Cascaded Implementation of an Iterative Inverse–Square–Root Algorithm, with Overflow Lookahead

Hercule Kwan
Trimble Navigation
2105 Donley Drive
Austin, Texas 78758

Robert Leonard Nelson, Jr.
Trimble Navigation
2105 Donley Drive
Austin, Texas 78758

Earl E. Swartzlander, Jr.
Department of Electrical and
Computer Engineering
University of Texas at Austin
Austin, Texas 78712

Abstract

We present an unconventional method of computing the inverse of the square root. It implements the equivalent of two iterations of a well-known multiplicative method to obtain 24-bit mantissa accuracy. We implement each "iteration" as a separate logic module and exploit knowledge about the relative error during computation to reduce the size of the implementation. We use overflow lookahead logic to facilitate the exponent computations. No division is required in the entire process. Examples and error analysis are given.

1 INTRODUCTION

The square–root operation plays an important role in computer arithmetic. The traditional Newton–Raphson method is extremely powerful, but has the drawback that it involves division, which does not lend itself to easy implementation [5]. It can be seen in [1], [2], [3], and [4]. Even one of the modern floating–point digital-signal-processor (DSP) chips uses iterative techniques to compute square root, provided that an initial approximation or "seed" is generated first [18]. Several papers ([6], [7], [8]) propose good initial values for the Newton–Raphson method. Most of them use polynomial approximations to model the square–root function. This approach further complicates the problem, because quite a few coefficients have to be calculated in order to get a good starting value. Several binary square–rooting algorithms have been described in [10] and [11]. Higher radix division and square–rooting algorithms are presented in [12], [13], [14], [15], [16] and [17]. These algorithms all use higher–radix redundant number systems, whose goals are to reduce the number of iterations. The higher the radix, the fewer iterations, since more bits are computed at each iteration. However, these methods also have two disadvantages: As Hwang notes, hardware complexity increases with the computation of multiples of high–radix divisors; and digit selection schemes are very complicated [19]. Furthermore, the Sweeney-Robertson-Tocher (SRT) method often used in higher–radix methods may not be that attractive in high–speed DSPs. A nonlinear digital–filter approach is presented in [20]. This method is quite interesting; however, it uses a commercially available DSP processor and implements the algorithms in software instead of hardware. Moreover, the nonlinear filter adds an unnecessary layer of complexity to the problem, which we naturally want to avoid.

In [21], Wallace proposed a method of computing the inverse of square root. He claimed that his method is faster than the traditional Newton–Raphson method; but, it involves two individual iterative equations and one needs to carry out multiplications in both equations at the same time. This becomes slower and less efficient if we want to shorten delay time and reduce hardware complexity.

This paper develops a method of forming the inverse of square root (i.e., one divided by square root). The inverse of the square root is often more useful than the square root itself, especially when used to normalize vectors. To normalize a vector, one typically takes the dot product of the vector with itself, computes the square root, takes the inverse, and then multiplies the vector components. The equation of the traditional Newton–Raphson method for the (non–inverse) square root is

$$x_{i+1} = \frac{1}{2}(x_i + \frac{a}{x_i}) \qquad (1)$$

Its relative error, ϵ_i, is defined as

$$\epsilon_i = \frac{x_i - \sqrt{a}}{\sqrt{a}} \qquad (2)$$

1063-6889/95 $4.00 © 1995 IEEE

and

$$\epsilon_{i+1} = \frac{\epsilon_i^2}{2(\epsilon_i + 1)} \tag{3}$$

where a is the square, x_i is the recursive solution approaching the square root of a, and ϵ_i is the error of x_i relative to the square root. This error equation implies that after each iteration, the number of significant mantissa bits is equal to about 1 more than twice the previous number. This means that, with a starting value of about 6-bit mantissa accuracy, we achieve 27-bit accuracy after two iterations. (Even a not-so-good starting value converges if this algorithm is allowed many iterations.)

