y and Z; instead the division $z = x/y$ must be carried out to double-precision, 56 sig. bits, using the first 24 for z and the last 32 to indicate, if not all zero, INEXACT.

Despite these differences we have tested the algorithm by running it on a VAX which happened to be available. The test compared SQRT(x):= $\sqrt{x}$ calculated by the algorithms above with a correctly rounded $\sqrt{x}$ obtained from a double-precision (56 sig. bit) calculation of $\sqrt{x}$ by rounding off its last 32 bits in accordance with the specified rounding mode R. The values chosen for x were all $2^{24} = 16777216$ consecutive arguments x between $x = 1$ and $x = 4 - 2^{-22}$ inclusive, plus the set of squares $x = 1, 4, 9, 16, 25, \ldots, 4095^2, 4096^2$. The test also checked that the inexact flag I correctly indicated whether or not $\sqrt{x}$ was exact. The test was programmed in *C* by Miss Heidi Stettner. No discrepancies were found. To test the test, it was re-run on the fast SINGLE-precision SQRT(x) program that we normally use on the VAX, and all instances were found where this program's error exceeds 1/2 ulp. [One ulp is one unit in the last (24th) place.]

General Comments

The foregoing algorithm was derived from one I developed in late 1962 for the IBM 7090 at the University of Toronto, and used subsequently for an IBM 7094, a CDC 6400, a DEC PDP-10 and our present VAX at the University of California at Berkeley. A similar but slightly slower algorithm was devised independently by Hirondo Kuki in 1963 for the IBM 7094 at the University of Chicago and used also on the IBM 7040. The algorithm is economical only when division is fast.

When division is slow but multiplication is fast, a better idea is to adapt the "Reciproot" algorithm that ran on the Ferranti-Manchester Mk. I perhaps as early as 1949:

Given $x > 0$ choose any $u_0 > 0$ and for $n = 0, 1, 2, 3, \cdots$ set $u_{n+1} := \max\left(u_n, \frac{3}{2}u_n - \frac{1}{2}x u_n^3\right)$ until u_n converges to $1/\sqrt{x}$ (which it does quadratically); finally $\sqrt{x} := (1/\sqrt{x}) \cdot x$.

When multiplication and division are both slow, $\sqrt{x}$ may be best calculated digit-by-digit using the "formal method" sometimes taught in high schools. This method requires either a remainder or two extra bits of $\sqrt{x}$ to determine correctly the round-bit and sticky-bit needed to round $\sqrt{x}$ and set the INEXACT flag as required by the standard. In fact, after calculating the first 26 sig. bits of $\sqrt{x}$ we must conclude that all subsequent bits are zeros just when the last 14 bits of the first 26 are all zeros. That no such statement could be true if "26" were reduced to "25" follows from the square root of 1.000...01. But on most

current machines an algorithm that exploits the remainder is probably best; such an algorithm, provided by Mr. George Taylor, is appended below. It too has been programmed for our VAX and tested as described above.

Acknowledgements:

Work reported herein was supported in part by the U. S. Department of Energy, Contract DE-AT03-76SF00034, Project Agreement DE-AS03-79ER10358, and by the Office of Naval Research, Contract N00014-76-C-0013.

Annotated Bibliography

Concerning the Proposed IEEE Floating-point Standard:

J. Coonen (1980) "An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic" in "Computer", Vol. 13, No. 1 (Jan. 1980), pp. 68-79. Published by the IEEE.

J. Coonen, W. Kahan, J. Palmer, T. Pittman, D. Stevenson (1979). "A Proposed Standard for Binary Floating Point Arithmetic", Draft 5.11 in *ACM SIGNUM Newsletter Special Issue*, (Oct. 1979), pp. 4-12.

W. Kahan, J. Palmer (1979). "On a Proposed Floating-Point Standard". Ibid. pp. 13-21.

Concerning conventional square root software:

W. J. Cody, W. M. Waite (1980). "A Software Manual for the Elementary Functions", ch. 4. Published by Prentice-Hall, N.J.

M. Andrews, S. F. McCormick, and G. D. Taylor (1979) "Evaluation of functions on microcomputers: square roots" in *Computer Math., Appl. vol. 4, pp. 359-367.*

