An Accurate, High Speed Implementation of Division by Reciprocal Approximation

D. L. Fowler and J. E. Smith

Astronautics Corporation of America
5800 Cottage Grove Rd.
Madison, Wisconsin 53716

Abstract

Newton-Raphson reciprocal approximation is a practical method for implementing high-speed division. It provides quadratic convergence and is based on multiplier technology which can take advantage of extensive hardware parallelism. While unlimited accuracy is theoretically possible, it is very important to minimize the number of iteration steps to improve performance and/or to reduce hardware requirements. Consequently, there is an important accuracy/speed/cost tradeoff in reciprocal approximation implementations.

This paper discusses a reciprocal approximation implementation with special attention given to the tradeoffs just mentioned. An interpolation method is used to insure that an initial approximation, held in a ROM table, is as accurate as possible. A new method for implementing the iteration steps is given. Special instructions are used so that maximum accuracy can be carried between iteration operations. For 64-bit floating point operands (53-bit mantissa), a table lookup and only two iterations are required. Meanwhile high accuracy is maintained. The rounded reciprocal rarely differs from a true "round to nearest" value based on an infinite precision result. When the results do differ (less than once every 1000 calculations), the difference in accuracy is shown to be less than 0.025 of a least significant bit (LSB).

Introduction

Of the four primary floating point operations, division is the least frequently used and the most difficult to compute. Generally speaking, division occurs three to four times less frequently than multiplication and takes three to four times as long to compute in most implementations. Furthermore, division algorithms tend to be difficult to pipeline due to the dependencies inherent in selecting quotient bits.

Given the above constraints, along with others such has hardware cost and chip "real estate", a good engineering solution to the problem of high speed floating point division is to first find the reciprocal of the divisor, and then multiply by the dividend. Such "reciprocal approximation" methods are typically based on the Newton-Raphson iteration method [1].

The Newton-Raphson iterative division technique starts with an initial approximation of the reciprocal of the divisor, usually implemented through a look-up table. Then an iterative method is used to form successive approximations where the error decreases as the square of the previous approximation. Therefore, with a sufficiently accurate starting approximation, the algorithm quickly converges to the reciprocal with desired accuracy. After the reciprocal is found it can be multiplied with the dividend to form the final quotient. (In fact the multiplication with the dividend can be done at an intermediate stage to improve overlap of the operations[2]).

Newton-Raphson division methods use multiplication as a basic operation; this leads to two advantages for high performance implementations.

(1) Hardware complexity is reduced because much of the division hardware can be shared with the multiplier.

(2) Division is more easily pipelined because multiplication is easily pipelined.

Division in the IBM 360/91 [3] used the Newton-Raphson method and used a significant amount of the multiplier hardware. In the CRAY-1 [2] (and all subsequent Cray Research processors) division is implemented via a series of instructions consisting of a reciprocal approximation instruction, an iterate instruction, and two multiplication instructions. This not only reduces hardware requirements, but it permits all the functional units to accept a new operand every clock period (an important factor in keeping vector control hardware simple). The Floating Point Systems array processors [4], perform division by reciprocal approximation, with most of the steps being done in software. A similar approach is used in the recently-announced Intel i860 [5]. In the Intel i860, a single chip processor, saving chip real estate was very likely a key factor in the decision to use reciprocal approximation for division.

A disadvantage of typical hardware implementations of the Newton-Raphson iteration method is that accuracy is less than with a direct division method. Theoretically, accuracy equivalent to other division methods can be attained if enough iteration steps are used and if enough bits of precision are maintained throughout. However, in the interest of speed and hardware cost, real implementations tend to minimize iteration steps and bits of precision. A small amount of accuracy is traded for speed and hardware cost savings. This paper examines the Newton-Raphson division method initially used in the Astronautics ZS-1 computer systems. In the ZS-1 special steps are taken to reduce losses in accuracy, while maintaining the speed and cost advantages of the method.

The Astronautics ZS-1

The Astronautics ZS-1 [6] is a high performance 64-bit computer, designed for scientific and engineering applications. Floating point addition, multiplication, and division are implemented with parallel pipelined functional units. In the prototype and beta systems, the functional units are built from standard TTL SSI and MSI circuits and 16 x 16 bit VLSI fixed point multiplier chips. Each of the three floating point units fits on a 18 x 18 inch printed wiring board (PWB).

The ZS-1 uses IEEE standard floating point formats [7], but in the interest of higher performance, not all aspects of IEEE standard arithmetic are followed. In the initial ZS-1 systems, floating point division is done by reciprocal approximation using a variation of the Newton-Raphson method described in this paper[1].

Reciprocal Approximation Algorithm

Because the final multiplication required to form the quotient is done with an ordinary floating point multiplication instruction, we consider only the problem of forming the reciprocal of a divisor.

We form the reciprocal of a floating point number, B, whose mantissa is b. The exponent arithmetic is very straightforward, and we concentrate on finding the reciprocal of b. The recursive solution for reciprocation of a floating point mantissa b using the Newton-Raphson method is:

$$X_{i+1} = X_i(2 - b \times X_i)$$

[1] After the ZS-1 prototypes were made operational, a new floating point unit was designed and built using a VLSI floating point chip set. With this new design, the Newton-Raphson method was dropped in favor of the direct division method provided by one of the VLSI chips.

X_0 is an initial approximation of $\frac{1}{b}$ and X_{i+1} is a successive approximation beginning with X_i.

The ZS-1 uses the following steps to form the reciprocal of b:

(1) X_0 is found from a ROM look-up table in the reciprocal approximation unit.

(2) bX_0 is formed by doing a multiplication with dedicated hardware in the reciprocal approximation unit.

(3) $(2 - bX_0)$ is the two's complement of bX_0 $(2 > bX_0 > 0.5)$.

(4) $X_1 = X_0(2 - bX_0)$ is formed by doing a multiplication with dedicated hardware in the reciprocal approximation unit.

(5) bX_1 is formed using the Multiplication Unit.

(6) $(2 - bX_1)$ is the two's complement of bX_1 $(2 > bX_1 > 0.5)$

(7) $X_2 = X_1(2 - bX_1)$ is formed using the Multiplication Unit.

X_2 is the final approximation of $1/b$.

