Series Approximation Methods for Divide and Square Root in the Power3™ Processor

Ramesh C.Agarwal
IBM Corporation
Research Division
Yorktown, NY
agarwal@watson.ibm.com

Fred G.Gustavson
IBM Corporation
Research Division
Yorktown, NY
gustav@watson.ibm.com

Martin S. Schmookler
IBM Corporation
Server Development
Austin, TX
martins@austin.ibm.com

Abstract

The Power3 processor is a 64-bit implementation of the PowerPC™ architecture and is the successor to the Power2™ processor for workstations and servers which require high performance floating point capability. The previous processors used Newton-Raphson algorithms for their implementations of divide and square root. The Power3 processor has a longer pipeline latency, which would substantially increase the latency for these instructions. Instead, new algorithms based on power series approximations were developed which provide significantly better performance than the Newton-Raphson algorithm for this processor. This paper describes the algorithms, and then shows how both the series based algorithms and the Newton-Raphson algorithms are affected by pipeline length. For the Power3, the power series algorithms reduce the divide latency by over 20% and the square root latency by 35%.

1. Introduction

The Power3 processor is a 64-bit implementation of the PowerPC architecture for the IBM RISC System/6000™ [1] workstations and servers. It has two identical floating point units which each execute all of the double precision floating point instructions that were in the Power™ and Power2 [2,3] processors, including square root. It also adds a complete set of single precision instructions. The dataflow pipeline in each unit is optimized for the double precision accumulate (multiply and add) instructions [3] which do not round the intermediate product.

The two units share the use of the FPR (floating point register) which contains the 32 architected registers, but each unit has its own copy of a FPB (floating point buffer) containing 24 renameable registers. The FPR has 7 read ports and 4 write ports. The write ports are used only to update the FPR from the FPBs for completed operations. All results from the execution units, as well as all floating point data loaded from memory or cache, go directly to both FPBs. This organization supports the speculative execution of the instructions.

Usually, denormal input operands are processed directly, but in some cases, issuing of instructions to a unit may be held up when a denormal input is detected and prenormalization is required. The instruction issuing control permits out-of-order execution when operands are not available for execution of the next instruction. To simplify the instruction issuing, no data dependent delays are permitted in the pipeline except for prenormalization. The need for prenormalization is detected early enough that subsequent instructions can be held up. Also, except for divide and square root, all instructions have the same latency, and independent instructions can begin in each unit every cycle.

This simplification of the instruction issuing was done at the expense of greater complexity in the pipelined dataflow. In earlier implementations of this architecture, the dataflow was simplified without significant loss in performance by allowing additional cycles when certain rare cases ocurred, such as underflow, overflow, or very long normalization shifts. Results were fed back into the pipeline for denormalization or for rebiasing the exponent when underflow or overflow occurred and exceptions were enabled. A shorter leading zero anticipator (LZA) [4] and normalize shifter was used that would handle more than 90% of the normalization shifts required, and results could be recycled through the shifter for the remaining cases. In the Power3 design, the LZA and normalize shifter are designed to handle the maximum shift possible. With 53 bit mantissas, the accumulate operation requires a 108 bit LZA and a 160 bit normalization shifter. A new LZA [5] was designed which is less complex than previous designs, permitting the full range required, yet with less chip area and with

less delay.

The normalization shift is also limited when the result is a denormal, to avoid any additional shifting to denormalize it [6]. Rebiasing of the exponent is also done within the same latency, as well as forcing maximum or minimum values when exceptions are disabled.

These constraints, along with the need for higher frequency operation, has resulted in a 4-stage execution pipeline. This does not include the accessing of operands from the register file. However, forwarding of both rounded and, in most cases, unrounded results for dependent operations makes the effective latency only three cycles. The technique used for forwarding unrounded results to the multiplier operand of the next instruction was described in [3], and consists of adding an additional copy of the multiplicand in the multiply array, since $x(y+1)$ is equal to $xy+x$. We cannot forward results back to both operands of the multiplier. However, we can forward a rounded result back to the mantissa alignment unit after its exponent has been forwarded for calculating the shift amount, thus making the effective latency to the add operand also three cycles.

Two common choices for implementing both divide and square root are SRT [7,8] and functional iteration [9,10]. The choice for square root usually defaults to whatever mechanism is provided for divide, with some modification as is necessary. SRT division is most efficiently implemented with a separate dataflow that doesn't require all the facilities needed for floating point addition, and which maximizes the number of quotient bits developed each cycle. Several techniques have been developed for improving the performance of this class of algorithm. It also allows the execution of other instructions concurrently with the divide or square root instruction. The total execution time for SRT division and square root is linear with respect to the number of bits of precision.

