

# High-Performance Floating Point Divide

Albert A. Liddicoat and Michael J. Flynn

Computer Systems Laboratory

Stanford University, Stanford, CA 94305

[liddicoat@stanford.edu](mailto:liddicoat@stanford.edu) and [flynn@umunhum.stanford.edu](mailto:flynn@umunhum.stanford.edu)

## Abstract

*In modern processors floating point divide operations often take 20 to 25 clock cycles, five times that of multiplication. Typically multiplicative algorithms with quadratic convergence are used for high-performance divide. A divide unit based on the multiplicative Newton-Raphson iteration is proposed. This divide unit utilizes the higher-order Newton-Raphson reciprocal approximation to compute the quotient fast, efficiently and with high throughput. The divide unit achieves fast execution by computing the square, cube and higher powers of the approximation directly and much faster than the traditional approach with serial multiplications. Additionally, the second, third, and higher-order terms are computed simultaneously further reducing the divide latency. Significant hardware reductions have been identified that reduce the overall computation significantly and therefore, reduce the area required for implementation and the power consumed by the computation. The proposed hardware unit is designed to achieve the desired quotient precision in a single iteration allowing the unit to be fully pipelined for maximum throughput.*

## 1 Introduction

Division can be expressed as the product of the dividend, and the reciprocal of the divisor,  $q = a/b = a \times (1/b)$ . Multiplicative techniques such as Newton-Raphson and series expansion algorithms are often used to compute the reciprocal for high-performance division [10]. The IBM *PowerPC<sup>TM</sup>* and *Power2<sup>TM</sup>* processors use Newton-Raphson algorithms to implement divide and square root. The *AMD – K7<sup>TM</sup>* [9] and IBM *Power3<sup>TM</sup>* [2] processors use algorithms based on series expansion for both divide and square root.

Typically, the first-order Newton-Raphson iteration with quadratic convergence is used. The first-order Newton-Raphson iteration requires two dependent multiplications per iteration. Rabinowitz [11] extended the Newton-

Raphson reciprocal recurrence to include higher-order polynomials. The convergence of the higher-order iteration is  $E_{i+1} = bE_i^{k+1}$ , where  $E_i$  is the error of the reciprocal approximation for iteration  $i$  and  $k$  is the order of the recurrence [5].

Series expansion algorithms are also used to compute the reciprocal using multiplicative iterations. The binomial series expansion technique, often called Goldschmidt's algorithm [6] [4], is based on the familiar Taylor series expansion of a function at a point. The binomial expansion algorithm requires two independent multiplications per iteration and provides quadratic convergence.

Recent work in the area of high-performance division has shown that higher-order iterations improve performance. Wong and Flynn [12] proposed a very-high radix division scheme that is based on look-up tables and Taylor series approximations for the reciprocal. Higher-order terms of the Taylor series are computed to increase the precision of successive quotient approximations. This approach offers linear convergence while retiring 10 or more bits per iteration. Ito, Takagi, and Yajima [7] developed an accelerated higher-order Newton-Raphson division and square root algorithm suitable for implementation using a multiply-accumulate unit. This implementation accelerates the convergence of the higher-order iteration by using a lookup table to estimate the cube of an intermediate value. Ercegovac, Lang, Muller, and Tisserand [3] proposed a method to compute the reciprocal and other functions based on argument reduction and series expansion. This method uses tables and small multipliers to compute the terms of a series expansion. Small serial multipliers are used to compute the square and cube of an intermediate value.

A multiplicative divide unit based on the higher-order Newton-Raphson reciprocal approximation is proposed and analyzed. A parallel cubing unit proposed by Liddicoat and Flynn [8] exposes additional computational parallelism. The parallel cube computation is extendable to compute higher powers and thus further accelerate the convergence of the approximation. The proposed divide unit exploits the computation parallelism exposed by the parallel powering

units. Furthermore, by using higher-order approximations the desired precision may be obtained in a single iteration allowing a fully pipelined implementation.

The various divide algorithms and implementations differ on several accounts. First, the inherent computational parallelism that allows latency reduction. Second, the sub-unit precision and the affect of the subunit precision on the latency, area, and power consumption required for the divide computation. Finally, the error convergence of the approximation determines whether it is feasible to compute the quotient to the desired precision in a single iteration.

This paper is organized as follows, sections 2 and 3 present the Newton-Raphson and binomial series expansion divide algorithms. In section 4 the higher-order Newton-Raphson divider and subunits are proposed. In section 5 significant hardware reductions applicable to the proposed architecture are presented and the final hardware configuration is discussed. In the remaining sections, the proposed divide unit is compared to alternate techniques and brief conclusions are presented.

## 2 Newton-Raphson Divide Unit

The first-order Newton-Raphson reciprocal approximation with quadratic convergence is expressed as,  $X_{i+1} = X_i(2 - bX_i)$ . The initial approximation,  $X_0$ , for the reciprocal of  $1/b$  is generally determined using a ROM lookup table before the first iteration begins. A fused multiply-subtract subunit may be used within the iteration to compute  $(2 - bX_i)$  in a single operation. Therefore, each iteration requires two dependent multiplication operations.