Execution of the division is implemented as follows. The *Reciprocal Approximation* instruction does the first four steps. Step 1 of the Newton-Raphson reciprocal approximation algorithm is done by a ROM look-up table on the Reciprocal Approximation Unit. Steps 2 - 4 are done on the same unit to generate an iteration on the look-up value. Steps 5 - 6 are done using the Multiplication Unit by a special *Iterate 1* instruction. Step 7 is done using the Multiplication Unit by a special *Iterate 2* instruction. Finally, the reciprocal is multiplied by the dividend to form the quotient using the Multiplication Unit via the normal multiply instruction.

In terms of the above algorithm, the two major enhancements provided by the ZS-1 for improving accuracy are:

(1) The way in which the ROM table look-up is done to form X_0. Since the Newton-Raphson algorithm has quadratic convergence, the accuracy of the initial approximation is squared upon each iteration. Therefore, special steps are taken to make the absolute value of the initial error as small as possible; any extra initial accuracy is amplified by later steps.

(2) In contrast to previous implementations of this algorithm, such as in the Cray Research processors, the instructions *Iterate 1* and *Iterate 2* are not implemented as simple floating point multiplications where the two's complement of the result of *Iterate 1* is formed before being used in the *Iterate 2*. In the ZS-1, the result of the *Iterate 1* instruction is stored in a special format to maintain high accuracy between the *Iterate 1* and *Iterate 2* instructions.

This paper concentrates on the forming of the reciprocal based on the mantissa. Calculation of the exponent is handled in a straightforward manner. In forming $1/B$ the exponent portion is simply $ER = -EB$, where EB is the unbiased exponent of B and ER is the unbiased exponent of the reciprocal $R = 1/B$. Of course the exponent must be adjusted along with the mantissa if normalization is needed and the bias value added in.

ZS-1 Rounding

In a practical high-speed implementation of reciprocal formation, the primary problem we are faced with is maintaining the accuracy of reciprocals when rounding is performed. The most accurate floating point rounding is achieved if the rounded result is the same as if the operation was carried out to infinite precision prior to rounding. This is a stated goal of IEEE standard arithmetic. We will refer to this type of rounding as **infinite precision rounding.** As a practical matter, operations can typically be carried to some finite precision to get the same result as if infinite precision were used. For forming a reciprocal, this means that the precision of the result before rounding must be twice the precision of the input operand and final result. For a 53-bit mantissa, this implies that 106 bits are needed prior to rounding.

In the ZS-1 design, we made the decision to maintain fewer than 106 bits of precision prior to rounding. This saved one complete iteration step, reduced the time to form a reciprocal by about 50 percent, and reduced the width of the parallel multiplier used for the final iteration step.

Using fewer bits of precision prior to rounding does mean that the results will not always agree with infinite precision rounding. On the other hand, we carefully control the error at each step of the algorithm so that after the final iteration and rounding are done, the result rarely differs from infinite precision rounding as defined by the IEEE standard. When it does differ, the difference in accuracy between the two methods is shown to be very slight.

The ZS-1 is architecture supports four different rounding modes.

(1) "Round to nearest" - The result is as close to the exact reciprocal as the floating point representation and implementation allows.

(2) "Round to positive infinity" - The result is guaranteed to be greater than or equal to the exact reciprocal.

(3) "Round to negative infinity" - The result is guaranteed to be less than or equal to the exact reciprocal.

(4) "Round to zero" - The result is guaranteed to be between zero and the exact reciprocal. (This is the same as round to $+\infty$ for negative numbers and round to $-\infty$ for positive numbers.)

It is important to note that we are willing to diverge from the IEEE standard rounding algorithms in the interest of improving performance and reducing hardware requirements. For example, in the infinity modes we do not insist that the result be the closest representable floating point result that bounds the correct result; we only insist that it bounds the correct result. The analysis of this divergence is included in this paper.

Implementation

This section describes how the reciprocal approximation is realized in hardware. Throughout this description, b represents the mantissa of the floating point number B discussed earlier, and q represents the mantissa of the number which approximates $1/B$. The initial approximation X_0 and first iteration X_1 are both done on the special purpose Reciprocal Approximation Unit. A block diagram of the hardware is shown in Fig. 1. The final approximation X_2 is done in two parts on the Multiplication Unit. A block diagram of the Multiplication Unit is shown in Fig. 2.

Initial Approximation

The initial approximation X_0 is done through a ROM look-up table. The size of the table is 32k by 16 bits. The 15 most significant bits (MSBs) of the mantissa, $b_1 b_2 \cdots b_{14} b_{15}$, excluding the hidden one, are used to do the initial approximation look up.

Since the table is addressed with b (the mantissa bits of B), where $2 > b \geq 1$, the resulting reciprocal X_0 will have the range $0.5 < X_0 \leq 1$. However, as explained below, the equation implemented the ROM look-up table limits the range of X_0 to $0.5 \leq X_0 < 1$, and therefore the form of a table entry will be $0.1 q_2 q_3 \cdots q_{15} q_{16}$.

To generate the ROM look-up table, rather than merely truncating the input after b_{15}, the values in the ROM table are for inputs with a one in the 16th bit of the mantissa b_{16} (the first "unseen" bit), and the rest of the bits, $b_{17} \cdots b_{52}$ are zeros. The ROM table contains the approximate reciprocals of:

$$1.b_1 b_2 \cdots b_{14} b_{15} 1000...$$

This technique, which we refer to as "ROM interpolation", in effect "averages" the error introduced during the look-up procedure due to truncation of the input, in much the same way as rounding the output minimizes error due to truncating the output.

The resulting approximate reciprocal output is rounded back to the LSB of the table entry. The output is rounded by adding a one to the bit location just past the output width of the table. The following equation is used to generate the ROM table:

$$X_0 = \frac{1}{(b' + 2^{-16})} + 2^{-17}$$

Where $b' = 1.b_1 b_2 \cdots b_{14} b_{15}$.

