Doubled-Precision IEEE Standard 754 Floating-Point Arithmetic

Prof. W. Kahan
Elect. Eng. & Computer Science
University of California
Berkeley, Ca 94720

Introduction:

Let "WP" stand for "Working Precision," which denotes
whichever is the widest floating-point format supported by some
computer that conforms to the IEEE standard 754-1985 for binary
floating-point arithmetic. (That standard specifies Single,
Double and Extended formats, but a conforming computer need not
support all of them.) Let "DP" stand for "Doubled-Precision,"
which denotes a simulation of arithmetic accurate to about twice
WP though carried out using nothing but standard WP arithmetic.

How well can such a simulation be carried out? The discussion
that follows addresses that question, but does not settle it.
Because this paper is intended to stimulate rather than stifle
interest in the question, proofs have been omitted; many of them
can be found in references cited in the Bibliography.

Why does it matter?

Many scientific and engineering calculations, particularly those
that involve either massive destructive cancellation or else vast
numbers of small corrections to undulating variables, would be
swamped by the effects of roundoff but for the use of some kind of
extra-precise arithmetic. Programming those calculations might be
simpler if higher precision were available merely for the asking,
but the exigencies of hardware or language implementation impose a
limit, usually small, upon the precision that can be programmed
conveniently and efficiently in higher-level languages. That is
why tricks have to be contrived, sometimes unconsciously, to
achieve the effect of higher precision without calling upon it
explicitly. Those tricks are generally confined to small but
crucial parts of a program, where they lurk amongst manipulations
of WP variables that seem entirely unexceptionable to the naked
eye. In some situations where WP variables would have to be
mixed with higher-precision variables if they were available, the
tricks, by avoiding both format conversions (between WP, and
higher-precision formats) and calls to subroutines that simulate
higher-precision arithmetic, run faster.

Unfortunately, tricky programs are dangerous, especially when
they simulate higher-precision floating-point arithmetic. Though
they seem to be portable, recompiling and running them on any
machine other than the one for which they were designed runs the
risk of malfunctions that, however rarely they may occur, are
bound to be mysterious. The epithet "pornographic" has been
attached to these tricky programs, but undeservedly; they
generally do possess some redeeming social merit.

Understanding these tricks is worth our while for two reasons.
First, if we have to use them, we shall have to know how and
when they work. Second, a program that originated elsewhere than

on the machine to which we wish now to adapt it may contain such
tricks, and we have to recognize them before we can deal with
them. The tricks are hard to recognize because, to achieve
ostensible portability. they perform only standard floating-
point operations upon WP variables. No Fortran EQUIVALENCE
statements matching INTEGER with REAL variables; no REAL
actual arguments for subprograms with INTEGER formal arguments:
none of Pascal's deceitful VARIANT RECORDs; in short, none of
the bit-twiddling that obviously obviates portability.

The games we play with floating-point might as well be played upon
a machine that conforms to the IEEE standard 754-1985 for binary
floating-point arithmetic. If we have to simulate DP in our
program, it will be simpler on that machine than upon almost any
other. And extra complications introduced to simulate DP on
diverse other machines will almost certainly continue to work on
that one. Let's avoid introducing those extra complications into
a subject that can already be characterized as the exaltation of
complicated ways to calculate what would be zero in the absence
of roundoff.

Notation:
First some notation. Upper-case letters S, T, X, Y, Z, ... will
denote DP numbers, each represented in the computer by a pair
of WP numbers denoted by corresponding lower-case and Greek
letters. For instance, the DP variable X is represented in
the computer by x and ξ ; in fact, $X = x+\xi$ exactly in so far as
that addition is construed not as WP arithmetic but infinitely
accurate. To help avoid confusion about the accuracy to which
various arithmetic operations are carried out, further notation
is required. We shall use brackets "[...]" to denote rounding
to WP in accordance with the IEEE standard 754; this means
rounding-to-nearest, with half-way cases rounded to nearest even.
(See the draft of the standard for a more explicit explanation.)
In particular, let ε be the difference between 1 and the
nearest other WP number $1-\varepsilon$; then $[1+\varepsilon] = 1$ according to
IEEE 754, whereas $1+\varepsilon$ would round to $1+2\varepsilon$ on a DEC VAX .

We shall also adopt a convention for assignment statements, in
which the symbol " := " appears: if such an assignment contains
only WP variables, then every arithmetic operation will be
presumed to be rounded to WP ; but if any DP variable appears
on either the left- or right-hand side of the assignment, then
the arithmetic will be presumed to be carried out in DP . For
instance, the assignment statement ... "ζ := (x-z) + y " implies
$\zeta = [[x-z]+y]$; but " T := (x-z) + y " implies that the DP
result T is obtained from a subtraction x-z carried out in DP
arithmetic, followed by addition of y in DP . Each such DP
operation is the result of a suitable sequence of WP operations.

DP sum of WP variables, Z := x+y :
Every DP variable Z shall be represented as an unevaluated sum
$z+\zeta$ of a pair of WP variables; normally they will be further
constrained by the requirement that $z = [z+\zeta] = [Z]$. This state
of affairs can be brought about by a procedure that implements the

assignment   " Z := x+y "   for any two WP variables x and y :
        If |x| < |y| then swap(x, y) : ... so |x| ≥ |y|
        z := x + y ;   ... rounded to WP
        ç := (x-z) + y ;   ... turns out to be exact!
    ... Now, ideally, Z = z+ç should equal x+y exactly.