Functional iteration can usually benefit from use of the high speed accumulate dataflow already available. Some additional hardware is needed, however, which consists of a lookup table for a starting value for the algorithm, and a state machine or microcode unit to execute the algorithm. With two separate floating point units available, concurrent execution of other instructions is still possible. Overall, comparison shows that for double precision, with two execution units, functional iteration often has significantly less latency and better overall performance than an SRT implementation, and usually requires less silicon area.

The most widely used method of functional iteration is the Newton-Raphson algorithm. It converges quadratically, which means that the precision is doubled with each iteration. A variation of this algorithm was used in the earlier IBM RISC System/6000 processors and the later Power2 based processors [11]. It allows use of microinstructions which are essentially identical to those available in the instruction set. Each iteration requires several dependent microinstructions. These earlier processors had an effective pipeline latency of only two cycles, and for these, the Newton-Raphson algorithm was fairly efficient, executing divide in 17 cycles and square root in 26. However, the effective 3 cycle latency of the Power3 dataflow would substantially increase the execution times for double precision divide and square root using that algorithm to 25 and 37 cycles respectively.

Other methods of functional iteration are usually based on a series approximation. The Goldschmidt algorithm [12] is one such method, but it too requires many dependent operations, as well as additional special operations requiring some modification of the dataflow. Instead, the Power3 uses algorithms based on power series approximations which are more efficient for use with longer pipelined dataflow latencies. Similar algorithms [18] were used in an earlier microprocessor chip from IBM called the 620. We also use a special algorithm for rounding which further improves performance. All rounding modes required by the IEEE-754 floating point standard are supported [13].

The following sections describe the microcode unit and tables, the rounding algorithm, and the algorithms for divide and square root. Then performance comparisons with the Newton-Raphson algorithm are given for different pipeline latencies. An analysis of the accuracy for each algorithm is included with the description.

2. The Microcode Unit and Tables

The algorithms are executed using a microcoded state machine. An independent operation such as the accumulate instruction can be started each cycle in the execution unit. Some special operations such as loading the operand registers from a lookup table, can also be executed. If the divisor operand is denormal, it must first be prenormalized. Two additional exponent bits are provided in the dataflow to handle the prenormalized operand and also to avoid underflow or overflow of any intermediate results during execution of the microcode. The underflow and overflow controls are turned back on only during execution of the last microinstruction. All instructions are executed in round to nearest mode, regardless of the mode specified, prior to execution of the rounding algorithms, which will be described below. When denormal results occur, additional cycles are taken in the form of additional microinstructions, but these cases are detected early enough that the instruction issuing mechanism is not affected. It merely waits for a

signal that the last microinstruction is being executed.

The microcode explicitly includes "wait" instructions where the next operation is dependent on the result of a previous operation that is still in the pipeline. Since all such dependencies are known, this simplifies the control. Similarly, the microcode explicitly specifies for each input operand of each operation whether it should come from the FPR or FPB, from one of several constants, or from either the third or fourth stage of the pipeline. Since we can forward results simultaneously from both the third and fourth stages, we can do an operation which is dependent on two previous results without requiring any temporary storage of either one.

For most cases, the latency for double precision divide is 20 cycles, which is when another operation dependent on that result could begin execution. An independent instruction could begin after 18 cycles. Similarly, the latency for double precision square root is 24 cycles. The latency for both single precision instructions is 16 cycles.

Each execution unit has its own microcode unit, so both can be executing divides or square roots at the same time. The instruction control registers for each unit can be loaded either from the instruction queue of the issuing unit or from the microcode unit. Once a divide or square root instruction begins, however, no other instructions can execute in that unit until the instruction is finished.

The divide instructions calculate $q=a/b$, and the square root instructions calculate $q=\sqrt{b}$. Both instructions use tables which are accessed by the high order bits of the b operand. These tables can be thought of as existing in a ROM, although they are actually synthesized using standard cell circuits. The algorithms for single precision divide and square root are shortened versions of the double precision algorithms, and use the same tables.

3. Precision and Rounding Requirements

Both algorithms must provide IEEE 64 bit double precision results in all four rounding modes. Therefore, we need 53 bit precision for the mantissa. The Newton-Raphson algorithm used in the previous processors required that we obtain intermediate results of the quotient and the reciprocal of the divisor which were both correct to within one ulp (unit in the last place), although not necessarily rounded properly. Then one additional iteration of the algorithm would produce a correctly rounded result, with the inexact flag properly set.