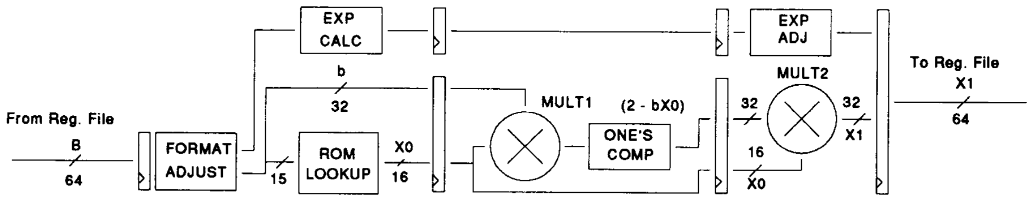

Figure 1: Reciprocal Approximation Unit Block Diagram

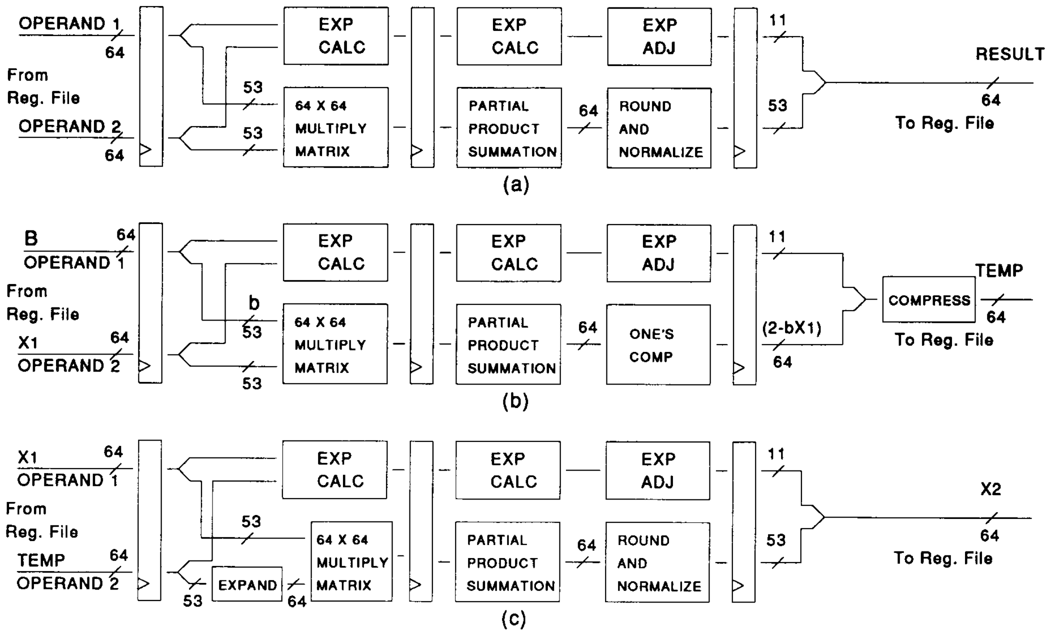

Figure 2: Multiplication Unit Block Diagram

a) Normal Multiplication
b) Iterate 1
c) Iterate 2

Determining the range of X_0 :

$$X_{0_{max}} = \frac{1}{1+2^{-16}} + 2^{-17}$$

$$X_{0_{max}} < 1$$

$$X_{0_{min}} = \frac{1}{(2-2^{-15})+2^{-16}} + 2^{-17}$$

$$X_{0_{min}} > \frac{1}{2}$$

The resulting approximation of X_0 is a number q of the form $0.q_1q_2q_3\cdots$ The 16 bits $q_1q_2\cdots q_{15}q_{16}$ are used as the entries in the look-up table. Though bit q_1 is always a one, it is still stored in the table rather than using the extra bit to store q_{17}. The reason for this is that the next two steps use the 16 MSBs of q as a multiplicand. The hardware implementations for these steps use 16 bit input VLSI multipliers which cannot take advantage of the additional bit q_{17}.

Taking into consideration the truncation of the output to 16 bits after rounding, the equation for the ROM look up function is actually:

$$X_0 = \frac{1}{(b'+2^{-16})} + [2r - (q - q')]$$

Where b' is defined above, and:

$$q = \frac{1}{(b'+2^{-16})} = 0.1q_2q_3\cdots$$

$$q' = .1q_2q_3\cdots q_{15}q_{16}$$

$$r = q_{17}.$$

Computation of X_1

The X_1 approximation is done by implementing the first iteration of the Newton-Raphson approximation in special purpose hardware. This is shown in Fig. 1.

First, $b \times X_0$ is formed by multiplying the 16 bits of X_0 by the 32 MSBs of b and rounding the result to 32 bits. The multiplication is implemented using standard 16 x 16 bit VLSI multiplier chips. For performance and space reasons, The lower 16 bits of the 48 bit product are not formed. However, a round of these bits is done into the LSB of the 32 bit product.

Next, we form ($2-b \times X_0$). Rather than generating the two's complement of $b \times X_0$ we can simply one's complement the bits of $b \times X_0$ as was done in the IBM 360/91 division algorithm [3]. This adds only a small error since the one's and two's complement differ only by an LSB. This avoids the time and hardware penalty of propagating a carry the length of the 32 bit result. This slight discrepancy introduces an error of -2^{-31}.

The next step is to multiply ($2-b \times X_0$) by X_0 to form X_1. The same multiplication scheme is used as above. However, an additional bias of 2^{-31} is added to the resulting product. The reason is to reduce the absolute error of X_1 as is explained in detail in the Error Analysis section. X_1 is the final result of the Reciprocal Approximation instruction and is accurate to approximately 31 bits in the mantissa. These bits are concatenated with 22 zeroes to form the full 53 bit mantissa and along with the exponent are stored as a floating point number in the register file.

Computation of X_2

The X_2 approximation requires two passes through the Multiplication Unit. The first half of the final reciprocal approximation is done using the already-present floating point multiply hardware and slightly altering its function as shown in Fig. 2. Rather than doing a simple multiply, the hardware is used to do an *Iterate 1*. The second half also uses the multiply unit to do an *Iterate 2*.

The equation for the final approximation is:

$$X_2 = X_1 \times (2 - X_1 \times b).$$

In the ZS-1, *Iterate 1* is used to form $(2 - X_1 \times b)$ and this intermediate value is stored in a general purpose floating point register. This value can simply be called TEMP. The *Iterate 2* instruction multiplies TEMP by X_1 to form the final reciprocal X_2. However, the way in which TEMP is stored requires *Iterate 2* to differ from a regular multiplication because a transformation must be done on TEMP before the multiplication can take place.