*Note: Do not omit parentheses from this procedure, nor from any
presented later. Note too that this procedure performs none but
standard WP floating-point operations upon WP variables.*

Provided z does not overflow beyond the range encompassed by WP
numbers, this simple procedure can be proved to work infallibly
for all finite WP variables x and y . But the proof cannot be
simple, because the procedure fails occasionally on arithmetics
that do not conform to IEEE 754 though they may be exemplary in
almost every other respect. Failure will occur when z or x-z
underflows unless underflow is *Gradual*, as it is on standard-
conforming machines but almost no others. Except for underflow,
this procedure works on all computers with binary floating-point
arithmetic rounded in a reasonable way; examples are DEC VAXs,
and the IBM PC's compiled BASIC but not its BASIC interpreter.
Failure can occur often on machines that subtract without carrying
a guard digit; among such machines are all CRAYs and all CDC
CYBERs, many UNIVACs, most TI calculators and most credit-card
calculators. Except for underflow, the procedure works also on
IBM /370's and other machines (AMDAHL, ...) with identical
hexadecimal arithmetic, but otherwise very rare failures will
occur on non-binary arithmetics whose addition with like signs is
rounded instead of chopped. For example, take x = 9999.9 and
y = 9999.7 on a machine that rounds to five significant decimals,
whereupon z = [19999.6] = 20000 and [x-z] = [-10000.1] = -10000
is not exact, so ç = -0.3 instead of the correct -0.4 .

Evidently the assignment   " Z := x+y "   can be implemented in a
simpler way on computers that conform to IEEE 754 than on almost
any others. That is significant because this assignment realizes
the crucial special case (N = 2) out of which emerges a more
general summation procedure upon which all arithmetic operations
upon floating-point operands of arbitrarily high precision (not
merely DP) can be based, at least in principal. The following
digression will explain.

Distillation of a Sum of N WP numbers.
For any given WP numbers $x_1, x_2, ..., x_N$, there must exist
another set of WP numbers $x'_1, x'_2, ..., x'_{N'}$ that satisfy both
    $x'_1 + x'_2 + ... + x'_{N'}$ = $x_1 + x_2 + ... + x_N$ exactly, and,
    for $0 < j < N' \leq N$, $x'_{j+1} = [x_j + x'_{j+1}]$ rounded to WP ,
provided no sum overflows. When N' > 2 the numbers {x_j} do
not necessarily determine the numbers {x'_j} uniquely, as the
following examples illustrate for binary floating-point arithmetic
rounded to 4 sig. bits the same way as IEEE 754 rounds to at
least 24 sig. bits:
        1000000000000 - 1000000 - 11 = 1111000000000 + 111100 + 1.
             1000000000 - 1000o - 1 = 1111oooo0 + 1111.
(The second example, which shows that N' is not determined uniquely, does not work with rounding like
   a DEC VAX's ; for that kind of rounding try instead
        1001000000000000 - 1000000000 - 100o + 1 = 10000000000000000 + 111100000 + 1111 .)

Despite a little ambiguity in the numbers $\{x_j\}$, calculating them is a plausible way to approximate the sum $\Sigma_j \, x_j$ by a DP approximation $x_{N'} + x_{N'-1}$, or more generally by an M-tuple precision approximation $x_{N'} + x_{N'-1} + \ldots + x_{N'-M+1}$. Neither approximation is "correctly rounded" in any worthwhile sense, but their respective relative errors are smaller than ϵ^2 and ϵ^M , where ϵ was defined under "Notation" above as the best bound for roundoff error in one WP operation. (When $M > N'$ the M-tuple precision approximation is actually perfect subject to the understanding that $x_j := 0$ when $j < 1$.)

Schemes to calculate the numbers $\{x_j\}$ using only WP operations have been known for about two decades. They work by iterating DP sums of two WP variables, " $Z := x+y$," in ways so similar to the way a refinery produces gasoline or alcohol that they deserve to be called *Distillation* schemes. The fastest one I know is presented nearby. All such schemes I know that have been proved correct work usually rather faster than has been proved, yet much slower than we would like; their average running times appear to grow at least as fast as $N \log(N)$ as $N \to \infty$. Asymptotically faster schemes, running in times that grow just linearly with N , exploit integer arithmetic upon the constituent *exponent* and *significand* fields into which WP numbers can be decomposed by, for instance, the *frexp* function in the standard C math. library. In principle, standard WP arithmetic operations are capable of accomplishing that decomposition; but, no matter how it be accomplished, it costs far too much time to be attractive for the values N that figure in DP arithmetic. $N = 4$ for DP addition " $Z := X+Y$." For DP multiplication " $Z := X*Y$," N can lie between 6 and 16 . We seek adequately accurate schemes that are faster than distillation or decomposition for values N like these. The next paragraph shows what to do when $N = 3$.

Add/Subtract DP + WP , $\quad Z := X+y$:
Since $X-y = X+(-y)$ and $y-X = (-X)+y$, any implementation of addition must encompass subtraction too, so let us concentrate upon addition. The implementation of DP := DP+WP is slightly more complicated than the implementation of DP := WP+WP above. X and y are assumed given; this means that x, ξ and y are given, and that $X = x+\xi$ with $x = [X] = [x+\xi]$. We seek to compute $Z = z+\zeta \doteq X+y$ so accurately that $|Z-(X+y)| \leq \epsilon^2 |X+y|$. Here is a way to do it; *but do not omit parentheses*:

$$\text{If} \quad |y| > |x| \quad \text{then} \quad \text{swap}(x, y)$$
$$\text{else if} \quad |y| < |\xi| \quad \text{then} \quad \text{swap}(y, \xi) \; ;$$
$$\ldots \qquad \text{Now} \quad |x| \geq |y| \geq |\xi| \; .$$

$t := (\xi+y) + x$;	... computed in WP
$\tau := ((x-t) + y) + \xi$;	... in WP
$z := t+\tau$;	... to WP
$\zeta := (z-t) + \tau$;	... exact in WP

$\ldots$ Now $\quad Z = z+\zeta \doteq X+y$.