The equation of the less-well-known multiplicative method [9] is

$$x_{i+1} = x_i\left(\frac{3}{2} - \frac{b}{2}x_i^2\right) \tag{4}$$

The relative error, ϵ_i, is defined as

$$\epsilon_i = \frac{x_i - \frac{1}{\sqrt{b}}}{\frac{1}{\sqrt{b}}} \tag{5}$$

and

$$\epsilon_{i+1} = -\frac{1}{2}\epsilon_i^2(\epsilon_i + 3) \tag{6}$$

where b is the inverse square, x_i is the recursive solution approaching the inverse square root of b, and ϵ_i is the error of x_i relative to the inverse square root. While slightly inferior to the Newton–Raphson method, this multiplicative method also almost doubles the number of significant mantissa bits at each iteration, so that only two iterations are needed for single precision with a practical starting value.

In our approach we use a cascaded implementation of an iterative algorithm. In an iterative algorithm, the output from a current iteration is fed back to become the input to the next iteration. We implement the equivalent of two iterations of the multiplicative method; but, instead of feeding data back iteratively through a single arithmetic unit, we use different logic for the two cascades of computation. Each stage has its own logic; and data are computed in a cascaded fashion.

Our goal is to achieve rapid convergence. In the first stage, we exploit the limited accuracy to remove the bottom bits of the calculation; we exploit the small error in the second stage to remove the middle of the calculation. We use a lookup table for initial approximations, because not too many bits are needed. The sign of the error alternates at each iteration, as can be seen in equation (6). We exploit this by controlling the sign of the error of the initial approximation so that we know the sign of the error throughout the computation.

In the first section, we review some basic concepts behind the classical square–rooting algorithms. The second section describes how to find good initial values for the multiplicative method. In the third section, we discuss the actual hardware implementation of our method. We then give two examples. Finally we present a formal error analysis of our method.

2 INVERSE OF SQUARE ROOT

It is well known [9] that the multiplicative method, as shown in equation (7), converges to the inverse square root of b, provided that the initial value x_0 is close enough to the actual value.

$$x_{i+1} = x_i\left(\frac{3}{2} - \frac{b}{2}x_i^2\right) \tag{7}$$

where

$$\lim_{i\to\infty} x_i = \frac{1}{\sqrt{b}} \tag{8}$$

On the other hand, if the initial value is too far from the actual value, this equation will diverge. In that case, the iterative solution will have the opposite sign each time, as can be seen easily from the equation. This is unlike the Newton–Raphson method, which promises convergence with any initial value. However, with a good initial value, the iterative answer given by this equation converges very fast and yields almost twice the previous number of significant mantissa bits at each iteration. Let ϵ_i be the relative error between x_i and the actual value $1/\sqrt{b}$ as defined in equation (5). It can be shown that

$$\epsilon_{i+1} = -\frac{1}{2}\epsilon_i^2(\epsilon_i + 3) \tag{9}$$

Equation (9) shows that the relative error is always negative after the first iteration. That is to say, the factor $(\frac{3}{2} - \frac{1}{2}bx_i^2)$ adjusts x_i at each iteration to be less than the actual value as it approaches the inverse square root.