Iterate 1 - First Half Approximation of X_2

The *Iterate 1* instruction uses the Multiplication Unit as shown in Fig. 2. The *Iterate 1* instruction uses b and X_1 as its arguments to generate TEMP. First b is multiplied by X_1 in the normal fashion. The result of this multiplication is a 64 bit number. Of the 106 bits which result from a 53 x 53 multiply, 42 of the lower order bits of the product are not formed for space and performance reasons. The effect this has on the reciprocal calculation is shown in the Error Analysis section.

Next ($2-X_1 \times b$) is formed. Again this is done by doing a one's complement, which adds little appreciable error.

Now, we note that TEMP, before being stored to a general purpose register, should be a number close to one since it is an approximation of

$2 - X_1 \times b$ ($X_1 \approx 1/b$). In fact it has the range:

$$1 + 2^{-29} + 2^{-33} + 2^{-63} > TEMP > 1 - 2^{-29} - 2^{-62}.$$

The derivation of this is shown in the Error Analysis section. Equivalently, TEMP is represented in binary has one of the two forms:

$$1.0000000000000000000000000000x...$$

or

$$0.1111111111111111111111111111x...,$$

where x is either 0 or 1.

If the normalized mantissa of TEMP is $1.m_1 m_2 m_3 \cdots m_{62} m_{63}$ then bits m_1 through m_{27} are largely redundant. These bits are either all 0's or all 1's. Bits m_{28} through m_{63} contain the useful information needed by the *Iterate 2* instruction to form the final approximation.

If the result of TEMP were stored as a normal floating point product, bits m_{53} through m_{63} would be lost, and therefore the error would increase. However, instead of storing the result in the normal fashion as:

s	$e_{10} \cdots e_0$	$m_1 m_2 \cdots \cdots m_{51} m_{52}$

It is stored as:

s	$e_{10} \cdots e_0$	$m_{17} m_{18} \cdots m_{62} m_{63} 00000$

The redundant bits m_{17} through m_{27} can be used to reconstruct the actual TEMP since m_1 through m_{16} are always the same.

Note that 16 bits are "compressed" to make room for storing the internal multiplier result. Though there are only 11 additional bits of information available, a shift of 16 bits is implemented. The multiplier used in the ZS-1 is only 11 bits more accurate than the register storage. If the multiplier had even more internal accuracy available, other redundant bits could be compressed.

Though not used in the ZS-1 implementation, the additional redundant bits include m_{17} through m_{26} (m_{27} must be kept to do the expansion). Also, 10 bits of the exponent are redundant since its only possible values are 2^0 (when TEMP ≥ 1) or 2^{-1} (when TEMP < 1) unbiased. Finally, the sign bit is always a zero (positive). In all there are 26 redundant mantissa bits, 10 redundant exponent bits plus the sign bit for a total of 37 redundant bits. So, up to 37 bits could be used for storage of normally "lost" accuracy due to storing to a general purpose register.

Iterate 2 - Second Half Approximation of X_2

The *Iterate 2* instruction uses X_1 and TEMP as arguments to form the result X_2 which is the final approximation of $1/b$. This is shown in Fig. 2.

Iterate 2 first expands the mantissa of TEMP back into a 64 bit number. Then the expanded TEMP is multiplied by X_1 to form X_2. Recall TEMP $= (2 - b \times X_1)$. Finally, X_2 is rounded based on the rounding mode back to 53 bits to be stored as the reciprocal of b. The details of this rounding are discussed in the Error Analysis section.

As a possible alternative to this approach for *Iterate 2*, the following scheme could be used. First, note that TEMP is expanded to fit the 64 bit input width of the multiplication matrix. The reason the multiplier input is 64 bits wide is that it is built from 16 x 16 bit VLSI multiplier chips. In other implementations of a floating point multiplier, a more efficient design may limit the multiplicand widths to the width of the mantissa. In such a situation, the *Iterate 2* instruction can be realized in a different manner.

For cases where the redundant bits in TEMP are zeros, TEMP is equal to $1 + ((2 - b \times X_1) - 1) \times 2^n$, where n = number of bits shifted out by compression in *Iterate* 1.

Rather than expanding TEMP prior to multiplication, simply multiply it by X_1. The correct product is generated by adjusting the bit positions of the partial products prior to their summation to compensate for the shift done during the compression of $(2 - b \times X_1)$. The redundant bits in TEMP can easily be forced to be zeros by controlling the error bias when generating X_1 and also during *Iterate 1*.

Error Analysis

Here, each operation done to generate the reciprocal approximation X_2 is analyzed. All the errors introduced in each step are considered and their effect on the final result is shown. This error analysis is used to determine the best rounding methods. In each analysis, the largest possible error is calculated in both the positive and negative direction rather than just the largest absolute error. The largest possible negative error is referred to as ε_{min}, while the largest possible positive error is called ε_{max}. Minimum error refers to the error closest to negative infinity rather than smallest absolute error.

The direction of error is maintained in the analysis because it has a bearing on the implementation of different rounding modes as is shown later.

Error Analysis of X_0

The error of the initial approximation ε_{X_0} is simply $X_0 - 1/b$, the approximated value minus the actual value. The error analysis follows.

$$\varepsilon_{X_0} = X_0 - 1/b$$

$$= \frac{(b - b') - 2^{-16}}{(bb' + b \times 2^{-16})} - [2r - (q - q')]$$

$$= \frac{(x - 2^{-16})}{(bb' + b \times 2^{-16})} + y$$

Where:

$$2^{-15} > x \geq 0, \quad 2^{-17} > y > -2^{-17}$$

The maximum and minimum ε_{X_0} occur at $b=1$, $b'=1$.

$$\varepsilon_{X_{0\max}} < \frac{(2^{-15} - 2^{-16})}{(1 + 2^{-16})} + 2^{-17}$$

$$\varepsilon_{X_{0\max}} < 1.5 \times 2^{-16}$$

$$\varepsilon_{X_{0\min}} > \frac{(0 - 2^{-16})}{(1 + 2^{-16})} - 2^{-17}$$

$$\varepsilon_{X_{0\min}} > -1.5 \times 2^{-16}$$

Therefore, $1.5 \times 2^{-16} > \varepsilon_{X_0} > -1.5 \times 2^{-16}$.