Proving this procedure's correctness is a challenging task. The crucial step is proving $(x-t)$ exact. The proof I know works for arithmetic that conforms to IEEE 754 ; I do not know how this procedure (mis)behaves when some other kind of arithmetic is used.

==

A Distillation Program:

Given $N > 0$ and $x_1, x_2, \ldots, x_N$, we wish to replace them by
N' and $x_1', x_2', \ldots, x_{N'}'$ in such a way as nearly minimizes N' yet
keeps $x_1' + x_2' + \ldots + x_{N'}' = x_1 + x_2 + \ldots + x_N$ exactly, using
only standard WP operations upon WP variables. We shall use a
DP addition " $Z := p+q$ " of WP variables p and q to provide
$Z = z+\zeta = p+q$ exactly while $z = [z+\zeta]$ rounded to WP ; this can
be realized using only WP arithmetic as described earlier.

The first step is to sort $\{x_j\}$ by magnitude to ensure that
$0 < |x_1| \leq |x_2| \leq \ldots \leq |x_N|$. The sorting procedure used here
should depend upon the provenance of the data; if $\{x_j\}$ is in no
special order then a Heap-sort is appropriate; otherwise a
Bubble-sort or Merge-sort may run faster. Omitting the sort
altogether can do no worse than slow down distillation.

The Distillation process proper consists of repeated summations
alternating in direction from small-to-big, then big-to-small.
Each summation leaves $\Sigma_j \, x_j$ unchanged. Each summation starts
with $x_1 + x_2 + \ldots + x_{kl}$ and $x_{kr} + \ldots + x_{N-1} + x_N$ already
distilled, and tracks changes in kl and kr . Here is the code:

```
Sort {xⱼ} so that  0 < |x₁| ≤ |x₂| ≤ ... ≤ |xₙ| ; ... omit zeros
kl := 1 ;  kr := N ;
while  kl < kr  do {                          ... Forward pass
    j := kl ;  p := xⱼ ;  k := 0 ;
    for  i = j+1 to N  do {
        if  i>kr & j-1>k  then { xⱼ := p ;  j := j+1 ;  p := xᵢ }
        else { q := xᵢ ;
               Z := p+q exactly; ... = z+ʃ  with  z = [z+ʃ]
               if  ʃ = 0  then  k := j
               else { xⱼ := ʃ ;  if  z ≠ q  then  k := j ;
                      j := j+1 ;  if  k = 0  then  kl := j } ;
               p := z } };
    kr := k ;  if  kl > 1 then  kl := kl-1 ;
    if  p = 0  then  N := j-1  else { xⱼ := p ;  N := j } ;
    if  kl < kr   then {                      ... Backward pass
        j := kr ;  p := xⱼ ;  k := N+1 ;
        for i = j-1 to 1 step -1 do {
            if i<kl & j+1<k then { xⱼ := p ;  j := j-1 ;  p := xᵢ }
            else { q := xᵢ ;
                   Z := p+q exactly; ... = z+ʃ  with  z = [z+ʃ]
                   if  ʃ ≠ q  then  k := j ;
                   if  ʃ = 0  then  p := z
                   else { xⱼ := z ;  j := j-1 ;  p := ʃ ;
                          if  k > N  then  kr := j } } } ;
        kl := k ;  if  kr < N then kr := kr+1 ;
        if p = 0 then  j: = j+1  else  xⱼ: = p ;
        if  j > 1 then { j:=j-1 ;  N:=N-j ;  kl:=kl-j ;  kr:=kr-j ;
                         for i = 1 to N  do  xᵢ := xᵢ₊ⱼ } } }.
```

==

Simple implementations of   " Z := X+y "   have many important
applications. especially to the summation of series, numerical
quadrature. and the solution of initial value problems. For
example. the recommended way to sum an infinite series $\sum_j y_j$ is
to first decide how many terms to retain. and then add them up,
starting with the smaller ones in order to avoid an excessive
accumulation of roundoff. But this is impractical advice when the
number of terms is both so big that there is no place to save them
and, at the same time, impossible to determine in advance except
by computing all the terms twice, once forward and once backward
in the order that they will be added. A better way is to perform
the addition in DP starting from the big terms at the beginning.
That DP sum will suffer scarcely more error than is already
inherent in the terms y_j , which we assume to be accurate to WP
at best except possibly for the first few, which are often exact.
Then summation can be terminated when $|y_j|$ is deemed negligible
by some criterion that could depend upon the accumulated sum so
far. Here is an outline of how the summation might be performed:

        errbnd := 0 ; Z := y_0 ; ... i. e., z := y_0 ; ζ := 0 ;
        for j = 1 to MaxJ do   ... MaxJ could run to millions
          { y_j : = ... ;
             Z := Z + y_j ;   ... i. e., $z+\zeta$:= $z+\zeta$ + y_j
             errbnd := errbnd + (bound for error in y_j)
          } until $|y_j|$ < Tolerance(j, z, errbnd) ; ...