After the final iteration has completed, the quotient is determined by multiplying the dividend with the reciprocal of the divisor. Figure 1(a) illustrates the first-order Newton-Raphson divide unit. Each multiplication within the iteration is dependent on the result produced by the previous multiplication and must be computed serially. If  $L$  iterations are required to achieve the desired quotient precision, then the latency of the divide unit implemented with a single multiplier is  $t_{div} = t_{lookuptable} + 2Lt_{mult} + t_{mult}$ . Using two multipliers the total latency may be reduced by one multiplication if the final multiplication with the dividend is overlapped in the last iteration since  $q = (aX_f) \times (2 - bX_f)$ . The latency for the first-order Newton-Raphson divide unit using two multipliers is  $t_{div} = t_{lookuptable} + 2Lt_{mult}$ .

The generalized Newton-Raphson reciprocal iteration may be expressed as the following  $k_{th}$  order iteration,  $X_{i+1} = X_i(1 + (1 - bX_i) + (1 - bX_i)^2 + \dots + (1 - bX_i)^k)$ . Here  $X_i$  is the  $i_{th}$  approximation of the reciprocal of the divisor,  $b$ . Figure 1(b) shows a third-order Newton-Raphson divide unit designed using standard multiplication, addition, and subtraction units. The subtraction and additions may be

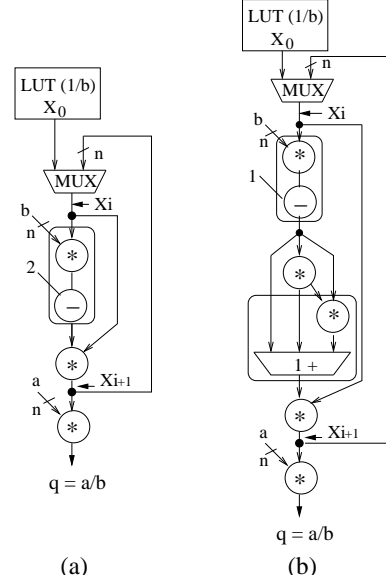


Figure 1. NR divide (a)1st order (b)3rd order.

fused with the multiplications as described previously without significantly increasing the multiplication latency.

If a single multiplier is used and  $L$  iterations are required for the desired quotient precision, then the latency of the third-order Newton-Raphson divide unit is  $t_{div} = t_{lookuptable} + 4Lt_{mult} + t_{mult}$ . Due to the faster convergence, one iteration of the third-order divide unit reduces the reciprocal error by the same amount as two iterations of the first-order divide unit. The latency for one iteration of the third-order divide unit is also equivalent to the latency for two iterations of the first-order divide unit.

As was the case with the first-order divide unit, the final multiplication with the dividend may be overlapped when two multipliers are available reducing the latency by one multiplication. The latency of the divide unit using two multipliers is  $t_{div} = t_{lookuptable} + 4Lt_{mult}$ . Again the latency for one iteration of the third-order divide unit is equivalent with the latency of two iterations of the first-order Newton-Raphson iteration. There is no benefit in using a higher-order iteration if full precision serial multiplications are used to compute the powers of  $(1 - bX_i)$ .

## 3 Division by Series Expansion

The disadvantage with the standard form of the Newton-Raphson divide is that the multiplications in the iteration are dependent and must be performed serially. Therefore, each iteration requires two or more serial multiplications. Binomial series expansion is another multiplicative division technique with quadratic convergence. The typical form of the series expansion recurrence is based on the Maclaurin

series were  $b = 1 + X$  as shown in equation 1.

$$g(X) = \frac{1}{b} = \frac{1}{1+X} = 1 - X + X^2 - X^3 + X^4 - \dots \quad (1)$$

After factoring equation 1 and multiplying by the dividend  $a$ , the quotient,  $q$ , can be expressed by the multiplicative series shown in equation 2.

$$q = \frac{a}{b} = a(1 - X)(1 + X^2)(1 + X^4)(1 + X^8)(1 + X^{16})\dots \quad (2)$$

Each multiplication in equation 2 quadratically reduces the error in the quotient  $q$  and is considered an iteration towards the final quotient. Here,  $q_0 = a$ ,  $q_1 = a * (1 - X)$ , and  $q_{i+1} = q_i(1 + X^{2^i}) = q_i \times r_i$  for  $i \geq 1$ .

Let  $r_0 = (1 - X)$ ,  $d_0 = (1 + X)$  and  $d_i = d_{i-1} * r_{i-1}$  then  $r_i = (2 - d_i)$  for  $i \geq 1$ . A multiplication and subtraction must be performed to obtain the next factor  $r_i$ . Within each iteration both  $d_i$  and  $r_i$  must be computed, and a fused multiply-subtract cannot be used.

Figure 2 shows a divide unit based on the iterative form of the binomial series expansion. The right side of the divide unit computes the next factor  $r_{i+1}$  while the left side computes the quotient approximation  $q_{i+1}$ . The factor  $r_{i+1}$  is independent of the quotient,  $q_{i+1}$ , computation and therefore the two multiplications may occur simultaneously.