Using ROM interpolation, by assuming m_{16} is a one when generating the ROM values, has a distinct advantage over using truncated inputs. As can be seen from the above equation, the error of the first approximation is balanced around zero. This results in the minimum absolute value of ε_{X_0}. This is important because this error value becomes squared in the next iteration as will be shown.

If we had simply assumed truncated ROM input values [1], our equation would be:

$$X_{0trunc} = \frac{1}{b'} + [2r - (q - q')]$$

$$\varepsilon_{X_{0trunc}} = \frac{(b - b')}{bb'} + [2r - (q - q')]$$

$$\varepsilon_{X_{0trunc}} = \frac{x}{bb'} + y$$

So,

$$\varepsilon_{X_{0trunc-max}} < 2^{-15} + 2^{-17} = 1.25 \times 2^{-15}$$

$$\varepsilon_{X_{0trunc-min}} > 0 - 2^{-17} = -1 \times 2^{-17}$$

Therefore,

$$2.5 \times 2^{-16} > \varepsilon_{X_{0trunc}} > -0.5 \times 2^{-16}$$

Even though the total range of the error is the same, the maximum absolute value of the error is greater.

Error Analysis of X_1

The first multiplication done in calculating X_1 introduces the following error terms:

$$0 > \varepsilon_{T1} -2^{-32}$$

$$2^{-32} > \varepsilon_{T2} > -2^{-32}$$

The first of the error terms is due to the truncation of b to 32 bits, prior to multiplication, and the second is due to the rounding of the product $b \times X_0$ to 32 bits. The only error introduced in the second multiplication is ε_{T3} which due to the rounding of the final product to 32 bits.

$$2^{-32} > \varepsilon_{T3} > -2^{-32}$$

Also, as mentioned earlier, a bias of $+2^{-31}$ is added to the product.

Writing the equation for X_1 including all the error terms:

$$X_1 = X_0 \times (2 - b \times X_0)$$

$$= (\frac{1}{b} + \varepsilon_{X_0}) \times (2 - (b + \varepsilon_{T1}) \times (\frac{1}{b} + \varepsilon_{X_0}) - \varepsilon_{T2} - 2^{-31}) + \varepsilon_{T3} + 2^{-31}$$

$$= (\frac{1}{b} + \varepsilon_{X_0}) \times (1 - \varepsilon_{X_0} \times b - \frac{\varepsilon_{T1}}{b} - \varepsilon_{T2} - 2^{-31}) + \varepsilon_{T3} + 2^{-31}$$

$$= \frac{1}{b} - (\varepsilon_{X_0})^2 \times b - \frac{\varepsilon_{T1}}{b^2} - \frac{\varepsilon_{T2}}{b} + \varepsilon_{T3} - \frac{2^{-31}}{b} + 2^{-31}$$

Where:

$$1.5 \times 2^{-16} > \varepsilon_{X_0} > -1.5 \times 2^{-16}$$

$$0 > \varepsilon_{T1} > -2^{-31}$$

$$2^{-32} > \varepsilon_{T2,}\ \varepsilon_{T3} > -2^{-32}$$

Therefore, the error for the first Newton-Raphson iteration is:

$$\varepsilon_{X_1} = X_1 - 1/b$$

$$= -b \times (\varepsilon_{X_0})^2 - \frac{\varepsilon_{T1}}{b^2} - \frac{(\varepsilon_{T2} + 2^{-31})}{b} + \varepsilon_{T3} + 2^{-31}$$

Now, we calculate the minimum and maximum values for ε_{X_1}:

$$\varepsilon_{X_{1\min}} > -b \times (\frac{2^{-16}}{b^2 + b \times 2^{-16}} + 2^{-17})^2 - 0 - \frac{(2^{-32} + 2^{-31})}{b} - 2^{-32} + 2^{-31}$$

$$> -b \times (\frac{2^{-16}}{b^2 + b \times 2^{-16}} + 2^{-17})^2 - \frac{1.5 \times 2^{-31}}{b} + 2^{-32}$$

The minimum value of this function in the range $2 > b \geq 1$ occurs at $b = 1$. Therefore,

$$\varepsilon_{X_{1\min}} > -(1.5 \times 2^{-16})^2 - (2^{-32} + 2^{-31}) + 2^{-32}$$

$$\varepsilon_{X_{1\min}} > -(2^{-30} + 2^{-34})$$

Calculating the maximum error:

$$\varepsilon_{X_{1\max}} < 0 + \frac{2^{-31}}{b^2} - \frac{(-2^{-32} + 2^{-31})}{b} + 2^{-32} + 2^{-31}$$

The maximum occurs at $b = 1$

$$\varepsilon_{X_{1\max}} < 2^{-31} + 2^{-32} - 2^{-31} + 2^{-32} + 2^{-31}$$

$$< 2^{-30}$$

Therefore the range of ε_{X_1} is:

$$2^{-30} > \varepsilon_{X_1} > -(2^{-30} + 2^{-34})$$

Error Analysis of X_2

The *Iterate 1* instruction uses b and X_1 as its arguments to generate TEMP. First we multiply b times X_1, which introduces an error of ε_{F1} since the multiplication unit is not exact. Then $(2 - X_1 \times b)$ is approximated by doing a one's complement operation. The complementation adds an error of -2^{-63}.

$$TEMP = 2 - X_1 \times b - 2^{-63} - \varepsilon_{F1}$$

Where:

$$2^{-63} > \varepsilon_{F1} > -2^{-62}$$

Now, calculating the minimum and maximum values of TEMP. TEMP should be a number close to one since it is an approximation of $2 - X_1 \times b$ $(X_1 \approx \frac{1}{b})$.

$$TEMP = 2 - X_1 \times b - 2^{-63} - \varepsilon_{F1}$$

$$= 2 - b \times (\frac{1}{b} + \varepsilon_{X_1}) - 2^{-63} - \varepsilon_{F1}$$

$$= 1 - 2^{-63} - b \times \varepsilon_{X_1} - \varepsilon_{F1}$$

Calculating $TEMP_{\max}$:

$$TEMP_{\max} < 1 - 2^{-63} - b_{\max} \times \varepsilon_{X_{1\min}} - \varepsilon_{F1_{\min}}$$

$$< 1 - 2^{-63} + 2 \times (2^{-30} + 2^{-34}) + 2^{-62}$$

$$< 1 + 2^{-29} + 2^{-33} + 2^{-63}.$$

Calculating $TEMP_{\min}$:

$$TEMP_{\min} > 1 - 2^{-63} - b_{\max} \times \varepsilon_{X_{1\max}} - \varepsilon_{F1_{\max}}$$

$$> 1 - 2^{-63} - 2 \times (2^{-30}) - 2^{-63}$$

$$> 1 - 2^{-29} - 2^{-62}$$

Since we can "compress" TEMP when storing it to a general purpose register, no additional error is introduced due to the store.