Now z approximates the sum of the series with an error partly
dependent upon how the Tolerance was defined. partly a sum of
the errors in the terms y_j (that part is bounded by errbnd),
but otherwise independent of the number of terms accumulated.

A very important application of   " Z := X+y " is to the numerical
solution of an initial value problem $dz(t)/dt = f(z(t))$. The
typical numerical process consists of repeated updates like

        $z(t+\Delta t)$:= $z(t) + f(z(t), \Delta t)*\Delta t$,

where now z(.) is the numerical approximation and f(.. .) is a
formula designed to yield better accuracy as Δt is made smaller.
Both z(.) and f(., .) are generally vectors. The worst rounding
error in this process is usually the one that occurs at the last
addition. It can cause the error accumulating in z(.) actually
to grow as $\Delta t \to 0$. To suppress it. update this way instead:

        y(t)   := $f(z(t), \Delta t)*\Delta t$;   ... to WP
     $Z(t+\Delta t)$:= $Z(t) + y(t)$; ... = $z(t+\Delta t)$ + $\zeta(t+\Delta t)$ to DP .

When z(.) is computed in this way its accuracy improves, as Δt
shrinks, until a limiting accuracy is achieved which is limited
almost entirely by the roundoff error that contaminates f(., .)
and by the time taken to perform the very many updates required
when Δt is very tiny.

In cases when the terms y_j being summed are uncertain because of
roundoff and always smaller than the current sum, the assignment
" Z := Z + y_j "   can be implemented in a way even simpler than was
presented above. but the details are not needed here. See them
in the paper by S. Linnainmaa (1974) cited in the bibliography.

DP Addition/Subtraction,   Z := X+Y :
Given X = $x+\xi$ and Y = $y+\eta$, we seek to implement the DP sum
Z := X+Y by some means faster than distillation of $x+\xi+y+\eta$ but
not too much less accurate. A slight extension of the previous

procedure seems the obvious thing to try first:

        If |x| < |y| then swap(X, Y) ;   ... so |x| ≥ |y|
        t := ((η+ξ) + y) + x ;   ... computed in WP
        τ := (((x-t) + y) + ξ) + η ;   ... in WP
        z := t+τ ;   ... to WP
        ς := (t-z) + τ ;   ... exactly in WP

Does $Z = z+ς ≠ X+Y$ to DP ? No; it fails because the sub-
expression (x-t) may occasionally be inexact, as can be seen in
the following example with binary arithmetic rounded to 4 sig.
bits. Try x = y = 1111. and ξ = η = o.o1 ; then t = 100ooo.
and [x-t] = [-10001.] = -1000o. or -1001o. according as rounding
follows IEEE 754 or a DEC VAX . Then τ = -0.1 or -10.1 resp.
instead of the correct -1.1 . The obvious procedure fails.

A slightly simpler procedure seems always to work much better:

        Given X = x+ξ and Y = y+η ,
        If |x| < |y| then swap(X, Y) ;   ... so |x| ≥ |y|
        t := (ξ + y) + x ;   ... computed in WP
        τ := (((x-t) + y) + ξ) + η ;   ... in WP
        z := t+τ ;   ... to WP
        ς := (t-z) + τ ;   ... exactly in WP

Because [x-t] = (x-t) exactly now, $Z = z+ς ≠ X+Y$ better than
before, at least when WP arithmetic conforms to IEEE 754, but
I cannot yet say how much better. At best, $Z = X' + Y'$ where
X' and Y' agree respectively with X and Y to about DP in the
sense that $|X'-X| ≤ ε^2|X|/2$ and $|Y'-Y| ≤ ε^2|Y|/2$, as the next
examples suggest:

Take x = 1010 , ξ = -o.1 , y = -1001 and η = -o.oooool using
arithmetic rounded again to 4 sig. bits., so $ε = 2^{-4}$. Now
X = 1001.1 and Y = -1001.000001 , so X+Y = 0.011111 . But the
computed values are t = 0 , τ = o.1 and z = Z = o.1 instead.
When ξ is changed from -o.1 to -o.o1111 , changing X+Y to
0.100001 , then t changes to 1 and τ to -o.1 , but z = Z
= o.1 unchanged from before. Consequently $Z ≠ X+Y$ to WP but
not DP . If these examples illustrate the worst that can happen
to the simple algorithm above then it is at least as good as has
widely been considered acceptable in the past; in fact, taking
IEEE 754's single-precision format (24 sig. bits, $ε = 2^{-24}$) as
WP would produce DP results at least as good as in the single-
precision (48 sig. bits) formats on CDC CYBERs and CRAYs.

But the simple algorithm is not good enough if we want $Z ≠ X+Y$
to DP so accurately that $|Z - (X+Y)| ≤ ε^2|X+Y|$, especially
when X and Y have opposite signs and mostly cancel. To achieve
that superior accuracy, it seems necessary to augment the simple
algorithm above so that it will do something special whenever
either t = 0 or [[x-t]+y] = 0 . Here is a suggestion:

```
        Given  X = x+ξ  and  Y = y+η ,  to compute  Z := X+Y :
 Start:  If  |x| < |y|  then  swap(X, Y) ;  ...  so  |x| ≥ |y|
        t := (ξ+y) + x ;                     ... computed in  WP
        if  t = 0  then
           { t := (x+y) + ξ ;               ... exactly in WP
             Z := t+η ;                      ... to  DP
             exit } ;
        u := (x-t) + y ;                     ... to  WP
        if  u = 0  and  η ≠ 0  then
           { X := t+ξ ;                      ... to  DP
             Y := η ;                        ... to  DP
             go back to  Start } ;
        τ := (u+ξ) + η ;                     ... in  WP
        z := t+τ ;                           ... to  WP
        ζ := (t-z) + τ ;                     ... exact in  WP
        ...  now  Z = z+ζ ≠ X+Y .
```

On what kinds of arithmetic, if any, can this procedure be
trusted? Must a trustworthy procedure be so complicated? Can a
trustworthy procedure be devised that is free from the tests and
branches that inhibit vectorization and other kinds of concurrent
execution?