Similarly to the Newton-Raphson division, a lookup table can be used to reduce the number of iterations required to obtain the desired precision. The first term  $(1 - X)$ , or product of the first few terms  $(1 - X)(1 + X^2)(1 + X^4)\dots(1 + X^{2^m})$ , is found in a ROM lookup table. Then  $d_{m+1}$  is computed by multiplying the result returned from the lookup table by  $(1 + X)$ . The initial quotient approximation for  $q_{m+1}$  is computed by multiplying the dividend,  $a$ , by the result returned from the lookup table such that  $q_{i+1} = a \times (1 - X)(1 + X^2)(1 + X^4)\dots(1 + X^{2^m})$ . These two multiplications are also independent and may occur in parallel. Then the iterations continue as before.

The latency of the binomial expansion divide algorithm depends on how many multipliers are used and the number of iterations,  $L$ , required to obtain the desired quotient precision. If one multiplier is used, then the divide unit latency is  $t_{div} = t_{lookuptable} + 2Lt_{mult} + t_{mult}$ . Here, the subtract must be overlapped with the quotient multiplications. If two multipliers are used then the latency reduces to  $t_{div} = t_{lookuptable} + t_{mult} + L(t_{2'sComp} + t_{mult})$ .

Interestingly, if a single multiplier is used the binomial expansion divide unit latency is equivalent to that of the Newton-Raphson divide unit. However, if two multipliers are used then the latency of the binomial expansion divider is reduced by approximately,  $L(t_{mult} - t_{2'sComp}) - t_{mult}$ .

It has been shown that the first-order Newton-Raphson algorithm and the binomial expansion algorithms are equivalent when  $X_0 = b - 1$ . In fact, they are two different ways of expressing the same computation. Both algorithms require the same number and type of operations. However, due to the way each algorithm is expressed, the multiplications in the Newton-Raphson iteration are dependent and must be performed serially while the two multiplications in the binomial expansion are independent and can be performed in parallel.

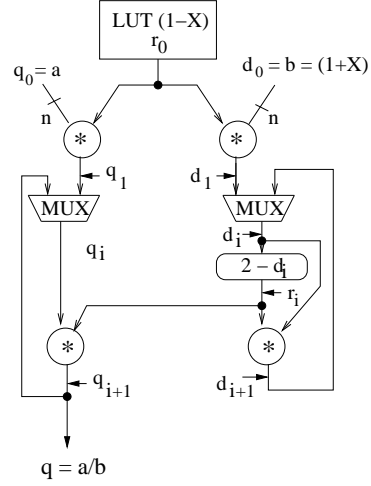


Figure 2. Binomial expansion divide unit.

## 4 Proposed Divide Architecture

The divide unit may compute the quotient directly and need not iterate to a solution. The error in the initial approximation and the convergence of the computation must be sufficient to guarantee the desired quotient precision will be achieved with one computation. This joint constraint implies that there is a tradeoff between the lookup table size and the computational complexity of the algorithm.

Next, we express the quotient directly as the product of the dividend and the  $k_{th}$ -order Newton-Raphson reciprocal approximation,  $q = \frac{a}{b} = aX(1 + (1 - bX)^1 + (1 - bX)^2 + \dots + (1 - bX)^k)$ . Here  $X$  is the initial estimation of  $\frac{1}{b}$  generally found in a lookup table with error,  $E_{lookuptable}$ . The quotient error is expressed as,  $E_q = bE_{lookuptable}^{k+1}$ . The  $k_{th}$ -order approximation increases the number of bits of precision of  $X$  by a factor of  $k + 1$ . Therefore, in order to compute an  $n$ -bit reciprocal in a single iteration the precision of the initial approximation must be  $\frac{n}{k+1}$  bits. The lookup table size must be  $(2^{\frac{n}{k+1}}) \times (\frac{n}{k+1})$  bits. For example, the third-order Newton-Raphson approximation would require a  $2^{\frac{n}{4}} \times \frac{n}{4}$  bit lookup table.

Figure 3 illustrates the hardware structure required to implement the proposed higher-order divide unit. The latency of the divide unit is  $t_{div} < t_{lookuptable} + 3t_{mult}$ . Here, it is assumed that the powers of  $(1 - bX)$  may be computed directly, in parallel, and faster than a full precision multiplication. Liddicoat and Flynn [8] propose a parallel cubing unit and describe a parallel squaring unit suitable for the proposed higher-order divide architecture. These powering units are described in more detail in subsection 4.3.

The proposed divide unit may be fully pipelined if two small multipliers, the powering units, and one full multiplier are used. Small multipliers are used to compute  $(1 - bX)$  and  $(aX)$  since  $X$  is approximately  $\frac{n}{k+1}$  bits in length. A third-order divide unit would be constructed out of two  $\frac{1}{4}$  size multipliers, one squaring unit, one cubing unit, and one full multiplier. A 24-bit implementation of the proposed third-order divide unit is presented and discussed in detail in the following subsections.