Our objective is to achieve 24–bit mantissa accuracy in two "iterations" of the multiplicative method. Instead of looping through this method twice, we implement the equivalent by cascading two stages together. However, the implementation of each of these stages is different. In the first stage, we need only slightly more than 12 bits of accuracy. It is therefore unnecessary to use a full–sized 24x24–bit array multiplier to compute the adjusting factor $(\frac{3}{2} - \frac{1}{2}bx_i^2)$. Thus we deliberately remove the bottom bits of the calculations. In so doing, we reduce the size of our implementation, shorten the computation delay, and still maintain the required accuracy. The values of x_i^2 used in the first stage are stored in read-only memory, alongside the values of x_i whose squares they represent. This not-too-great increase in the size of the ROM eliminates the logic and time needed to multiply x_i by itself to get the square. We use a denormalizing shifter to shift b before multiplying it by x_i^2. Since the second stage needs full accuracy, we use multipliers with bigger sizes to fulfill the requirements. The adjusting factor computed at the second stage has the form $(1 + \delta)$, where δ is a very small number. The subtract operation that appears in the equation $(\frac{3}{2} - \frac{1}{2}bx_i^2)$ is somewhat misleading; it can be approximated by shifting bx_i^2 by one bit to the right, inverting the bits, and appending it to $1.\overbrace{00\ldots00}^{n\ zeros}$ to form $1.\overbrace{00\ldots00}^{n\ zeros}b_nb_{n+1}b_{n+2}b_{n+3}\ldots$, where $b_n, b_{n+1}, \ldots$ are the bottom bits of $-\frac{1}{2}bx_i^2$. This is due to the fact that $\frac{1}{2}bx_i^2$ is very close to 0.5 and the adjusting factor is a value close to 1 with a lot of zeros in the middle. The final mantissa is obtained by multiplying x_i by the adjusting factor and has the form $x_i + \delta x_i$. Note that the final form of the adjusting factor implies that we can remove the middle of the calculation (all the middle bits are zeros). There are two methods to calculate the initial values for the first stage. As we shall see, the knowledge of the sign of the error, σ_i, is very useful.

σ_i	+	−
Adjusting factor	$1 - \delta$	$1 + \delta$
σ_{i+1}	−	−
Adjusting factor	$1 + \delta$	$1 + \delta$
σ_{i+2}	−	−

Table 1. Two approaches to compute inital values.

In our implementation, we impose a condition that the sign of σ_i of bx_i^2 must be predictable. If the sign of b and x_i^2 each is predictable, then σ_i is predictable. If we use an initial value slightly bigger than the inverse square root, then σ_0 is positive. Then the first adjusting factor, $(\frac{3}{2} - \frac{1}{2}bx_i^2)$, must be slightly smaller than 1 in order for the recursive solution to converge to the actual value. On the other hand, if we use an initial value slightly smaller than the inverse square root, then σ_0 is negative. The adjusting factor, therefore, must be made greater than 1 to bring the solution close to the actual value. Table 1 summarizes σ_i of the two approaches. Now let us discuss which is better for implementation.

We need to reduce the accuracy of the number b and the initial estimates of x_i and x_i^2 in such a way that the result is, in one case, no smaller than the full-length number or, in the other case, no larger.

In the latter case (σ_0 negative), we can satisfy this requirement simply by truncating the full-length values. The first case (σ_0 positive) is more of a problem. One exact way is to first negate the number, truncate it, and then negate it again. This is hard and time–consuming. A better way is to add 1s in all bits to be truncated, and then truncate the sum. However, delay time is not improved either. To shorten delay time, we can approximate the value by truncating the number first and "jamming" the new LSB to 1. This method is easier and produces a value very close to the exact method.

Let us define two functions. The first one is *pseudofloor*(). It gives the largest fixed point number after truncation that is smaller than the argument. The second function is *pseudoceiling*() and it gives the smallest fixed point value that is bigger than the argument. These two functions are similar to the integer functions *floor*() and *ceiling*(). The difference is that our functions do not return integer results; they operate on the mantissa bits after the binary point and produce results with the precision required by our computations.

Assume a number b is in the range of $[b_i, b_{i+1})$. To start with negative error, one needs to make $\frac{1}{2}bx_i^2 < \frac{1}{2}$, or equivalently, $bx_i^2 < 1$ so that the adjusting factor is a little bit bigger than 1. We need to find the pseudofloor of b and x_i^2 in order to make sure that bx_i^2 is less than 1. It is very easy to obtain the pseudofloor of b; just truncate the bottom bits of b to zeros. The value x_i is obtained from the biggest number in the interval, i.e., bottom bits of the number are all ones. Square it and truncate the bottom bits to zeros. This gives the pseudofloor of x_i^2. Multiplying b and x_i^2 yields a number slightly less than 1. Shifting the product one bit to the right divides it by 2. Subtract the number from $\frac{3}{2}$. That produces an adjusting factor slightly bigger than 1.