DP product of WP variables, $Z := x*y$:
In the absence of special hardware, representing a product $x*y$
of WP variables exactly as a sum of WP variables $z+ζ = x*y$
takes trickery that was discovered almost twenty years ago by two
Dutchmen, T. J. Dekker and G. W Veltkamp. The basis for the
trickery is a procedure for rounding a WP number x to half-
precision. Let $h(x) := x$ rounded to just half WP , and let
$θ(x) := x - h(x)$; here think of $h(x)$ as the *head* of x and
$θ(x)$ as its *tail*, both WP numbers with zeros across at least
half their sig. bits. Then we may express $x*y$ exactly as a sum,
$$x*y = h(x)*h(y) + h(x)*θ(y) + θ(x)*h(y) + θ(x)*θ(y) ,$$
in which each product is computed exactly in WP arithmetic; and
this sum of four terms can be distilled into a sum of two. To
carry out this strategy we must first discover how precise WP is
without using anything but standard WP arithmetic, then define
the function $h(x)$, then condense the distillation process into
something more economical.

Throughout the discussion, WP arithmetic is assumed to be binary
floating-point rounded either as specified in IEEE 754 or as a
DEC VAX does it. For any other kinds of arithmetic the tricks
below become appreciably more complicated.

The precision of WP is characterized by the constant $ε$, which
can easily be computed thus: $ε := | (2.0/3.0 - 0.5)*3.0 - 0.5 |$.
This $ε$ is a power of 2 : either $ε = 2^{-2k}$ or $ε = 2^{1-2k}$
according as the number of sig. bits carried in WP is even or
odd. A similar WP computation produces a multiplicative *mask*
$m = 2^k + 1$ needed to compute the function $h(x)$:

$$b := 1/|(4.0/3.0 - 1)*3 - 1| ; \quad ... \quad = 1/(2\varepsilon)$$
$$r := \sqrt{(4*b)} ; \quad\quad\quad ... \quad = \sqrt{(2/\varepsilon)}$$
$$m := ((r + r*b) - r*b) + 1 ; \quad ... \quad = 2^k + 1$$
$$h(x) := (x - m*x) + m*x ; \quad ... \quad = x \text{ rounded to } k \text{ sig. bits}$$
$$\theta(x) := x - h(x) .$$

Round every arithmetic operation above except $\sqrt{}$ to WP ; but $\sqrt{}$ is accurate enough if accurate to 1% . Do not omit parentheses.

Now the DP assignment " Z := x*y " can be implemented:
$$z := x*y ; \quad\quad\quad ... \quad \text{rounded to } WP$$
$$\zeta := ((h(x)*h(y) - z) + h(x)*\theta(y) + \theta(x)*h(y)) + \theta(x)*\theta(y) .$$

Then $Z = z+\zeta = x*y$ exactly because ζ is computed without any rounding error. But the cost is high: seven WP multiplications and ten additions or subtractions. The cost grows somewhat when over/underflow makes scaling necessary. For instance the product m*x required for h(x) could overflow spuriously even though h(x) lies well within range; in such a case compute b*h(x/b) instead of h(x) . If arithmetic conforms to IEEE 754 there is no way for $\theta(x)$ to underflow, but on other machines that risk has to be addressed too. For simplicity's sake we shall skip over the scaling techniques that would be required to deal with over/underflow conscientiously.

In principle, the DP assignment " Z = x*y " can be used to compute products to arbitrarily high precision. The product
$$(\Sigma_i \ x_i)*(\Sigma_j \ y_j) \ = \ \Sigma_i \ \Sigma_j \ x_i*y_j \ = \ \Sigma_i \ \Sigma_j \ (z_{ij} + \zeta_{ij})$$
can be *distilled* into a sum of a nearly minimal number of WP numbers, though in practice a different expression
$$\Sigma_i \ \Sigma_j \ (\ h(x_i)*h(y_j) \ + \ h(x_i)*\theta(y_j) \ + \ \theta(x_i)*h(y_j) \ + \ \theta(x_i)*\theta(y_j) \ )$$
may take less time because it reuses heads and tails. Omitting negligible products too saves more time, as we shall see.

DP Multiplication,   Z := X*Y :

Given $X = x+\xi$ and $Y = y+\eta$, a reasonably accurate and economical DP product $Z := X*Y$ is achieved by approximating $X*Y = x*y + x*\eta + \xi*y + \xi*\eta$ by its first three terms, of which only the first need be computed in DP :

$t+\tau = \mathcal{T}$

$$t := x*y ; \quad\quad\quad ... \quad = t+\tau \text{ exactly in } DP$$
$$\tau := (x*\eta + \xi*y) + \zeta ; \quad ... \quad \text{in } WP$$
$$z := t+\tau ; \quad\quad\quad ... \quad \text{to } WP$$
$$\zeta := (z-t) + \tau ; \quad\quad ... \quad \text{exact in } WP$$
$$... \quad \text{Now} \quad Z = z+\zeta \doteq X*Y .$$

The cost is nine WP multiplications and fifteen additions or subtractions.