In the Power3, we use a faster method, but it requires some slight modifications to the dataflow. First, we must produce a result which is accurate to within 1/4 ulp prior to rounding, and be able to observe the two bits below the least significant bit. Each possible value of those two bits corresponds to a quarter ulp. The infinitely precise result must either be in that quarter ulp or in one of the two adjacent quarter ulps.

For the round to nearest rounding mode, when the bits are both zeros or both ones, then the approximate result q_a is rounded to nearest to give the correct result. But we still must set some result flags properly. One is the inexact exception bit required by the IEEE-754 standard [13], and another flag indicates when an inexact result is rounded up. We take care of both of these with a special error calculation in which we do not use the full result, but only determine its sign and whether or not it is zero. If q_r is the rounded result, then for divide we calculate $c = a - bq_r$, and for square root, we calculate $c = b - q_r q_r$. It is important to note that both of these calculations depend on using the accumulate operation to avoid an erroneous result of zero.

When the bits are 01 or 10, then we know the result must be inexact, but we don't know whether q_a should be rounded up or down. We need to do a special error calculation using q_h which is the truncated value of q_a plus a half ulp. For divide, we calculate $c = a - bq_h$. The multiplier has been specifically designed to accomodate one additional bit in the multiplicand to allow this calculation. We cannot, however, do $c = b - q_h q_h$ for square root. Instead, we calculate q_a twice, once rounded down and once rounded up. Calling these two results q_t and q_u, it can then be shown, using Tuckerman rounding (p. 115 in [11], p. 134 in [16]) that q_a should be rounded up only if $b > q_t q_u$. Therefore, we calculate $c = b - q_t q_u$.

For the directed rounding modes, we only need to use q_r in the error calculation, but may need to correct the result by adding or subtracting an ulp. The rounder is used for adding an ulp, but to subtract an ulp, q_r is multiplied by 1 - 1/2 ulp (exponent of -1 and mantissa of all ones) and the result is truncated.

In all cases, we only need from the error calculation its sign and in some cases whether or not it is zero. This information is available from the third pipeline stage and is saved. The cycle after the error calculation is started, we immediately start down the pipeline a value of q which is either the correctly rounded result or one which needs to be incremented. By the time it reaches the third pipeline stage, we know whether or not it should be incremented.

An important part of both the divide and square root double precision algorithms is the method of achieving the required quarter ulp accuracy. The first term of the power series is used as an initial approximation, accurate to 7 or 8 bits, while the sum of the remaining terms constitutes a correction term. The initial approximation is

only 53 bits, and therefore contains a rounding error that most likely exceeds the quarter ulp budget. Therefore, we calculate a residual value from the first term and use that as a factor in the correction term. This residual term is similar to the error calculation that is used in the final rounding algorithm, and is also obtained using the accumulate operation.

In each of the following sections, we first describe the derivation of the power series that is used, and then provide a brief analysis which demonstrates how the correction term compensates for both the approximation error and the roundoff error of the first term. It also provides a way to calculate the most significant contributors to the remaining error, and shows that they are well within a quarter ulp. For the sake of clarity and brevity, we dismiss higher order error terms which are much less significant.

The single precision algorithms do not need the residual term as a factor, because all intermediate calculations, including the first term, are double precision, and therefore have insignificant rounding errors for single precision.

4. Divide Algorithm

Description and derivation. Let $q=a/b$, where a and b are double precision IEEE floating point numbers. Without loss of generality, we can assume that $1.0 \le a,b < 2.0$. The algorithm uses a 128 word table for an initial approximation y_0 of $1/b$. The table used gives a two-sided approximation, i.e., y_0 may be less than or greater than $1/b$. Each word in the table has 13 bits, including the implied one.

Let us define $e = 1 - b\,y_0$ where $|e| < 2^{-8}$.

Then $1/b = y_0/(1-e) = y_0\,(1 + e + e^2 + e^3 + \ldots)$ (1)

and, neglecting rounding errors,

$$q = a\,y_0\,(1 + e + e^2 + e^3 + \ldots) \quad . \qquad (2)$$

This needs to be correct to within a quarter ulp prior to final rounding, which corresponds to 55 bits of precision for the intermediate result. The dataflow does not provide for more than 53 bits of mantissa for intermediate results of operations, except for what was described above. If we first calculate an initial approximation $q_0 = ay_0$, the above calculation can be rewritten as the sum of this approximation plus a much smaller correction term. This is a well-known strategy described in Agarwal, et al [16] in which the correction term is several orders of magnitude smaller than the initial approximation, thus the rounding errors which occur during calculation of the correction term have no effect on the final result. At this point, we also truncate the infinite series, which introduces some error to the correction term. This is the approximation error of the algorithm. Therefore,

$$q_1 = q_0 + a\,e\,y_0\,(1 + e + e^2 + \ldots + e^5) \qquad (3)$$

where q_1 represents the properly rounded value of q.