On the other hand, if one uses positive σ_i, the pseudoceilings of b and x_i^2 need to be calculated so that bx_i^2 will be greater than or equal to 1. The pseudoceiling of a number v is obtained by adding 1 to the least significant bit, truncating the sum to the desired number of bits and keeping the truncated number slightly bigger than v.

We should note that extra addition associated with truncation is needed in the first stage of our computations if we start with positive σ_i. Delay in computations will therefore increase substantially and it counteracts our original goal. Therefore, we choose to implement the negative–error approach.

3 IMPLEMENTATIONS

Before we describe our implementation, let us define some notation. A floating point number v can be represented as $s_v\, e_{v,7}\, e_{v,6} \dots e_{v,0}\, m_{v,22}\, m_{v,21} \dots m_{v,1}\, m_{v,0}$, where $e_{v,i}$ denotes the i-th exponent bit of number v and $m_{v,j}$ the j-th mantissa bit of number v. Let u be a value between 1/4 and 1 and whose exponent LSB and 7 mantissa MSBs match the 8–bit address (i.e. $e_{u,0}m_{u,22} \dots m_{u,16} = e_{v,0}m_{v,22} \dots m_{v,16}$). Let r and q be the initial value and its square obtained from the ROM look–up table respectively, where $r = 1/\sqrt{u}$ and $q = 1/u$. The exponent is calculated independently and in parallel with the mantissa. We use the exponent LSB to perform a trivial 1–bit denormalization of mantissa. The mantissa of the result depends only on the exponent LSB and the mantissa of the input. The exponent of the result, with one exception, depends only on the exponent of the input. Thus this problem no longer involves floating point calculations but fixed point. The resulting exponent combines with the mantissa answer to form again a floating point answer. The exception occurs when the denormalized input mantissa is equal to 1/4, which leads to an unnormalized output mantissa of 2. But we put special logic to anticipate this exception. When this "overflow lookahead" detects the occurrence, it adds a 1 to the exponent of the result. We need not correct the mantissa, since the only error is in the MSB, which is never actually generated.

A small ROM table is easily implementable in today's VLSI technologies. It also occupies a very small area in the circuit layout. Moreover, hardware execution of a table lookup is extremely fast, compared to software–instruction execution. We store the initial approximations and their squares in a ROM look–up table. We use the least significant bit (LSB) of the exponent and the upper 7 bits (MSBs) of the mantissa of the number to form the address of the ROM table.

The initial approximations are computed according to the rules we mention in Section 2. It is necessary to store only the required number of bits of these numbers in the ROM table. By experimentation with a simulation, we have decided to store 8 bits of x_i and 12 bits of x_i^2. (We actually use 9 bits of x_i if we count the implied 1 in the beginning of the number.) To get these initial values of x_i, truncate them to 8–bit accuracy after the binary point. Square the truncated values and truncate them to 12–bit values to form x_i^2. The reasons for this choice are as follows: To start the computation, we need at least 6 bits of mantissa accuracy. We want to estimate the error, which should be less than 2^{-12}, in the first adjusting factor $(\frac{3}{2} - \frac{1}{2}bx_i^2)$. That is why we need about 12–bit accuracy in x_i^2. These errors are due to argument truncation and function truncation. Argument truncation is caused by our truncating the input number as the address to our ROM table. This address is not an exact representation of the input argument since it can be applied to the range of numbers with the same address. Function truncation comes from the ROM table where the number retrieved has only 8–bit accuracy. Therefore, the adjusting factor needs to compensate these errors and steer the initial estimate so that it converges to the actual value.

In our implementation, the exponent computation depends on whether the final mantissa overflows (i.e., mantissa ≥ 2.0). If it overflows, we need to add 1 to the exponent of the result and (implicitly) renormalize the mantissa to 1 (all mantissa bits cleared). Since we do not know the result until the last stage, the exponent computation would be slowed down. However, overflow occurs only when the input mantissa is 1/4, with address $1000\ 0000_2$ and mantissa bits all zeros. Therefore, we detect this situation early in the beginning, so that this information is available to the exponent circuit ahead of time.