DP Division, Z := X/Y :

Given $X = x+\xi$ and $Y = y+\eta$, we may approximate X/Y by a process reminiscent of long division as carried out by hand and portrayed in the following diagram:

$$
\begin{array}{r}
t+\tau \\
\hline
y+\eta \ ) \ \overline{x+\xi+0} \\
yt+\eta t \\
\hline
r+... \\
y\tau+... \\
\hline
...
\end{array}
$$

Here is the program:

$$t := x/y ; \qquad\qquad\qquad\qquad ... \text{ to } WP$$
$$s := y*t ; \qquad\qquad\qquad ... = s+\sigma \text{ exactly in } DP$$
$$\tau := (((x-s) - \sigma) + \xi - \eta*t)/y ; \quad ... \text{ to } WP$$
$$z := t+\tau ; \qquad\qquad\qquad\qquad ... \text{ to } WP$$
$$\zeta := (t-z) + \tau ; \qquad\qquad\qquad ... \text{ exact in } WP$$
$$... \text{ Now } Z = z+\zeta \doteqdot X/Y .$$

It costs two WP divisions. eight multiplications and seventeen
additions or subtractions, not much more than DP multiplication.

Division can be carried out to arbitrarily high precision in a
similar way at a cost not much worse than that of multiplication.

DP Square root, $Z := \sqrt{X}$:
If $t \doteqdot \sqrt{x}$ then $t + (x-t^2)/(2t) \doteqdot \sqrt{x}$ to twice as many figures.
Then the series expansion $\sqrt{(x+\xi)} = \sqrt{x} + \xi/(2\sqrt{x}) - ...$ explains
why $t + (x - t^2 + \xi)/(2t)$ approximates $\sqrt{X} = \sqrt{(x+\xi)}$ to DP by
combining the two processes. None but WP operations appear in
following program for a DP square root:
$$t := \sqrt{x} ; \quad ... \text{ to } WP$$
$$\tau := 0.5*((((x - h(t)^2) - 2*h(t)*\theta(t)) - \theta(t)^2) + \xi)/t ;$$
$$z := t+\tau ;$$
$$\zeta := (t-z) + \tau ;$$
$$... \text{ Now } Z \doteqdot \sqrt{X} .$$

Conclusions:
Processes for *distilling* sums and for computing *heads* and *tails*
provide a foundation in principal for floating-point arithmetic of
arbitrarily high precision implemented entirely in WP . That is
why theoretical questions like
      " How accurately can ... be computed in WP arithmetic? "
cannot be answered in absolute terms; such questions make sense
only if some limit is imposed upon the time and space that will be
spent upon the computation. Such questions about DP addition
and subtraction have not yet been settled fully satisfactorily,
although reasonably efficient implementations with relative errors
not much worse than ε^2 have long been known for multiplication,
division and square root.

Can the DP addition algorithm suggested above, or some others,
settle all remaining questions about DP arithmetic to everyone's
satisfaction? No. Serious questions about software engineering
will remain. Consider a program that achieves its robustness and
efficiency partly by exploiting a little DP arithmetic. Such a
program may work correctly, after recompilation, on a wide range
of computers of diverse manufacture that all conform to the IEEE
standard 754 . Among such machines are Apple Macintoshes, ELXSI
6400's. IBM RT-PCs. Sun IIIs. HP Precision Architectures, etc.
In the absence of underflow the program will likely work correctly
on any DEC VAX. But the program is likely to fail occasionally
when run on other commercially significant machines, including
IBM /370s, CDC Cybers. Univac 1100s. and Crays. Who deserves
the blame for those failures of an ostensibly portable program?

If a small change could render that program fully portable without
any serious performance penalty, then the original program's
lapses could be blamed upon the programmer's inexperience or

indifference. But nobody has ever found an implementation of DP
arithmetic that is portable to all the machines mentioned above,
and that not for lack of looking. Then should DP arithmetic be
prohibited because it is not portable? Such a prohibition seems
immoral, for it denies the benefits of better arithmetic to those
who have paid for it, as well as unenforceable.

In the past, high-precision floating-point arithmetic has been
regarded as a fixture of the programming environment, usually
implemented in machine language if available at all, certainly
not something that could be embedded within a small portable
program written entirely in some Fortran-like language. Now,
if we think of high-precision arithmetic merely as a programming
technique to be employed in small doses as needed, can we look
beyond our present horizons to see a wider range of numerically
stable portable procedures?