If the calculations were all done using a higher precision, then aey_0 could be replaced by q_0e. However, as was mentioned in the previous section, the correction term must contain a residual term as a factor, since q_0 has been rounded to 53 bits, in order to compensate for its rounding error. This term is calculated as $e_2 = a - bq_0$, and is used to replace ae, since

$$a - b\,q_0 = a - b\,a\,y_0 = a\,(1 - b\,y_0) = a\,e \quad . \qquad (4)$$

The e_2 calculation is dependent on the accumulate function for its accuracy. The bq_0 product cannot be rounded prior to subtracting it from a, otherwise at least eight bits of precision would be lost. Thus, the algorithm, whose steps are shown in Table 1, calculates

$$q_1 = q_0 + e_2 y_0\,(1 + e + e^2 + \ldots + e^5) \qquad (5)$$

Brief analysis. To show how e_2 compensates for the rounding error in the initial approximation, we replace q_0 with $q_0 + r$, where r is its rounding error and is less than an ulp of q_0. Then from (4), $e_2 = a - b\,(q_0 + r) = ae - br$. We also replace y_0 with $(1 - e)/b$. We then obtain, after regrouping terms,

$$q_1 = q_0 + (a/b)\,e\,(1 - e^6) + r - r + re^6 \quad .$$

We can certainly ignore $r \cdot e^6$, leaving a result no longer affected by r. But $q_0 = a\,y_0 = (a/b) \cdot (1 - e)$, thus

$q_1 = \frac{a}{b}(1 - e^7)$, where $|e^7| < 2^{-56}$ is the relative error of the algorithm, corresponding to less than 1/8 ulp of q_1.

To this value we must add the rounding errors which occur during calculation of the correction term. Since $|e_2| < 2^{-7}$, each of the rounding errors must contribute no more than 2^{-7} ulp. There are six steps in the double precision calculation shown in Table 1 which may contribute to this rounding error. They are the steps shown as starting in cycles 2, 6, 7, 8, 10, and 11. Thus, the total error including the algorithmic error is no more than 0.18 ulp, and within the required 1/4 ulp. In an experimental simulation using one million trials, the maximum observed error was 0.1259 ulp. This is barely above the algorithmic approximation error of 1/8 ulp.

For single precision, we get $q_1 = (a/b)\cdot(1-e^4)$ where $|e^4| < 2^{-32}$, which represents the relative approximation error and translates to 1/256 ulp of q_1. We do not need to do an e_2 calculation, since all intermediate results are done in double precision. For the same reason, we do not

have to consider the arithmetic round-off errors, which are insignificant. Thus, the approximation error is again very comfortable.

Cy	DP Divide	SP Divide
1	Table lookup y_0	Table lookup y_0
2	$e = 1 - b \cdot y_0$	$e = 1 - b \cdot y_0$
3		
4		$q_0 = a \cdot y_0$
5	$q_0 = a \cdot y_0$	
6	$t_1 = 0.5 + e \cdot e$	$t_1 = 1 + e \cdot e$
7	$y_1 = y_0 + y_0 \cdot e$	$t_3 = q_0 + q_0 \cdot e$
8	$e_2 = a - b \cdot q_0$	
9		
10	$t_2 = 0.75 + t_1 \cdot t_1$	$q_1 = t_1 \cdot t_3$
11	$t_3 = y_1 \cdot e_2$	
12		
13		$c = a - b \cdot q_{1(r, h)}$
14	$q_1 = q_0 + t_2 \cdot t_3$	$q_{1f} = (1, 1 - .5\ ulp) \cdot q_1$
15		
16		
17	$c = a - b \cdot q_{1(r, h)}$	
18	$q_{1f} = (1, 1 - .5\ ulp) \cdot q_1$	

Table 1: Cycle for start of each microcode step

Table 1 lists the starting cycle for each step for both double and single precision. The table illustrates an unusual way of calculating the polynomial which saves several cycles. The product $t_2 \cdot t_3$ in step 14 represents the correction term

$e_2 y_0 (1 + e + e^2 + e^3 + e^4 + e^5)$ which is calculated as

$e_2 \cdot y_0 \cdot (1 + e) \cdot (0.75 + (0.5 + e^2)^2)$.