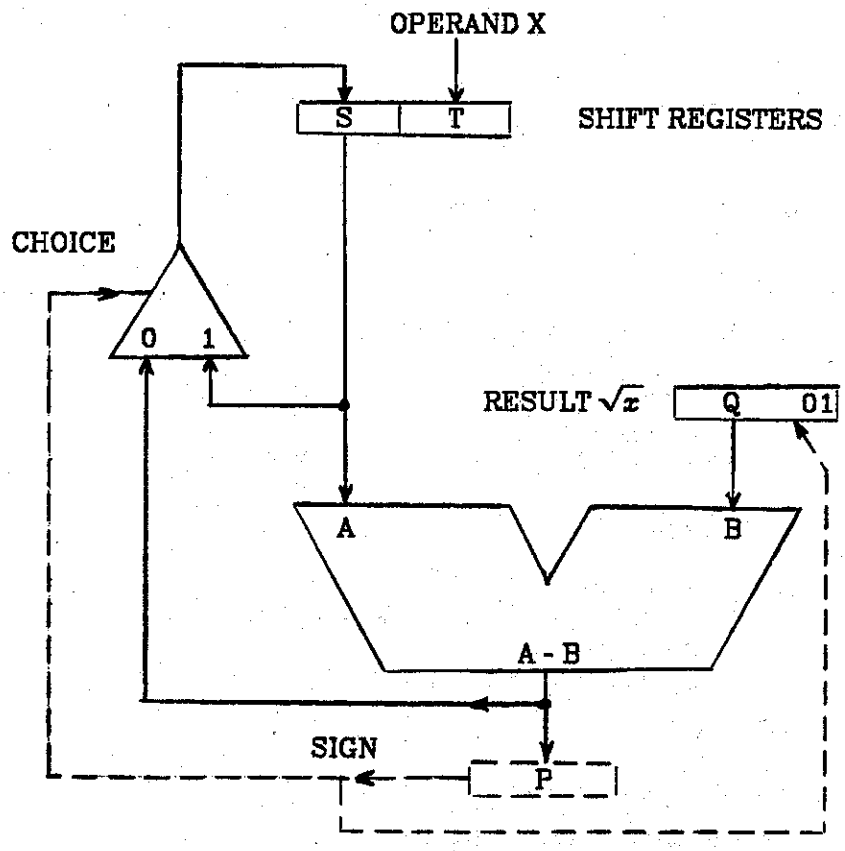
Concerning digit-by-digit square root:

K. Hwang (1979). "Computer Arithmetic", pp. 360-411, and references cited therein. Published by Wiley, N.Y.

Appendix

George Taylor's bit-by-bit square root algorithm:

This simulates a hardware design in software. The hardware would look like this:



But the program below uses 32-bit registers for all variables except the rounding mode *R*, the inexact flag *I*, and the boolean variable Roundbit. The hardware would use registers which need to be no wider than the desired result's unpacked fraction plus 3 bits.

The second loop in Step 2 shifts the S register rather than the (S,T) register pair. The last iteration of the second loop as written requires a 25 bit subtraction. Special coding for this step can reduce the width to 24 bits, which is the length of the result's unpacked fraction.

Step 0 Filter out special operands as is done above to ensure that x is positive, normalized and finite. $1.0 \leq fraction\ of\ x < 2.0$

Step 1 Initialize

$Z := X$ AND $0080\ 0000_{16}$ ... extract last bit of exponent
$Y := \lfloor X/2 \rfloor$ AND $7F80\ 0000_{16}$ ... retain half of biased exponent

if (Z = 0) then {
$Y := Y + 1F80\ 0000_{16}$ ... unbiased exponent is odd, so add 63
$T := ((2 \cdot X)$ OR $0100\ 0000_{16})$ AND $01FF\ FFFF_{16}$ }
else {
$Y := Y + 2000\ 0000_{16}$ ... unbiased exponent is even, so add 64
$T := X$ AND $00FF\ FFFF_{16}$}

$S := 0$
shift (S,T) left 9

$S := S - 1$
$Q := 5$ ... first bit of result must be "1"

Step 2 Inner loop

repeat 12 times (26 for double, 32 for extended):
shift (S,T) left 2
$P := S - Q$
$Q := 2 \cdot Q - 1$
if ($P \geq 0$) then $S := P$; $Q := Q + 4$

$Q := \lfloor Q/4 \rfloor$

repeat 12 times (27 for double, 32 for extended):
$P := S - Q - 1$ ... one's complement subtraction
$Q := 2 \cdot Q$
if($P \geq 0$) then $S := 4 \cdot P + 3$; $Q := Q + 1$
else $S := 4 \cdot S$

Step 3 Round and set the Inexact flag as appropriate

Roundbit := Q AND 00000001_{16}
$Q := \lfloor Q / 2 \rfloor$
if (*Roundbit* or ($S > 0$)) then
{ *I* := *TRUE* ... set the Inexact flag
if ($R = 11$) then $Q := Q + 1$ ... round to $+\infty$

if (*Roundbit* and ($R = 10$)) then $Q := Q + 1$ ... round to nearest

Step 4 Pack result

$Y := Y$ OR (Q AND $007F\ FFFF_{16}$)
return $\sqrt{x} = y$