To realize this function, we literally need a 24–input gate. Of course, explicit implementation of a 24–input gate is not practical in most technologies. However, the extremely loose requirement on propagation time due to the parallel computation of the mantissa allows the use of a tree of cascaded smaller gates to emulate the 24–input gate. With these cascaded smaller gates, we achieve overflow lookahead and remove the dependency on the mantissa computations.

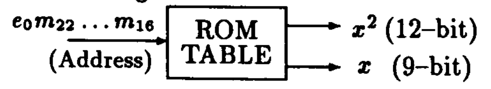
3.1 Algorithm

The mantissa is calculated in two separate stages. In the first stage, we use the knowledge of the error and force the adjusting factor to be slightly bigger than 1. First form the address pointing into the ROM table by combining the exponent LSB and the upper 7 MSBs of the mantissa of v (i.e. $e_{v_0}m_{v_{22}} \dots m_{v_{16}}$). This 8–bit address is used to obtain the initial value and its square from the ROM table.

We use a group of 2:1 multiplexers to form a denormalizing shifter. The input value b is shifted one bit to the right if the exponent LSB is equal to 1.

The value bx_i^2 is close to 1. By subtracting one half of bx_i^2 from 1.5 yields a result extremely close to 1. If the product bx_i^2 is slightly bigger than 1, it has the representation $1.\overbrace{00\ldots00}^{n\ zeros}b_nb_{n+1}b_{n+2}b_{n+3}\ldots$. Division by 2 yields $0.\overbrace{100\ldots00}^{n\ zeros}b_nb_{n+1}b_{n+2}b_{n+3}\ldots$ and $(\frac{3}{2}-\frac{1}{2}bx^2)_i$ $= 1.\overbrace{00\ldots000}^{n+1\ zeros}\bar{b}_n\bar{b}_{n+1}\bar{b}_{n+2}\bar{b}_{n+3}\ldots$. The lower bits are shifted one bit to the right, inverted, and placed in a register to form the adjusting factor.

In the first stage, $\frac{1}{2}bx_i^2 = 0.0a_na_{n-1}a_{n-2}\ldots$ (binary). The adjusting factor, $(\frac{3}{2}-\frac{1}{2}bx_i^2)$, is equal to $1.0\bar{a}_n\bar{a}_{n-1}\bar{a}_{n-2}\ldots$. We use a 15-bit by 12-bit multiplier to compute bx_i^2. Then we use a 16-bit by 9-bit multiplier to compute the product of x_i and the adjusting factor. The logic diagram of the first stage is shown in Figure 1.



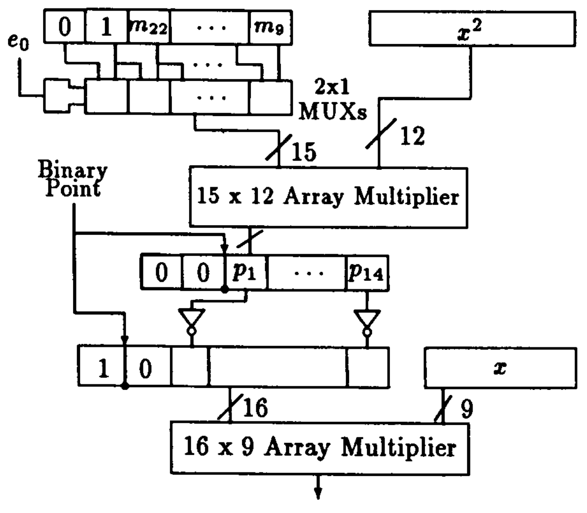


Figure 1. Logic diagram of the first stage.