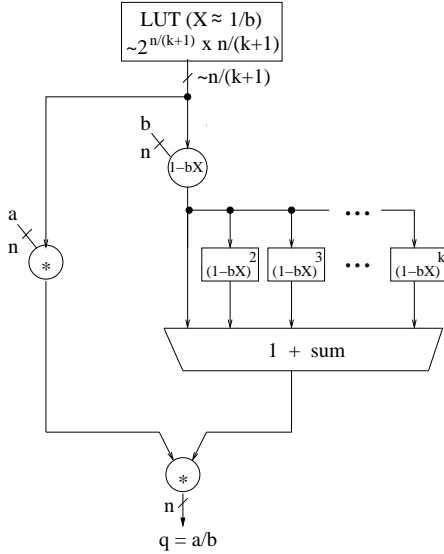


Figure 3. Proposed higher-order NR divide.

#### 4.1 Lookup table

An initial approximation for the reciprocal of the divisor is determined by a table lookup. The  $h + 1$  most significant bits of the divisor are used to index the lookup table and the  $m + 1$  most significant bits of the reciprocal approximation are returned from the lookup table. A  $2^h \times m$  bit ROM with  $h$  address bits and an  $m$ -bit word size is used for the lookup table since the most significant bit is constant for normalized IEEE floating point operands [1].

In order to eliminate the need to represent negative numbers in the computation, the lookup table must be programmed such that the  $(1 - bX)$  fused multiply-subtract

always produces a positive result. Furthermore, if the result of the  $(1 - bX)$  computation is positive then the computed reciprocal will be equal to or less than the true reciprocal. This can be demonstrated by realizing that the exact reciprocal can be computed using an infinite-order Newton-Raphson approximation. If  $(1 - bX)$  is positive then the exact reciprocal is proportional to an infinite sum of decreasing positive terms and a finite-order approximation is the truncation of this infinite series.

To guarantee that  $(1 - bX) \geq 0$ , the lookup table must return a value that when multiplied by  $b$  is always less than or equal to one. This can be accomplished by programming the table entries such that the value stored in each lookup table address is less than the reciprocal for all possible values of  $b$  that map to that particular address location. Let  $b_{trunc}$  be equivalent to the divisor  $b$  truncated to  $h + 1$  bits. To determine the value to store in each address of the lookup table we first add  $2^{-h}$  to  $b_{trunc}$  so that  $b_{trunc} + 2^{-h} > b$  for any possible  $b$ . The reciprocal of  $b_{trunc} + 2^{-h}$  is then computed noting that  $\frac{1}{b_{trunc} + 2^{-h}} < \frac{1}{b}$ . Finally the result of  $\frac{1}{b_{trunc} + 2^{-h}}$  is truncated to  $m + 1$  bits. All of the table entries are of the form  $0.1xxx...xxx$ . The first digit to the right of the radix point is always one and therefore does not need to be stored in the lookup table. If the lookup table is programmed according to this procedure, then it will always return an approximation less than  $\frac{1}{b}$  and the result of the  $(1 - bX)$  fused multiply-subtract operation is always positive. The value to be programmed in each address of the lookup table are a function of the address, the number of bits used to address the lookup table  $h$ , and the number of output bits from the table  $m$ .

To determine the best lookup table size we exhaustively simulated several table sizes using full precision computations. The results from these simulations are shown in table 1. The table size of  $2^7 \times 7$  bits was selected for the reciprocal approximation since the maximum error was less than  $0.5ulps$  (unit in the last place). Furthermore, the number of leading zeros to the right of the radix point in the  $(1 - bX)$  computation is guaranteed to be six or more when using a lookup table of this size.

Table 1. Lookup table sizes for 24-bit operand

Ad bits	Wd bits	Tbl Size	Lead 0's	Error
6	6	384	5	1.40 ulps
6	7	448	5	0.95 ulps
7	6	768	5	0.53 ulps
*7	7	896	6	0.13 ulps

## 4.2 Computing $(1 - bX)$

The first arithmetic operation that follows the lookup table access is the  $(1 - bX)$  fused multiply subtract. Since  $X$  is approximately  $\frac{n}{k+1}$  bits, a small multiplier is used.

The divisor  $b$  is in the normalized IEEE single precision format. Therefore,  $b_{23} = 1$  and bits  $b_{22}$  through  $b_0$  depend on the value stored in  $b$ . In order to compute the result of  $(1 - bX)$ ,  $b$  is sign extended with eight leading zeros and then the two's complement of  $b$  is taken to produce  $-b$ . The sign extended  $-b$  is represented by 8 leading 1's, then one zero, followed by the complement of bits  $b_{22}$  to  $b_0$  and an additional one is added to bit  $b_0$  to complete the two's complement. The bits from  $X$  are used to select partial products of  $-b$ . Lastly the constant "1.0" is added to the partial product array (PPA) to account for the 1 that  $-bX$  is subtracted from to form  $(1 - bX)$ . Figure 4 illustrates the hardware unit required to calculate  $(1 - bX)$ .

Recall that exhaustive simulation indicates that the leading six bits to the right of the radix point will always be zero. These bits do not have to be computed, further reducing the hardware needed for the  $(1 - bX)$  computation. The boxed area in figure 4 indicates the columns in the PPA that need to be summed to compute the 24 most significant bits of  $(1 - bX)$ . The PPA can be reduced using a Wallace tree structure in four CSA delays. The area required to implement the PPA for the  $(1 - bX)$  unit is approximately 30% of the size of a 24-bit direct multiplier partial product array.