Annotated Bibliography:

The first instance I know of a computation that recovered rounding
errors in order to cancel them off was S. Gill's version of the
Runge-Kutta method, "Process for the Step-by-Step Integration of
Differential Equations in an Automatic Digital Computing Machine,"
Proc. Cambridge Phil. Soc. 47 (1951) 96-108. His trick is still
widely misunderstood in terms of algebraic identities, correct
for the fixed-point computations for which they were designed,
that confer no benefit upon floating-point computation. However,
the ideas behind Gill's trick were exploited correctly in codes
that solved differential equations on the IBM 7090 around 1959;
they carried a term from the MQ register, that would otherwise
have been discarded, forward to the next time-step, thereby much
attenuating the accumulation of roundoff over vast numbers of tiny
time-steps. Because Fortran forbade any reference to the MQ
register, these codes had to be written in assembly language and
are now lost. I was able to transfer several of these codes to
Fortran II with the aid of the "DLAF Package" of Fortran -
callable functions that I contributed to the IBM 7090 SHARE
library in the early 1960's; cf. SHARE SD#3021 (renumbered from
#1480), resubmittal dated Jan. 15, 1964, 3-page abstract + 2-
page listing + 10-page write-up, available from IBM only on
microfiche. The write-up describes numerous applications besides
the ones mentioned above in this paper. Part of the package's
efficiency was owed to the passage of function arguments *by value*
when they were expressions evaluated in the AC-MQ registers. But
by 1963 the advent of Fortran IV, which passed arguments solely
by reference, had rendered the package less efficient than the
full double-precision supplied with the compiler.

By 1965 ways had been discovered to recover addition's rounding
error in programs written entirely in Fortran-like languages,
albeit at the cost of a little extra arithmetic. The ideas were
published by O. Møller in two notes on " ... quasi double
precision ...." *BIT* 5 (1965) 37-50 and 251-255, and by me in a
letter, "Further Remarks on Reducing Truncation Errors," *Commun.
ACM* 8 (1965) 40. A thorough analysis, validation and comparison
of tricks like these has been published by S. Linnainmaa in
"Analysis of some known methods of improving the accuracy of
floating-point sums," *BIT* 14 (1974) 167-202.

Meanwhile, to sum a doubly infinite series that suffered from
massive cancellation, I had stumbled upon a distillation process
very like the one presented in this paper. But I was baffled when
it gave different results in double-precision on the new IBM 7094
than on the old IBM 7090, and different yet again on the CDC
6400. The old 7090's results turned out to be correct. In the
end the discrepancies were attributed on the 7094 to a lack of a
guard digit in its double-precision hardware, on the 6400 to
its Fortran compiler's use of two instead of five floating-point
instructions to effect a single-precision subtraction. These
flaws undermined the computation of the DP sum " Z := x+y " of
two WP numbers, invalidating the distillation process. These
flaws caused anguish to others besides myself; see my "Survey of
Error-Analysis," *Proc. IFIP Congress 1971*, ed. by C. V. Freeman
(1972), North-Holland Publ. Co., Amsterdam, vol. 2, 1214-1239. It
contains a tricky implementation of the DP sum " Z := x+y " of
two WP variables that works correctly on all North American
computers with built-in floating-point hardware except the CDC
Cyber 205. Therefore distillation, and hence DP arithmetic,
can be implemented in principle (but inefficiently) in a way
that is completely portable to all computers but that one.

The first distillation algorithm published was M. Pichat's
"Correction d'une Somme en Arithmetique a Virgule Flottante,"
Numerische Math. 19 (1972) 400-406. It is astonishingly simple,
without the presort and the backward pass in my algorithm; yet
his will distill a large number N of summands in not much more
time on average than twice what mine takes. In the worst case his
scheme can take time proportional to N^2 , whereas the worst case
for mine, though far better than that, has yet to be determined.

G. Bohlender contributed two improvements to distillation in
"Floating-Point Computation of Functions with Maximum Accuracy,"
IEEE Trans. Computers C-26 no. 7 (1977) 621-632 and "Genaue
Summation von Gleitkommazahlen," *Computing Supplement* 1 (1977)
1-21. One improvement was an elegant formalism by which to prove
the convergence of distillation iterations. The other was a
family of stopping criteria suitable for use when less than full
accuracy is required in the final sum. These stopping criteria
for Pichat's distillation, and for another one that presorts
the summands, are described also in the book *Computer Arithmetic
in Theory and Practice* by U. Kulisch and W. Miranker (1981)
Academic Press, New York, 192-209; but the programs therein
look much more complicated than mine partly because they refer to
the exponent and significand fields of floating-point numbers as
if the programs were intended for machine-language. Actually the
directed roundings are the only features they need that might not
be found in conventional Fortran-like languages. For a clearer
account of Bohlender's algorithm see "Parallel Algorithms for
the Rounding-Exact Summation of Floating-Point Numbers" by H.
Leuprecht and W. Oberaigner, *Computing* 28 (1982) 89-104. Their
directed roundings are superfluous for most practical purposes;
iteration could just as well be stopped as soon as one found a
distilled partial sum $x_{k} + \ldots + x_{N-1} + x_N$ with no fewer terms
than desired and with $|x_{k}| = (|x_1| + \ldots + |x_{k-1}| + |x_{k}|)$ as
computed in WP .

Some questions about distillation still weigh upon my mind. Why
do all distillation algorithms usually run so much faster than
anyone has been able to prove? Why does the backward pass in my
algorithm usually speed it up a little, sometimes a lot, but
sometimes slow it down very slightly compared with Pichat's?

DP multiplication has been more troublesome in the past fifteen
years than in the previous fifteen. Older machines implemented
the DP product " Z = x*y " of two WP variables in one machine
instruction (IBM 7090, UNIVAC 1107, DEC PDP-10) or two (CDC
6600), far faster than achievable using half-precision heads
h(x) and tails θ(x) ; but recent designs have lacked any such
DP capability. That atrophy could be a byproduct of a linguistic
misconception among the designers of machines intended to support
Fortran-like languages. Here is why.