In the second stage, we first compute x_i^2 and then multiply the entire (not truncated) b to it. A 24-bit precision, not 24x24, array multiplier is needed. In this stage, bx_i^2 is slightly smaller than 1. From our previous description, we know that the product bx_i^2 has the form $(1+\delta)$. Multiplying the adjusting factor to x_i results in $x_i+\delta x_i$. This suggests that we can add a very small number, δx_i, to x_i. We remove the middle of the calculation because all the bits in the middle are zeros. If the final mantissa exceeds 2, then it is renormalized to 1 again (all mantissa bits set to 0) and the exponent of result will increase by 1. The result is the final mantissa. Figure 2 shows the logic of the second stage.

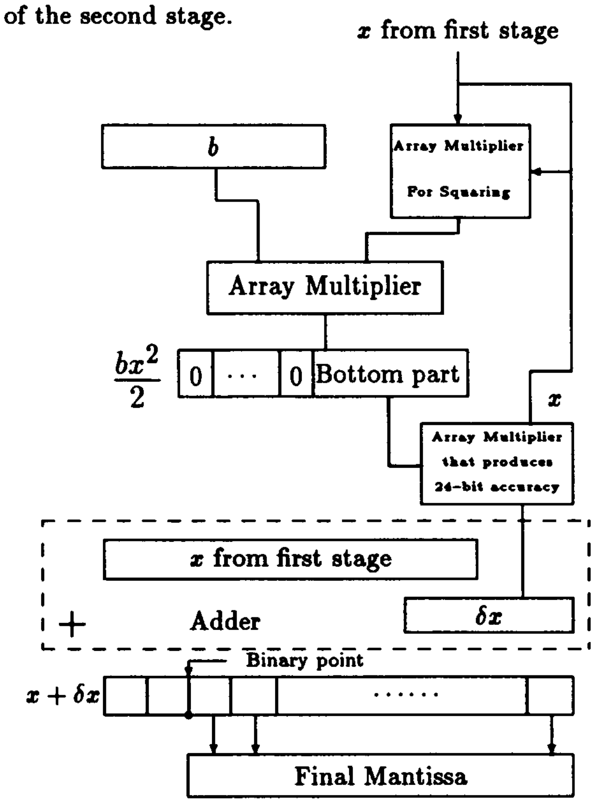


Figure 2. Logic digram for the second stage.

The exponent is computed as follows. We start with an input number v of the form 2^{e_v-127} [1, 2). We want to find its inverse square root 2^{e-127} [1, 2). First obtain the exponent value of v. Examine the LSB of e_v. If LSB is zero, the exponent is even; otherwise, it is odd.

When exponent is even,

$$\begin{aligned} e'_{even}-127 &= \frac{1}{2}(-e_v+126) \\ e'_{even} &= \frac{1}{2}(-e_v+126+254) \\ e'_{even} &= \frac{1}{2}\overline{e_{v,7-1}}+63 \end{aligned} \tag{10}$$

When exponent is odd,

$$\begin{aligned} e'_{odd}-127 &= \frac{1}{2}(-e_v+125) \\ e'_{odd} &= \frac{1}{2}(-e_v+125+254) \\ e'_{odd} &= \frac{1}{2}\overline{e_{v,7-1}}+62 \end{aligned} \tag{11}$$

where e'_{even} and e'_{odd} are the exponents of result when input exponent is even and odd respectively.

If an overflow occurs in the final mantissa, then add 1 to the sum. The result is the exponent of the inverse square root of v. The logic implementation of the exponent circuit is shown in Figure 3. Note that the logic in the middle adds a 1 to each of the input bits. Also we integrate the overflow bit with the exponent circuit since the exception only occurs when the input exponent is odd.

We now summarize our method in Sections 3.1.1 and 3.1.2.

3.1.1 Mantissa

The initial value for the mantissa of the inverse of square root of v can be found by looking up in the ROM table.

(1) First form the 8-bit address by combining the exponent LSB and the upper 7 MSBs of the mantissa of v (i.e. $e_{v,0}m_{v,22}\ldots m_{v,16}$).

(2) Then obtain the value pointed to by this address from the ROM table. (Note: The exponent of this value will be passed to another separate path to compute the final exponent, as described in Section 3.1.2).