If the Power3 had a true 3-stage pipeline, both single and double precision would be one step shorter. However, an unrounded result from the third stage cannot be forwarded to both operands of the multiplier. Therefore, the t_1 and t_2 steps must each wait an additional cycle. But analysis of the double precision algorithm using a dependence graph shows that this only results in one additional cycle.

5. Square Root Algorithm

Description and derivation. The algorithm for square root differs from the divide algorithm in several ways. The divide algorithm used a simple Taylor series and a 128 word table containing short words of 13 bits each. The square root algorithm uses a series similar to one described in [14] by Ito et al but uses economized Chebyshev coefficients. The lookup table has only 32 words, but each word contains two values totaling 65 bits, including the implied ones.

Let $q = \sqrt{b}$, where b is a double precision IEEE floating point number. The five high order mantissa bits after the implied one bit specify an interval of mantissa values and are used to access the lookup table. Each of the 32 words has two values, q_0 and y_0. Each q_0 is a 12-bit approximation of $\sqrt{b}$ for the value of b at the midpoint of the corresponding interval, when the biased exponent is even. The corresponding y_0 is a full 53-bit rounded approximation of $1/(q_0)^2$. When the biased exponent is odd, we multiply q_0 by $\sqrt{2}$. This extra step adds one cycle to the algorithm for the Power3, but eliminates the need for separate table values which results in some area savings. No additional computations are needed for y_0, however. Although its values for this range are halved, that only affects its exponent which is obtained directly from the exponent of b. From now on, we will use q_{0s} and y_{0s} to indicate that they have been scaled to correspond to the least significant exponent bit.

Let us define $e = 1 - by_{0s}$, where $|e| < 2^{-6}$. We also define $e_1 = q_{0s} \cdot q_{0s} - b$. The value e_1 will be used as the residual term which the correction term will contain as a factor. From our previous definitions, we can see that $e_1 y_{0s} = e$, except for roundoff error. The calculations of e and e_1 both depend on the accumulate function for the required accuracy. Then

$$q = q_{0s} \cdot \sqrt{1 - \frac{e_1}{(q_{0s})^2}} \cong q_{0s} \sqrt{1 - (e_1 \cdot y_{0s})} \quad . \qquad (6)$$

The function $\sqrt{1 - (e_1 \cdot y_{0s})}$ can be approximated using a finite polynomial in $e_1 y_{0s}$ as $(1 + e_1 y_{0s} \cdot P(e_1 y_{0s}))$. This polynomial will be described later. The error thus introduced is the approximation error of the algorithm. The use of y_{0s} in place of $1/(q_{0s})^2$ also introduces a small error which we will later account for. Therefore,

$$q_1 = q_{0s} + q_{0s} \cdot e_1 \cdot y_{0s} \cdot P(e_1 \cdot y_{0s}) \qquad (7)$$

where q_1 represents the properly rounded value of q. Just as we had in the divide algorithm, we have an initial approximation q_{0s} and a product representing a correction term which is several orders of magnitude smaller.

For computing the polynomial, we can use e in place

of $e_1 y_{0s}$, which allows its calculation to begin three cycles earlier. However, we need to use e_1 in the correction term product to compensate for any rounding error in q_{0s}. Therefore, the final equation which we use for the algorithm, whose steps are shown in Table 2, is

$$q_1 = q_{0s} + q_{0s} \cdot e_1 \cdot y_{0s} \cdot P(e) \quad (8)$$

where $P(e) = c_0 + c_1 e + c_2 e^2 + c_3 e^3 + c_4 e^4 + c_5 e^5$. (9)

The Taylor series expansion for $\sqrt{1-e}$ would give values of -1/2, -1/8, -1/16, -5/128, -7/256, and -21/1024 for the coefficients c_0 through c_5. However, we instead use a Chebyshev approximation to reduce the degree of the polynomial, thus minimizing the number of cycles. We also used separate sets of coefficients for positive and negative values of *e*. The values for q_0 and y_0 in the table correspond to the values of *b* at the midpoints of each interval, which is where the sixth fraction bit is a one followed by all zeros. Therefore, when bit six is zero, *e* is usually positive (in some cases it might be just slightly negative due to using rounded values in the tables), and we select the coefficients for positive values of *e*. Similarly, when bit six is one, we use the coefficients for negative values of *e*.

If the Taylor series coefficients were used in place of the Chebyshev coefficients, then two additional terms would be needed in *P*(*e*), and the calculation would take two additional cycles.