The *Iterate 2* instruction uses X_1 and TEMP as arguments to form the result X_2 which is the final approximation of $1/b$. Only one additional error (ε_{F2}) is introduced in this step, the floating point multiplication error. The range of this error is the same as ε_{F1}.

$$2^{-63} > \varepsilon_{F2} > -2^{-62}$$

Writing the equation for X_2:

$$X_2 = X_1 \times TEMP + \varepsilon_{F2}$$

$$= (\frac{1}{b} + \varepsilon_{X_1}) \times (2 - b \times (\frac{1}{b} + \varepsilon_{X_1}) - 2^{-63} - \varepsilon_{F1}) + \varepsilon_{F2}$$

$$= (\frac{1}{b} + \varepsilon_{X_1}) \times (1 - b \times \varepsilon_{X_1} - 2^{-63} - \varepsilon_{F1}) + \varepsilon_{F2}$$

$$= \frac{1}{b} - \varepsilon_{X_1} - \frac{2^{-63}}{b} - \frac{\varepsilon_{F1}}{b} + \varepsilon_{X_1} - b \times (\varepsilon_{X_1})^2$$

$$- 2^{-63} \times \varepsilon_{X_1} - \varepsilon_{F1} \times \varepsilon_{X_1} + \varepsilon_{F2}$$

Simplifying and removing the insignificant terms, we have:

$$X_2 = \frac{1}{b} - \frac{2^{-63} + \varepsilon_{F1}}{b} + \varepsilon_{F2} - b \times (\varepsilon_{X_1})^2$$

The error of X_2 (ε_{X_2}) is:

$$\varepsilon_{X_2} = X_2 - \frac{1}{b}$$

$$= \frac{-(2^{-63} + \varepsilon_{F1})}{b} + \varepsilon_{F2} - b \times (\varepsilon_{X_1})^2$$

$$= \frac{-(2^{-63} + \varepsilon_{F1})}{b} + \varepsilon_{F2} - b \times (-b \times (\varepsilon_{X_0})^2 - \frac{\varepsilon_{T1}}{b^2} - \frac{(\varepsilon_{T2} + 2^{-31})}{b}$$

$$+ \varepsilon_{T3} + 2^{-31})^2$$

$$= \frac{-(2^{-63} + \varepsilon_{F1})}{b} + \varepsilon_{F2} - b \times (-b \times (\frac{x - 2^{-16}}{bb' + b \times 2^{-16}} + y)^2 - \frac{\varepsilon_{T1}}{b^2}$$

$$- \frac{(\varepsilon_{T2} + 2^{-31})}{b} + \varepsilon_{T3} + 2^{-31})^2$$

The preceding equation contains all of the error terms introduced in the reciprocal approximation.

By not performing compression on the *Iterate 1* result, the same error equation for ε_{X_2} is obtained. However, in this case:

$$2^{-53} > \varepsilon_{F1} > -2^{-53}$$

assuming round to nearest after *Iterate 1*. ε_{F1} becomes by far the dominant error term.

Rounding

Conceptually, rounding of X_2 to fit the 53 bit mantissa width is done in two steps. First a **correction** is done to bias the error introduced during the reciprocal approximation process. Next a **round** is done to fit the mantissa to the 53 bit representation to be stored. Both the "correction" and "round" depend on the rounding mode and are done in parallel during the calculation of X_2.

To determine how the correction and round are done, we calculate the minimum and maximum values of ε_{X_2}. Recall:

$$\varepsilon_{X_2} = \frac{-(2^{-63} + \varepsilon_{F1})}{b} + \varepsilon_{F2} - b \times (\varepsilon_{X_1})^2$$

The minimum value of this function in the range $2 > b \geq 1$ occurs at $b = 1$, when $\varepsilon_{X_1} = \varepsilon_{X_{1\min}}$. Therefore,

$$\varepsilon_{X_{2\min}} > -(2^{-63} + 2^{-63}) - 2^{-62} - (2^{-30} + 2^{-34})^2$$

$$> -(2^{-60} + 2^{-61} + 2^{-63} + 2^{-68})$$

$$> -14 \times 2^{-63}$$

What this error value indicates is that after the final iteration, the result internal to the multiplication unit could be low by slightly greater than 13 but less than 14 LSBs. This is important to know when considering rounding to infinity, which must guarantee that the resulting reciprocal is greater than or equal to the true reciprocal. The correction is done by adding 14 LSBs (14×2^{-63}) to the result, which will ensure that the result is high. Then the round is done by adding ones to the 54th through 64th bits so that a one in any of these bit locations will be propagated to the 53 bit.

Calculating the maximum error:

$$\varepsilon_{X_{2\max}} < \frac{-(2^{-63} - 2^{-62})}{b} + 2^{-63} - b \times (0)^2$$

This maximum occurs at b=1,

$$\varepsilon_{X_{2\max}} < 2^{-62}$$

What this positive error indicates is that after the final iteration, the result internal to the multiplication unit could be high by no greater than 2 LSBs. This is important for rounding to negative infinity, when the result must be less than or equal to the true reciprocal. The correction for round to negative infinity can be done by subtracting 2 LSBs (2×2^{-63}) from the result, forcing the result low. Then, the round is done by merely truncating the 54th through 64th bits so that ones in any of these locations will not affect the 53rd bit.