(3) The value obtained is the initial value x_0 for equation (7).

(4) Set b to $0.1m_{v,22}\ldots m_{v,9}$.

(5) If the exponent LSB of v is 1, then shift b one bit to the right. Otherwise, if the exponent LSB of v is 0, then b stays unchanged.

(6) After x_0 and b are estimated, apply them to equation (7) to obtain x_1, the mantissa of the inverse of square root of v computed from the first stage. (*Note: The computations in this step are greatly reduced as we describe in Section 3.1.*)

(7) Apply x_1 and b to equation (7) to obtain the mantissa of the inverse of square root of v.

(8) If the mantissa is equal to or greater than 2, set all the mantissa bits to zeros. Add 1 to the exponent computed in Section 3.1.2.

3.1.2 Exponent

The exponent can be computed easily by the following steps:

(1) Obtain exponent $e_{v,7}e_{v,6}\ldots e_{v,0}$ of the floating point number v.

(2) Invert exponent and obtain $\overline{e_{v,7}e_{v,6}\ldots e_{v,0}}$.

(3) Shift $\overline{e_{v,7}e_{v,6}\ldots e_{v,0}}$ by 1 bit to the right.

(4) If the exponent is even (i.e., $e_{v,0} = 0$), then add 00111111_2 or 63 (Hex) to the shifted number; otherwise add 00111110_2 or 62 (Hex).

(5) If there is an overflow in the mantissa of result, add 1 to the sum obtained in step (4).

(6) The final sum is the exponent for the inverse square root of v.

Lastly, combine the mantissa and exponent bits computed above. The result is the inverse of square root of the number v.

3.2 EXAMPLES

3.2.1 $v = 0.03568$

Mantissa: (1) Exponent $e_{v,0}m_{v,22}\ldots m_{v,16} = 00010010_2$. (2) The smallest x in this address range (from the ROM table) = 1.31640625. (3) From the ROM table, $x^2 = 1.732421875$ ($01.10\ 1110\ 1110_2$). (4) $b = 0.100\ 1001\ 0001\ 0010\ 1_2$. (15-bit after exponent) (5) $bx^2\ /\ 2 = 0.011\ 1111\ 0100\ 1011_2$ and the adjusting factor $(\frac{3}{2} - \frac{1}{2}bx^2) = 1.000\ 0000\ 1011\ 0100_2$. (6) Result obtained from the first stage = 1.323637484 (13-bit accuracy). (7) Apply the above result to the second stage. The result is 1.32351108 (26-bit accuracy).

Exponent: (1) Exponent $e_{v,7}e_{v,6}\ldots e_{v,0} = 0111\ 1010_2$. (2) $\overline{e_{v,7}e_{v,6}\ldots e_{v,0}} = 10000101_2$. (3) Since $e_{v,0} = 0$, the exponent is even. (4) After shifting 1 bit to the right, the number is 01000010_2. (5) $01000010_2 + 00111111_2 = 10000001_2$.

3.2.2 $v = 24.678$

Mantissa: (1) Exponent $e_{v,0}m_{v,22}\ldots m_{v,16} = 1100\ 0101_2$. (2) The smallest x in this address range (from the ROM table) = 1.60546875. (3) From the ROM table, $x^2 = 2.577148437$. (4) Since the exponent LSB of v is 1, $b = 0.011\ 0001\ 0101\ 1011_2$. (5) $bx^2\ /\ 2 = 0.011\ 1111\ 1100\ 1100_2$ and the adjusting factor $(\frac{3}{2} - \frac{1}{2}bx^2) = 1.000\ 0000\ 0110\ 0110_2$. (6) Result obtained from the first stage = 1.61046624 (14-bit accuracy). (7) Apply the result to the second stage. The result is 1.61040463 (26-bit accuracy).