For single precision, we only need $P(e) = c_0 + c_1 e$, but the two Chebyshev coefficients are not the same as for double precision. Also, since a residual term is not needed as a factor in the correction term, the single precision calculation is $q_1 = q_{0s} + q_{0s} \cdot e \cdot P(e)$. (10)

The steps shown in Table 2 for the microcode are the same as they would be for a true 3-stage pipelined unit.

Brief analysis. We now show that e_1 compensates for the rounding error in the initial approximation by replacing the first term in (8) with $q_{0s} + r$, where *r* is its rounding error. Although q_{0s} is also a factor in the correction term of (8), the effect of r will not be included at this time since we will later include it with other rounding errors which occur in calculating the correction term. The residual term becomes

$e_1 = (q_{0s})^2 - b + 2r \cdot q_{0s}$ after ignoring the r^2 term. Next, after substitutions and regrouping in (8), we get

$$q_1 = q_{0s} + (y_{0s}(q_{0s})^2 - b y_{0s}) \cdot q_{0s} \cdot P(e) + r + 2r y_{0s} \cdot (q_{0s})^2 \cdot P(e) \quad (11)$$

For the last product, which is quite small, we can use the approximations $y_{0s} \cdot (q_{0s})^2 = 1$ and $P(e) = -1/2$. This product then becomes -*r*, compensating for the +*r* term. For the remaining terms, we will make further substitutions. Since y_{0s} is a 53 bit approximation of $(q_{0s})^2$, then $y_{0s}(q_{0s})^2 = 1 + u$, where *u* is a very small value due to roundoff errors of both q_{0s} and y_{0s} . After making this substitution and then replacing $1 - b \cdot y_{0s}$ with *e*,

$$q_1 = q_{0s} + q_{0s} \cdot eP(e) + q_{0s} \cdot uP(e) \quad . \quad (12)$$

But $eP(e) = -1 + \sqrt{b y_{0s}} + t$, where *t* is the truncation error of *P(e)*. So,

$$q_1 = q_{0s}\sqrt{b y_{0s}} + q_{0s} \cdot uP(e) + q_{0s} \cdot t$$

But since $q_{0s}\sqrt{y_{0s}} \cong 1 + u/2$, we get

$$q_1 = \sqrt{b} + (u/2)\sqrt{b} + q_{0s} \cdot uP(e) + q_{0s} \cdot t \quad (13)$$

Now we use $\sqrt{b} = q_{0s} + q_{0s} \cdot eP(e)$ in the second term only, and use $P(e) = -1/2 - e/8$, ignoring all higher order powers of *e*, and get

$$q_1 = \sqrt{b} - (3/8 \cdot q_{0s} \cdot ue) + q_{0s} \cdot t \quad . \quad (14)$$

Thus we arrive at the desired result with two error terms which we can evaluate. The maximum absolute value of *u* can be determined by calculating *u* for all 64 possible values of q_{0s} , and is $3.1 \cdot 2^{-53}$. The relative error of the first term is therefore less than $(3/8) \cdot 3.1 \cdot 2^{-6} \cdot 2^{-53}$ which corresponds to .018 ulp of q_1. Since the values of q_{0s} , *u* and *e* are not independent, a more rigorous analysis would result in a smaller value. The second error term is the algorithmic error due to the finite polynomial, and can be shown to be less than .012 ulp of q_1.

Each of the rounding errors which occur during the calculation of the correction term contribute no more than 2^{-7} ulp, since the magnitude of $eP(e)$ is less than 2^{-7}. Only the 6 steps shown starting in cycles 3, 8, 9, 10, 13 and 14 of Table 2 need be included. The values calculated in the other steps are all used only with higher powers of *e*. Thus, the total of all errors is no more than 0.08 ulp, which is comfortably within the required 1/4 ulp.

We have a more rigorous analysis which shows that the error is actually less than 1/16 ulp. This permits us to use a rounding algorithm that uses four bits below the least significant bit of q_1 prior to rounding. This was used in the 620 processor, a predecessor of the Power3. The maximum observed error in simulation was 1/40 ulp.

Furthermore, both the 620 and the Power3 processors have undergone a great deal of rigorous testing using both random and directed testcases.