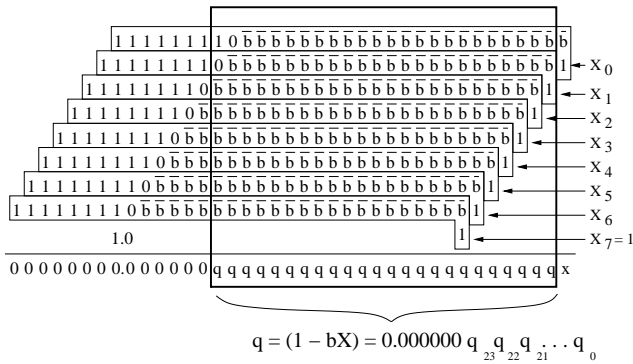


Figure 4.  $(1 - bX)$  fused multiply-subtract unit.

## 4.3 Computing the powers of $(1 - bX)$

A squaring circuit that computes the square of a 24-bit operand 25% faster with slightly less than half the area required to perform a 24-bit direct multiplication is used to compute  $(1 - bX)^2$ . Figure 5 illustrates the squaring unit partial product array reduction.

Similarly the cube may be computed directly and concurrently with the square using the cubing unit proposed by

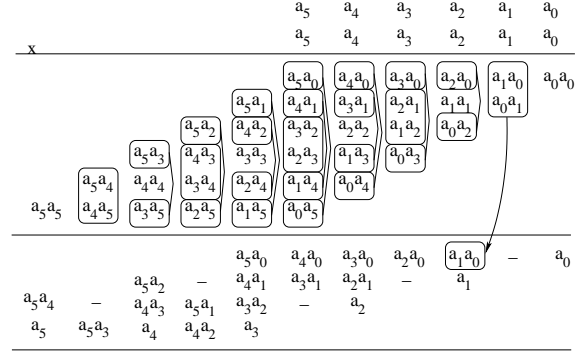


Figure 5. Squaring unit PPA reduction using

$$2 * a_i a_j = a_i a_j + a_j a_i.$$

Liddicoat and Flynn [8]. Figure 6 shows the partial product array required to compute the parallel cube for a 4-bit operand. Figure 6 also identifies three reductions that may be applied to reduce the size of the PPA. The terms from the 1st reduction have a weight of one while terms from the 2nd and 3rd reductions have a weight of three. The terms with a weight of three are summed in carry save fashion using a Wallace Tree. Then the three times multiple is computed and summed with the 1X terms using a carry free addition stage.

The precise cube of the 24-bit input produces a 72-bit result. The exact cube is not needed to achieve the desired quotient precision for the divide unit. In section 5, truncations of the reduced cube PPA are studied. These reductions not only decrease the area requirement but also the latency of the cube operation. The reduced parallel cube is approximately 63% faster than computing the cube using direct multipliers and requires only about 10% of the area that is required by a single  $n$ -bit direct multiplier.

The parallel cubing unit is easily extendable to compute higher-order powers of  $(1 - bX)$ . Using higher-order powers of  $(1 - bX)$  will accelerate the convergence of the Newton-Raphson approximation and reduce the precision needed by the initial reciprocal estimate  $X$ .

## 4.4 Computing the Final Multiplications

The multiplication of  $a \times X$  is a small multiplication since  $X$  is approximately  $\frac{n}{k+1}$  bits. The multiplication area is about 30% that of a full 24-bit direct multiplication. A slow area efficient multiplier may be used since the result of this computation is not needed until after the powers of  $(1 - bX)$  have been computed. The final multiplier computes the product of  $(aX)$  with the sum of powers of  $(1 - bX)$  to produce the final quotient. This is the only full multiplication that is required by the proposed divide unit.

				$a_3$	$a_2$	$a_1$	$a_0$
$x$				$a_3$	$a_2$	$a_1$	$a_0$
$x$				$a_3$	$a_2$	$a_1$	$a_0$
				$a_3a_0a_0$	$a_2a_0a_0$	$a_1a_0a_0$	$a_0a_0a_0$
				$a_3a_0a_1$	$a_2a_0a_1$	$a_1a_0a_1$	$a_0a_0a_1$
				$a_3a_1a_0$	$a_2a_1a_0$	$a_1a_1a_0$	$a_0a_1a_0$
				$a_3a_2a_0$	$a_2a_2a_0$	$a_1a_2a_0$	$a_0a_2a_0$
				$a_3a_1a_1$	$a_2a_1a_1$	$a_1a_1a_1$	$a_0a_1a_1$
				$a_3a_0a_2$	$a_2a_0a_2$	$a_1a_0a_2$	$a_0a_0a_2$
				$a_3a_3a_0$	$a_2a_3a_0$	$a_1a_3a_0$	$a_0a_3a_0$
				$a_3a_2a_1$	$a_2a_2a_1$	$a_1a_2a_1$	$a_0a_2a_1$
				$a_3a_1a_2$	$a_2a_1a_2$	$a_1a_1a_2$	$a_0a_1a_2$
				$a_3a_0a_3$	$a_2a_0a_3$	$a_1a_0a_3$	$a_0a_0a_3$
				$a_3a_3a_1$	$a_2a_3a_1$	$a_1a_3a_1$	$a_0a_3a_1$
				$a_3a_2a_2$	$a_2a_2a_2$	$a_1a_2a_2$	$a_0a_2a_2$
				$a_3a_1a_3$	$a_2a_1a_3$	$a_1a_1a_3$	$a_0a_1a_3$
				$a_3a_3a_2$	$a_2a_3a_2$	$a_1a_3a_2$	$a_0a_3a_2$
				$a_3a_2a_3$	$a_2a_2a_3$	$a_1a_2a_3$	$a_0a_2a_3$
				$a_3a_3a_3$	$a_2a_3a_3$	$a_1a_3a_3$	$a_0a_3a_3$
① 1X	$a_3$	-	-	$a_2$	-	$a_1$	-
② 3X	$a_3a_2$	$a_3a_1$	$a_3a_0$	$a_2a_1$	$a_2a_0$	$a_1a_0$	$a_0a_0$
③ 3X	$a_3a_2a_1$	$a_3a_1a_0$	$a_3a_0a_0$	$a_2a_1a_0$	$a_2a_0a_0$	$a_1a_0a_0$	$a_0a_0a_0$