For round to nearest, we try to make the final result as close to the actual reciprocal as possible. For the case analyzed here, we have a result X_2 which has the following error range:

$$2 \times 2^{-63} > \varepsilon_{X_2} > -14 \times 2^{-63}$$

One might think that by adding 6×2^{-63} to correct the final result prior to rounding would be the best way to round to nearest. It is true that this leads to the smallest possible absolute error, but it does not generate the correctly rounded result the greatest percentage of the time. This is because the error density through the range of ε_{X_2} is not constant. In fact, it was found through simulating this algorithm that the errors are clustered around -1×2^{-63} and drop off rapidly on either side. This is shown in Figure 3. We chose a correction which would lead to the correctly rounded result the greatest percentage of the time. So in fact the best correction for round to nearest is obtained by adding 2^{-63} to the final result before rounding the mantissa to 53 bits. Then the round is done by adding a one to 54th bit location which is one less than the LSB of the resulting mantissa.

Experimental Results

In order to simulate the algorithm presented here, a program was written in the C programming language. This was done prior to the completion of the hardware. We have since verified that the results of the simulations agree exactly with the actual hardware results.

The following table compares the theoretical error limits as derived in the Error Analysis section and errors found through the processing of approximately two million random number inputs. Generation of the random numbers was done using the C library function **random()** to form the upper and lower halves of the mantissa.

Error	Theoretical Value	Experimental Value
$\varepsilon_{X0_{max}}$	1.5×2^{-16}	1.4848×2^{-16}
$\varepsilon_{X0_{min}}$	-1.5×2^{-16}	-1.4930×2^{-16}
$\varepsilon_{X1_{max}}$	2^{-30}	0.9396×2^{-30}
$\varepsilon_{X1_{min}}$	$-(2^{-30} + 2^{-34})$	$-0.8380 \times (2^{-30} + 2^{-34})$
$\varepsilon_{X2_{max}}$	2×2^{-63}	1.2674×2^{-63}
$\varepsilon_{X2_{min}}$	-14×2^{-63}	-10.157×2^{-63}
$TEMP_{max}$	$1+2^{-29}+2^{-33}+2^{-63}$	$1+2^{-31}+2^{-32}+\cdots$
$TEMP_{min}$	$1-2^{-29}-2^{-62}$	$1-2^{-30}-2^{-31}-\cdots$

The error density prior to rounding, as found experimentally is shown in Fig. 3. As can be seen the error is clustered between -2^{-63} and -2^{-62}.

Round to Nearest

Because we do not always provide the same result as with (IEEE-like) infinite precision rounding, it is possible to generate a 53-bit mantissa which differs by one LSB from a true infinite precision rounded result. However, the difference in accuracy between our result and one with infinite precision rounding is actually much less than one significant bit.

When X_2 is generated by the *Iterate 2* instruction, the error analysis shows that the result internal to the Multiplication Unit which has 64 bits of mantissa can be high by as much a 2 LSBs and low by as much as 14 LSBs. Since the error density is not constant through this range, simulation was used to determine how to best "correct" the mantissa prior to rounding. It was found that adding one LSB (2^{-63}) generated the greatest percentage of results agreeing with the infinite precision rounded value.

For round to nearest, the correction is done by adding 2^{-63} to bias the error. The round is done by adding a one to the 54th bit. If a one is in this bit location, it will propagate up to the 53rd bit. Since the ZS-1 result can still be low by as much as 13×2^{-63} after the correction is done, the 54th bit may be a 0 while with infinite precision rounding it would be a 1. In this case, after rounding, the ZS-1 result would be one LSB less than the infinite precision rounded result. However, in the worst case, the differences in accuracies would be small. The bounding worst case would be when the 54th through 64th bits of the true reciprocal are $2^{-54} + 2^{-60} + 2^{-61}$.

In this case, the infinite precision round would add a one to the 53rd bit q_{53}, so the error of this representation due to rounding would be:

$$\varepsilon_{\infty} = 2^{-53} - (2^{-54} + 2^{-60} + 2^{-61})$$
$$= 2^{-54} - 2^{-60} - 2^{-61} \approx +0.488 \times 2^{-53}$$

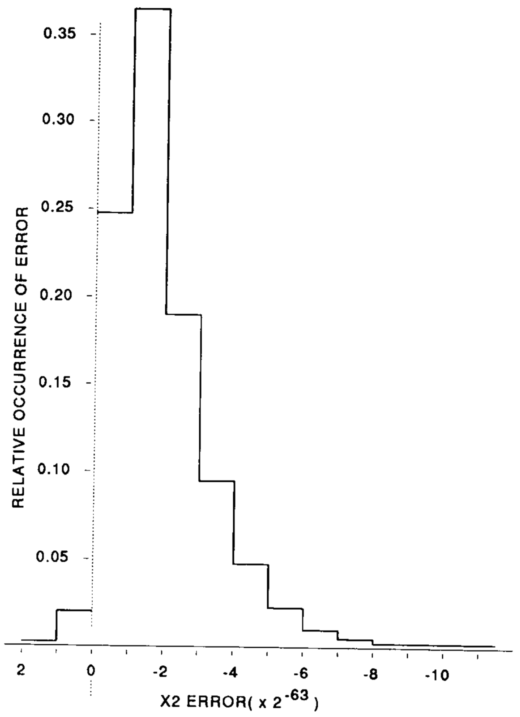

Figure 3: Reciprocal Error Internal to Multiplication Unit After Iterate 2 (Prior to Correction or Rounding)

In infrequent cases (those on the far right of the graph in Fig. 3), it would be possible for q_{54} in the ZS-1 representation to be a zero. In this case, nothing would be added to bit q_{53}, and the error of this representation would be:

$$\varepsilon_{ZS-1} = -(2^{-54} + 2^{-60} + 2^{-61}) \approx -0.512 \times 2^{-53}$$

Therefore, in the worst case, the absolute difference in accuracy between the ZS-1 round and the infinite precision round is 0.024×2^{-53}, or 0.024 of an LSB.

This is the theoretical worst case. As can be seen from Fig. 3 the instances of the ZS-1 approximation being this low are quite infrequent.

Experimentally, the maximum difference found was less than 0.015×2^{-53}, or 0.015 of an LSB.