Cy	DP Square Root	SP Square Root
1	Table lookup q_0, y_{0s}	Table lookup q_0, y_{0s}
2	$e = 1 - b \cdot y_{0s}$	$e = 1 - b \cdot y_{0s}$
3	$q_{0s} = q_0 \cdot (1 \text{ or } \sqrt{2})$	$q_{0s} = q_0 \cdot (1 \text{ or } \sqrt{2})$
4		
5	$t_3 = c_4 + c_5 \cdot e$	$t_1 = c_0 + c_1 \cdot e$
6	$t_4 = c_2 + c_3 \cdot e$	$q_{0e} = q_{0s} \cdot e$
7	$esq = e \cdot e$	
8	$t_5 = c_0 + c_1 \cdot e$	
9	$e_1 = q_{0s} \cdot q_{0s} - b$	$q_{1t} = q_{0s} + q_{0e} \cdot t_1$
10	$t_1 = y_{0s} \cdot q_{0s}$	$q_{1u} = q_{0s} + q_{0e} \cdot t_1$
11	$t_6 = t_4 + esq \cdot t_3$	
12		
13	$q_{0e} = t_1 \cdot e_1$	$c = b - q_{1t} \cdot q_{1u}$
14	$t_7 = t_5 + esq \cdot t_6$	$q_{1f} = (1, 1 - .5 \text{ ulp}) \cdot q_1$
15		
16		
17	$q_{1t} = q_{0s} + q_{0e} \cdot t_7$	
18	$q_{1u} = q_{0s} + q_{0e} \cdot t_7$	
19		
20		
21	$c = b - q_{1t} \cdot q_{1u}$	
22	$q_{1f} = (1, 1- .5 \text{ ulp}) \cdot q_1$	

Table 2: Cycle for start of each microcode step

6. Comparison with the Newton-Raphson Algorithm for Different Pipeline Lengths

We will do our comparisons with the implementations of the Newton-Raphson algorithm that were used in the Power and Power2 processors. They were optimized for the same architecture, and used lookup tables of similar size. They are described by Markstein in [11]. Similar variations are described by Markstein, et al [15] for a processor having a different architecture. These algorithms provide full IEEE compatible results for all rounding modes, without using any special rounding algorithms. Therefore, we have also determined what the timings would be for the Newton-Raphson algorithms using the rounding algorithm available in the Power3. The Power and Power2 processors only provided double precision implementations, but here too we have determined what the timings would be for single precision.

The Newton-Raphson divide implementation uses a table for the reciprocal which provides 9 bits of precision. It computes the quotient by iteratively refining guesses of the quotient and the reciprocal of the divisor. The calculations for the quotient and reciprocal are interleaved to reduce delays due to dependency. Some shortcuts not described in the references reduce the number of steps required.

The Newton-Raphson square root implementation as described in [11] uses a table for the reciprocal of the square root that provides 8 bits of precision. It interleaves calculations for refining guesses of the square root with those for guesses of its reciprocal. This too is different from most other implementations.

The timings shown in the tables will correspond to the number of steps required in the microcode, including wait steps. This also corresponds to the number of cycles from the start of the instruction to when another independent instruction could begin. The actual latency which corresponds to when another dependent instruction could begin would be (n-1) cycles longer, where n is the number of stages in the pipeline. For n greater than 3, all of the timings are linear functions of n.

Table 3 shows the timings for the different implementations of divide and for different pipeline lengths. The heading N-R+ corresponds to the Newton-Raphson but with the special 2-bit rounding algorithm used in the Power3. It is apparent that the series algorithm gives better timings in all cases, and that for longer pipelined units, the difference is much greater. Nevertheless, we show that the Newton-Raphson would also be greatly improved by the special rounding algorithm, and for single precision, would have the same timings.

	Double Prec			Single Prec		
	Pwr3	NR	NR+	Pwr3	NR	NR+
1 cy pipe	11	15	12	8	10	8
2 cy pipe	13	16	14	10	11	10
3 cy pipe	17	23	19	13	15	13
4 cy pipe	21	30	24	16	19	16
5 cy pipe	25	37	29	19	23	19
n cy pipe	4n+5	7n+2	5n+4	3n+4	4n+3	3n+4

Table 3: Performance comparison for divide

The timings for square root shown in table 4 show a much greater benefit from use of the series algorithm.

	Double Prec			Single Prec		
	Pwr3	NR	NR+	Pwr3	NR	NR+
1 cy pipe	16	19	17	9	13	11
2 cy pipe	18	24	22	11	18	16
3 cy pipe	22	35	31	14	26	22
4 cy pipe	26	46	40	17	34	28
5 cy pipe	31	57	49	20	42	34
for	n > 3	n > 1				
n cy pipe	5n+6	11n+2	9n+4	3n+5	8n+2	6n+4

Table 4: Performance comparison for square root

For floating point units such as in the earlier processors where additional cycles are required for long normalization shifts, the series algorithms provide even more performance improvement. Such cases arise in the error calculations used for correcting a previous estimate. In the Power3 double precision algorithms, both the divide and square root each have two such calculations. This does not include the special difference calculation used for rounding, since the data result itself is not needed and thus we can inhibit normalization. On the other hand, the Newton-Raphson algorithm has five such calculations for divide and seven for square root.