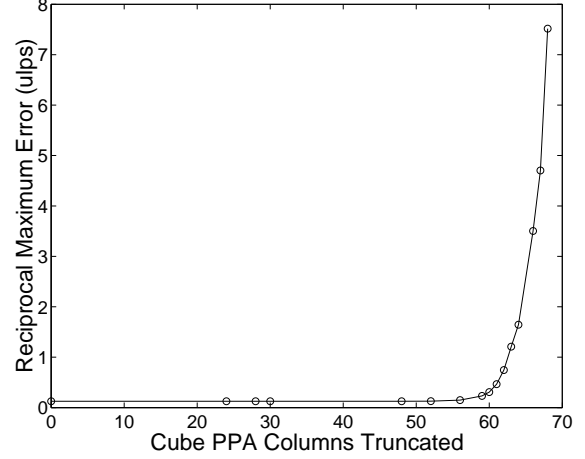
**Figure 6. Parallel cubing unit PPA reductions, (1)  $a_i = a_i a_i a_i$ , (2)  $3 * a_j a_i = (a_j a_i a_i + a_i a_i a_j + a_i a_j a_i)$ , and (3)  $6 * a_k a_j a_i = (a_k a_i a_j + a_k a_j a_i + a_j a_k a_i + a_j a_i a_k + a_i a_k a_j + a_i a_j a_k)$ .**

## 5 The Final Hardware Configuration

The number of bits in a multiplier PPA, increase by the square of the operand length  $n^2$ , while the squaring unit area grows by  $\frac{1}{2}n^2$  and the cubing unit grows by  $\frac{1}{6}n^3$ [8]. Therefore, effort must be made to minimize the intermediate operand length and the required output precision from each subunit. Significant reductions in the hardware area required to implement the subunits are presented in this section.

The divide unit has been exhaustively simulated to determine the maximum error in the reciprocal computation for various truncations of the cubing unit PPA. Figure 7 indicates the reciprocal maximum error in terms of ulps for varying truncations of the cubing unit PPA. A sharp knee exists in the curve when 60 or more of the least significant columns have been truncated. Only the eight most significant columns of the cubing unit PPA are required to achieve an error of less than 0.5 ulps.

Additionally, the divide unit has been exhaustively simulated to determine the maximum error in the reciprocal for various truncations of the squaring unit given that 59, 60 and 61 columns have been truncated from the cubing unit PPA. Similarly, a sharp knee exists in the curve plotting the reciprocal error versus the number of least significant



**Figure 7. The reciprocal error versus the cubing unit PPA column truncation (24-bit).**

columns truncated from the squaring unit. Truncating up to 29 columns from the squaring unit PPA does not significantly increase the reciprocal error.

The reciprocal accuracy is less than 0.5 ulps for the design points listed in Table 2. Design 2 was selected since the maximum PPA height and number of bits in the PPA are minimized. Furthermore, Design 2 achieves the smallest maximum error. Therefore, the least significant 31 columns of the squaring unit PPA and the least significant 60 columns in the cubing unit PPA may be truncated. The PPA area for the squaring unit is less than 15% of the size of a 24-bit direct multiplier while the PPA area for the cubing unit is less than 10% of the size of a 24-bit direct multiplier. Since the squaring and cubing units have been significantly reduced, the latency of these units is less than that of a single multiplier. In fact the cube computation is 63% faster than can be computed using serial multipliers.