Round Nearest Error	Theoretical Value	Experimental Value
$ABS(\varepsilon_{ZS-1} - \varepsilon_{\infty})_{max}$	0.024×2^{-53}	0.015×2^{-53}

Round to +/- Infinity, Zero

Since the *Iterate 2* result can be low by 14 LSBs prior to rounding to positive (negative) infinity, X_2 is "corrected" by adding 14×2^{-63} for positive (negative) results. And, since the *Iterate 2* result can be high by 2 LSBs prior to rounding to negative (positive) infinity, X_2 could be "corrected" by subtracting 2×2^{-63} for positive (negative) results. Actually, this case is handled by biasing the error during *Iterate 1* so that $\varepsilon_{X2} \le 0$, consequently no subtraction to correct the result is needed prior to rounding.

For round to positive (negative) infinity for positive (negative) results, ones are added to the 54th through 64th bits so that a one in any of these bit locations will be propagated to the 53rd bit. When the ZS-1 result differs from an infinite precision rounded one, the ZS-1 is high (low) by one LSB.

For round to negative (positive) infinity for positive (negative) results, the 54th through 64th bits are merely truncated so that ones in these bit locations will not affect the 53rd bit. When the ZS-1 result differs from the infinite precision rounded one, the ZS-1 is low (high) by one LSB.

Round to zero can be viewed as round to negative (positive) infinity for positive (negative) results. We do not treat it as a separate case.

Summary

The following tables summarize simulation data for positive results. The method used here is compared to the result obtained if the mantissa were computed to infinite precision prior to rounding. The following tables summarize the differences found for the different rounding modes comparing the ZS-1 result to the infinite precision rounded result.

(1) Round to nearest:

Correction	Same as ∞ round	High	Low	% Differing
0	1997251	35	2714	0.138
2^{-63}	1998197	534	1269	0.090
2^{-62}	1997583	1839	578	0.121

This confirms that the best correction for rounding to nearest is to add 2^{-63} to the final result before rounding.

(2) Round to positive infinity (correct by adding 14 LSBs).

Correct	High	Low	% High
1977792	22208	0	1.11

Note that all results not agreeing with the infinite precision rounded result are high by one LSB. This correctly produces an upper bound on the exact result.

(3) Round to negative infinity (correct by subtracting 2 LSBs).

Correct	High	Low	% Low
1993883	0	6117	0.306

Note that all results not agreeing with the infinite precision rounded one are low by one LSB. This correctly produces an lower bound on the exact result.

Conclusions

We have presented the reciprocal formation method used in the ZS-1 prototype systems. A main feature of our implementation is enhanced accuracy over commonly-used methods. There are two ways in which accuracy was improved.

(1) A ROM interpolation technique was used for computing table values to minimize the absolute error of the first approximation. As was shown in the Error Analysis section, the absolute value of the error in X_0 was reduced from 2.5×2^{-16} to 1.5×2^{-16}

(2) A "compressed" intermediate value was carried from the *Iterate 1* to the *Iterate 2* instruction.

To demonstrate the overall accuracy improvements from each of the above techniques, we re-ran the error simulation experiments. In the first case, we did not use ROM interpolation for computing table values, but used truncated values, as is commonly done. The remainder of the implementation was exactly the same as in the ZS-1, except the "correction" values used were adjusted to minimize the number of errors. In the second case, we used an implementation that uses an *Iterate 1* result that is simply rounded and stored as a normal floating point number, rather than in our "compressed" form. This models the algorithm used in the Cray Research processors, but with the IEEE floating point format and rounding.

The following table contains results for both sets of simulations: the first uses ROM values based on truncated inputs and the second uses a CRAY-like iteration method. Otherwise, the algorithms used in each case are unchanged from the ZS-1 method. In all the simulations, the same set of 2 million randomly selected operands were used. For comparison, results for the complete ZS-1 algorithm are repeated.

Method	Same as ∞ round	High	Low	% Differing
ZS-1	1998197	534	1269	0.090
Input trunc.	1993642	2185	4173	0.312
Cray-like	1596238	201663	202099	20.188

We see that using ROM values based on truncated inputs causes over three times as many results to differ from the true infinite precision rounded reciprocal. The Cray-like iteration method results in over 200 times as many results differing from the infinite precision rounded reciprocal.

The following table contains the worst-case absolute value of the additional error introduced by not computing the reciprocal to infinite precision and performing a true round to nearest.

Method	Additional Error
ZS-1	0.015×2^{-53}
Input trunc.	0.031×2^{-53}
Cray-like	1.000×2^{-53}

We see that truncating the input values used for computing the ROM values leads to twice the added error than with the ZS-1 method, and using a Cray-like iterate leads to additional error that is almost two orders of magnitude larger than with the ZS-1 method.

Finally, all the discussion and analysis in this paper has concentrated on the formation of a reciprocal. Of course, a complete division must also multiply the dividend by the reciprocal. When this is done, additional rounding errors may be introduced by the multiplication. All division by reciprocal algorithms require this "extra" floating point operation with its attendant "extra" rounding error, and the method described here is no exception.

References

[1] S. Waser and M. J. Flynn, *Introduction to Arithmetic for Digital Systems Designers.* New York: CBS College Publishing, 1982.

[2] Cray Research, Inc., *CRAY-1 S Series, Hardware Reference Manual.* Chippewa Falls, WI: Cray Research, Inc., Publication HR-808, 1980.

[3] S. F. Anderson, J. G. Earle, R. E. Goldschmidt, and D. M. Powers, "The IBM System/360 Model 91: Floating-Point Execution Unit," *IBM Journal*, pp. 34-53, January 1967.

[4] A. E. Charlesworth, "An Approach to Scientific Array Processing: The Architectural Design of the AP-120B/FPS-164 Family," *Computer*, vol. 14, September 1981.

[5] Intel Corporation, "i860 64-Bit Microprocessor," Order No. 240296-001, February 1989.

[6] J. E. Smith, et al, "The ZS-1 Central Processor," *Proc. ASPLOS II*, pp. 199-204, October 1987.

[7] D. Stevenson, "A Proposed Standard for Binary Floating-Point Arithmetic," *Computer*, vol. 14, pp. 51-62, March 1981.