Fortran appears to offer almost what we want; it has a function
DPROD(x, y) that returns the exact DOUBLE PRECISION product of
two SINGLE PRECISION variables x and y . And some dialects of
Fortran provide QPROD(X, Y) , the EXTENDED (i. e. quadruple
precision) product of DOUBLE PRECISION variables. But if XX
and YY are variables of type EXTENDED there is no way to return
an exact product, call it OPROD(XX, YY) , because there is no
OCTUPLE PRECISION or DOUBLE EXTENDED data type in the language.
Hence there seems to be no way to supply a two-word result from an
operation upon one-word operands of the widest wordsize available.

But actually, if the COMPLEX data attribute is supported in the
language, OPROD(XX, YY) could be of type COMPLEX EXTENDED. The
function DPROD(x, y) could be a RECORD in Pascal, which
traditionally has only one REAL type; the STRUCT construction
in the language C is a humane way to provide QPROD. But C and
Pascal came too late to influence the designers of those machines
upon which now we must compute heads and tails.

Other ways exist that avoid collisions with compilers by taking
refuge in the run-time library. One way is to provide a multiply-
add function   prad(a,b,c) := [a*b + c] rounded to WP with just
one rounding error; then z := prad(x,y,0) and ζ := prad(x,y,-z)
supply the DP product   z+ζ = Z := x*y of WP operands in just
two operations. The full DP product   z+ζ = Z := (x+ξ)*(y+η)
might take just six operations instead of twenty-four:
          τ := x*η + ξ*y ;   ... to WP
          z := prad(x,y, τ) ;   ζ := prad(x,y,-z) + τ .

Another way is to provide a DP *accumulator*, a special global
variable into which library programs *dpadd*, *dpmul*, ... written
in machine language (as are *log* and *cos*), put their result.
This approach was followed for the DLAF package mentioned above.
A similar approach, but carried to far greater lengths in a
super-accumulator for scalar products, is pursued by Kulisch and
Miranker in their book mentioned above and in the realization of
their ideas by IBM's "High-Accuracy Arithmetic Subroutine
Library (ACRITH)," program numbers 5664-185, 5665-337, 5666-320,
documentation order numbers GC33-6163-02 and SC33-6164-02 (1986).
But such schemes do not lend themselves to portable programming in
the standardized Fortran-like languages as they are to-day.

A major contribution to portability was published by T. J. Dekker
in "A Floating-point Technique for Extending the Available
Precision." *Numerische Math.* **18** (1971) 224-242. He showed how to
implement $h(x)$ and $\theta(x)$ and the exact DP product " Z := x*y "
of WP variables in an ostensibly portable way. From the exact
DP product and sum he implemented all the arithmetic operations
upon DP variables. S. Linnainmaa, in "Software for Doubled-
Precision Floating-Point Computations," *ACM Trans. Math. Soft.* **7**
(1981) 272-283, has generalized Dekker's algorithms, proofs
and error bounds so that they really are portable to a wide range
of *faithful* arithmetics with diverse radices and odd word-sizes.
But not all arithmetics. Multiplication and subtraction on a CRAY
are unfaithful, as is subtraction on CDC Cybers and Univac
11xx's, so his codes must malfunction on those machines. And
even when his DP addition/subtraction works, it is somewhat
less accurate than the second of my DP addition programs
presented above. To reach the accuracy to which my third program
aspires, but in a universally portable way, seems to require so
complicated a program that nobody has written it.

If high-precision arithmetic were all we wished to perform, we
should choose a programming environment in which that is made as
convenient as possible. Such an environment is provided by ...
MACSYMA's Bigfloat arithmetic; see R. J. Fateman "The MACSYMA
   Big-Floating-Point Arithmetic System," Proc. 1976 ACM Symposium
   on Symbolic and Algebraic Computation, ed. by R. D. Jenks.
   DOE/MACSYMA and VAXIMA (prog. #9631) can be obtained from the
   National Energy Software Center, Argonne National Laboratory,
   9700 South Cass Ave., Argonne IL 60439; tel. # (312) 972-7250
R. P. Brent's MP Package; GIVE REFERENCE >>>>>>>>>>>>>
T. E. Hull's "Numerical Turing Language;"   >>>>>>>>>>>>>

But high-precision is not what we most often need. Instead, we
often need arithmetic with a little more precision -- DP is
enough -- in a small part of an otherwise WP program intended
for the widest possible distribution. That kind of arithmetic is
simulated very sparingly in the excellent elementary function
codes found in W. J. Cody and W. Waite, *Software Manual for the
Elementary Functions* (1980), Prentice-Hall Inc., Englewood
Cliffs, New Jersey. The *math.* library of elementary functions,
distributed in 1986 by the Regents of the University of California
with 4.3 BSD UNIX, does likewise. But algorithms that achieve
their speed and accuracy by using a little DP arithmetic pose a
major headache for the would-be distributor of portable software.

The would-be consumer of such codes can best avoid that headache
by purchasing a machine whose arithmetic conforms to ANSI/IEEE
Standard 754-1985 for Binary Floating-Point Arithmetic as
described in document no. SH10116 available from the IEEE Inc.,
345 East 47th St., New York NY 10017. (For a more readable
description see "A Proposed Radix- and Word-length-independent
Standard for Floating-point Arithmetic" by W. J. Cody et al.,
IEEE MICRO (Aug. 1984) 86-100.) Tricky codes that simulate
DP arithmetic in a way intended to be as portable as possible may
run slower than they have to, but they will almost certainly run
correctly on a machine that conforms to that standard.

~~~~~~

Still to come: more applications to
          Algebraic functions
          Correctly rounded WP division