Finally, the divide unit has been exhaustively simulated to determine the maximum error in the reciprocal for various truncations of the  $(1 - bX)$  multiply unit given that the cubing unit PPA has been truncated by 60 columns and the squaring unit by 31 columns. The least significant three columns of the  $(1 - bX)$  multiply-subtract unit may be trun-

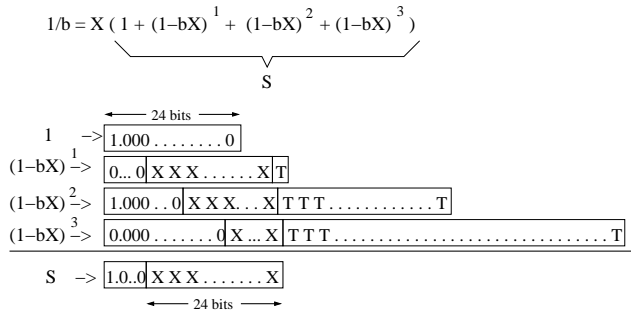
**Table 2. Reciprocal error for truncated units.**

	Cb trunc	Sqr trunc	Err(ulp)	PPA ht	PPA bits
1	59	32	0.43	16	139
*2	60	31	0.42	12	140
3	61	29	0.49	12	147

cated while maintaining an error of less than 0.5 ulps. A maximum reciprocal error of 0.496 ulps is achieved with the cubing unit truncated by 60 columns, the squaring unit truncated by 31 columns, and the  $(1-bX)$  multiply-subtract unit truncated by 3 columns.

Let's re-examine the third-order Newton-Raphson reciprocal approximation  $\frac{1}{b} = X * (1 + (1-bX) + (1-bX)^2 + (1-bX)^3) = XS$  where  $S = (1 + (1-bX) + (1-bX)^2 + (1-bX)^3)$ . In figure 8 the bit fields for each of the four components of  $S$  have been aligned. In this figure the X's represent bits that will be computed by columns in the PPA for each unit and the T's indicate columns that may be truncated from the PPA for each unit. From this diagram it is clear that most of the bits form the  $(1-bX)$  multiply subtract unit will contribute to the 24 most significant bits of  $S$ , while only about  $\frac{1}{3}$  of the columns from the squaring unit and approximately  $\frac{1}{8}$  of the columns from the cubing unit contribute to the 24 most significant bits of  $S$ . Progressively less computation is required to achieve the higher-order terms of the Newton-Raphson iteration.

The design that was presented in the preceding discussion was selected to minimize latency and hardware area under the constraint of computing the reciprocal to less than 0.5 ulps error. By slightly increasing the number of columns in the square and cube computations, the worst-case reciprocal error will be improved. The lookup table precision required depends on the order of the Newton-Raphson iteration and the precision of the sub-unit computation. Increasing the number of columns in the sub-unit PPAs will decrease the lookup table precision needed to achieve a given worst-case error. The lookup table area may be reduced by 50% for each bit of precision that it is reduced. Therefore, the designer may trade off computational complexity for area or vice versa.



**Figure 8.**  $3^{rd}$  order Newton-Raphson approximation sub-unit precision.

## 6 Summary

Division by functional iteration utilizes multiplication as the fundamental operation. We presented the standard Newton-Raphson reciprocal iteration and the binomial series expansion reciprocal iteration. The computational parallelism in these approaches is limited to two parallel multiplications.

We proposed a divide unit architecture based on the higher-order Newton-Raphson reciprocal iteration. The divide unit uses truncated squaring, cubing and powering units. A 24-bit third-order implementation was used as an example to describe the divide unit in detail.

We found that the first  $(1-bX)^1$ , second  $(1-bX)^2$ , third  $(1-bX)^3$  and fourth  $(1-bX)^4$  order computations require progressively less precision. Reducing the precision of the higher-order computations will maximize the efficiency, reduce the latency and minimize the power consumption of the overall computation. The reduction in the precision of the higher-order computations in the proposed architecture differs from the typical Newton-Raphson or series expansion approach. The later algorithms require full precision computation after the first iteration.

Furthermore, truncating the subunits significantly reduces the area required to implement the divider. The  $(1-bX)$  fused multiply-subtract unit, squaring unit, cubing unit, and the  $(aX)$  multiply unit require respectively 30%, 15%, 10%, and 30% of the area that is required by a direct multiplier. If the divider is designed with separate units then the entire implementation would be less than the size of two full precision multipliers. The final multiplication may be performed on a shared multiplier further reducing the dedicated hardware requirement by the divide unit.

A 53-bit IEEE double precision divide unit was also designed and tested. The same design techniques were applied to the double precision unit. To reduce the lookup table size the highest-order 5 columns of the  $(1-bX)^4$  parallel computation were included. Adding a few additional columns for the  $(1-bX)^4$  terms only increased the 1X PPA by a total of 8 bits and the 3X PPA by a total of two bits. Using a  $2^{14} \times 14$  bit lookup table the 53-bit reciprocal can be computed in one iteration with the squaring unit and cubing unit truncated to approximately 14% and 12% of the size of a 53-bit direct multiplier. A second 53-bit design was studied using a lookup table of  $2^{13} \times 14$  bits, half the size of the previous 53-bit design point. The truncated squaring unit was approximately 14% of the size of a 53-bit direct multiplier and the truncated cubing unit was approximately 25% of the size of a 53-bit direct multiplier. The 53-bit designs were proportionally very similar to the results for the 24-bit designs indicating that the proposed architecture scales well over the studied range.