7. Conclusions

We have described new algorithms for divide and square root which are based on power series approximations. These algorithms have fewer dependent operations than the Newton-Raphson algorithms, and therefore provide better performance, especially for floating point units having more stages in their pipeline.

The algorithms have been subjected to extensive testing, first with simulation and later after hardware became available. Recently, we verified the algorithms for the difficult rounding cases described in [17].

References

[1] R. R. Oehler and R. D. Groves, "IBM RISC System/6000 processor architecture, "IBM J.Res.Dev., vol.34, no.1, pp.23-36, Jan.1990.

[2] T. N. Hicks, R. E. Fry and P. E. Harvey, "POWER2 floating-point unit: Architecture and implementation," IBM J.Res.Dev., vol.38, no.5, pp.525-536, Sept.1994.

[3] R. K. Montoye, E. Hokenek and S. L. Runyon, "Design of the IBM RISC System/6000 floating-point execution unit, "IBM J.Res.Dev., vol.34, no.1, pp.59-70, Jan.1990.

[4] E. Hokenek, R. K. Montoye, "Leading-zero anticipator (LZA) in the IBM RISC System/6000 floating-point execution unit, "IBM J.Res.Dev., vol.34, vo.1, pp.71-77, Jan.1990.

[5] M. S. Schmookler and D. G. Mikan Jr, "Two State Leading Zero/One Anticipator (LZA)," U.S. Patent no. 5,493,520, Feb.1996.

[6] M. S. Schmookler, A. A. Bjorksten and D. G. Mikan Jr, "Fast Floating Point Result Alignment Apparatus," U.S.Patent no. 5,764,549, June 1998.

[7] D. Harris, S. Oberman M. Horowitz, "SRT Division Architectures and Implementations," in Proc.13th IEEE Symp. on Computer Arithmetic, pp.18-25, July 1997.

[8] J. E. Robertson, "A new class of digital division methods," IRE Trans.Electronic Computers, vol.EC-7, pp.218-222, Sept. 1958.

[9] M. J. Flynn, "On Division by Functional Iteration," IEEE Trans. Computers, vol.19, no.8, Sept.1958.

[10] P. Soderquist and M. Leeser, "An Area/Performance Comparison of Subtractive and Multiplicative Divide/Square Root Implementations," in Proc.12th IEEE Symp.on Computer Arithmetic, pp.132-138, July 1995.

[11] P. W. Markstein, "Computation of elementary functions on the IBM RISC System/6000 processor," IBM J.Res.Dev., vol.34, no.1, pp.111-119, Jan.1990.

[12] S. F. Anderson, J. G. Earle, R. E. Goldschmidt and D. M. Powers, "The IBM System/360 Model 91: Floating-Point Execution Unit," IBM J.Res.Dev., vol.11, no.1, pp.34-53, Jan.1967.

[13] "IEEE Standard for Binary Floating-Point Arithmetic," ANSI/IEEE Standard 754-1985.

[14] M. Ito, N. Takagi and S. Yajima, "Efficient Initial Approximation and Fast Converging Methods for Division and Square Root," in Proc. 12th IEEE Symp. Computer Arithmetic, pp.2-9, July 1995.

[15] A. H. Karp, P. Markstein and D. Brzezinski, "Floating Point Arithmetic Unit Using Modified Newton-Raphson Technique for Division and Square Root," U.S.Patent no. 5,515,308, May 1996.

[16] R. C. Agarwal, J. W. Cooley, F. G. Gustavson, J. B. Shearer, G. Slishman and B. Tuckerman, "New Scalar and Vector Elementary Functions for the IBM System/370," IBM J. Res. Dev., vol. 30, no. 2, pp.126-144, Mar. 1986.

[17] M. Cornea-Hasegan, "Proving the IEEE Correctness of Iterative Floating-Point Square Root, Divide, and Remainder Algorithms," Intel Technology Journal, 2nd Quarter 1998.

[18] R. C. Agarwal, A. A. Bjorksten, and F. G. Gustavson, "Method and System for Performing Floating-Point Division Using Selected Approximation Values," U.S. Patent no. 5,563,818, Oct. 1996.