Table 3 summarizes the area requirements of the mul-

**Table 3. Divide area comparison (IEEE DP)**

Algorithm	Lookup Tbl Size	HW Area
N-R 1 <sup>st</sup> -order	28,672B	2 Mult.
Series Expansion	28,672B	2 Mult.
N-R 3 <sup>rd</sup> -order	28,672B	2 Mult.
Ito... [7]	61,440B	1 Mult-Acc.
Erc... [3]	65,536B	≈ 2 Mult.
Proposed Arch.	14,336B	≈ 2 Mult.

**Table 4. Divide latency comparison**

Algorithm	Iter.	Comp. Latency
N-R 1 <sup>st</sup> -order	2	1SM+3FM
Series Expansion	2	1SM+2Sub+2FM
N-R 3 <sup>rd</sup> -order	1	1SM+3FM
Ito... [7]	4	4FMAC
Erc... [3]	1	2SM+2SNM+1SDA+1FM
Proposed Arch	1	2SM+1FM

multiplicative divide techniques for IEEE double precision operands. The lookup table requires most of the dedicated area needed to implement the divide unit. Table 4 summarizes the latencies for the multiplicative division algorithms discussed. In the table the following abbreviations are used; SM=small multiply  $\frac{n}{4} \times n$ , SNM=small narrow multiply  $\frac{n}{4} \times \frac{n}{4}$ , FM=full multiply  $n \times n$ , SUB=subtract  $n - n$ , FMAC=full multiply accumulate  $n \times n - n$ , SDA=3 operand signed digit adder. The proposed divide unit has the lowest latency and area requirements. Additionally, the number of iterations required by each algorithm to achieve an error reduction of  $bE_{initial}^4$  is listed. The implementations listed in Table 4 and 5 were selected for comparison since each one requires the area of approximately two multipliers or less.

The proposed architecture is easily amenable to a fully pipelined implementation. Since the quotient is computed in a single pass through the subunits. A new divide operation can be dispatched each cycle. This differs from the first-order Newton-Raphson iteration and binomial series expansion technique that require multiple iterations to achieve the same convergence as the proposed architecture. Our algorithm and the Ercegovac, Lang, Muller, Tisserand approach are fully pipeline-able without a significant increase in hardware.

## 7 Conclusions

A fast, efficient, and high-throughput divide unit is proposed. This unit utilizes the higher-order Newton-Raphson

reciprocal approximation. Parallel squaring, cubing and powering units perform low latency concurrent computation and reduce the overall latency of the divide unit. It has been demonstrated that progressively less computation is required to compute the second, third and higher-order terms. Therefore, significant hardware reductions are achievable by truncating the powering unit partial product arrays. The proposed architecture achieves the desired precision in a single iteration and is amenable to a fully pipelined implementation that dispatches one divide instruction per cycle. Designing an optimal divide unit for a specific operand length requires balancing the subunit precision and lookup table size.

## References

- [1] ANSI/IEEE Std 754-1985, IEEE Standard for Binary Floating-Point Arithmetic, 1985.
- [2] R. C. Agarwal, F. G. Gustavson, and M. S. Schmookler. Series Approximation Methods for Divide and Square Root in the *Power3<sup>TM</sup>* Processor. In *Proc. 14th IEEE Symp. on Computer Arithmetic*, pages 116–123, April 1999.
- [3] M. D. Ercegovac, T. Lang, J.-M. Muller, and A. Tisserand. Reciprocation, Square Root, Inverse Square Root, and Some Elementary Functions Using Small Multipliers. *IEEE Transactions on Computers*, 49(7):628–637, July 2000.
- [4] M. D. Ercegovac, D. W. Matula, J.-M. Muller, and G. Wei. Improving Goldschmidt Division, Square Root, and Square Root Reciprocal. *IEEE Transactions on Computers*, 49(7):759–763, July 2000.
- [5] M. Flynn. On Division by Functional Iteration. *IEEE Transactions on Computers*, C-19(8):702–706, August 1970.
- [6] R. E. Goldschmidt. Applications of Division by Convergence. Master's thesis, Dept. of Electrical Engineering, Massachusetts Institute of Technology, Cambridge, Mass., June 1964.
- [7] M. Ito, N. Takagi, and S. Yajima. Efficient Initial Approximations and Fast Converging Methods for Division and Square Root. In *Proc. 12th IEEE Symp. on Computer Arithmetic*, pages 2–9, July 1995.
- [8] A. Liddicoat and M. Flynn. The Parallel Square and Cube Computation. In *IEEE 34th Asilomar Conference on Signals, Systems and Computers*, October 2000.
- [9] S. F. Oberman. Floating Point Division and Square Root Algorithms and Implementation in the AMD-K7 Microprocessor. In *Proc. 14th IEEE Symp. on Computer Arithmetic*, pages 106–115, April 1999.
- [10] S. F. Oberman and M. Flynn. Division Algorithms and Implementations. *IEEE Transactions on Computers*, 46(8):833–854, August 1997.
- [11] P. Rabinowitz. Multiple-Precision Division. In *Communications of the ACM*, volume 4, page 98, February 1961.
- [12] D. Wong and M. Flynn. Fast Division Using Accurate Quotient Approximations to Reduce the Number of Iterations. In *IEEE Transactions on Computers*, pages 981–995, August 1992.