Exponent: (1) Exponent $e_{v,7}e_{v,6}\ldots e_{v,0} = 1000\ 0011_2$. (2) $\overline{e_{v,7}e_{v,6}\ldots e_{v,0}} = 01111100_2$. (3) Since $e_{v,0} = 1$, the exponent is odd. (4) After shifting 1 bit to the right, the number is 00111110_2. (5) $00111110_2 + 00111110_2 = 01111100_2$.

4 ERROR ANALYSIS

We estimate our proposed method's error by computing the relative error in the initial value of x that we use to seed the first stage, then propagating this error through two iterations of equation (5). This error as a function of x is the worst-case error in the

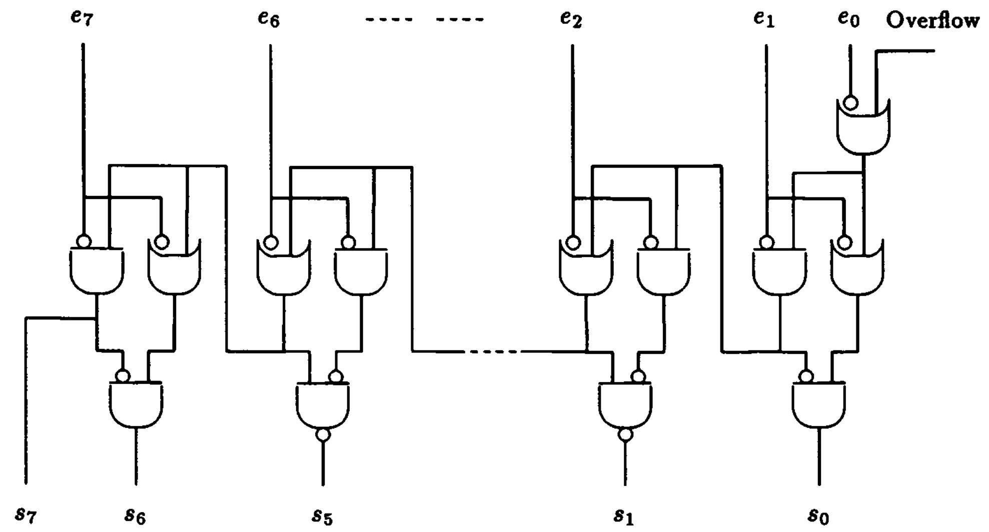


Figure 3. Logic implementation of the exponent circuit.

sense that we assume that it is the greatest possible error for any initial value of x in its neighborhood.

Although intermediate computational errors due to rounding, truncation, and the substitution of the ones-complement for the twos-complement negative in the first stage do contribute to the overall error, we rely on a margin of several guard bits at each level to assure that these are small compared to the final rounding to a 24-bit mantissa. We note specifically that small errors early in the computation share in the benefit from the relative-error reduction predicted by equation (5).

The initial value of x has two error components, one from truncating the address to reduce the table's length and the other from truncating the entries to reduce its width. The address-truncation error is

$$\begin{aligned} \epsilon_{address} &= \frac{\Delta x}{x} \\ &\cong \frac{dx}{db}\frac{\Delta b}{x} \\ &= -\frac{x^2 \Delta b}{3} && (12) \\ &= -\frac{x^2}{1536} \quad \text{for } .25 \le b < .5 && (13) \\ & \quad -\frac{x^2}{768} \quad \text{for } .5 \le b < 1 && (14) \end{aligned}$$

Because of the 8-bit address that we use, Δb is 1/128 of the interval [.25,.5), which is 1/512, or 1/128 of the interval [.5,1), which is 1/256.

Truncating the table entries introduces an error

$$\epsilon_{entry} = \frac{\Delta x}{x} = \frac{1}{256x} \qquad (15)$$

Δx is 1/256 because we store a 8-bit mantissa fraction.

The worst-case total relative error is then

$$\begin{aligned} \epsilon_{total} &= -\frac{1}{256x} - \frac{x^2}{1536} \quad \text{for } .25 \le b < .5 && (16) \\ & \quad -\frac{1}{256x} - \frac{x^2}{768} \quad \text{for } .5 \le b < 1 && (17) \end{aligned}$$

We pass this error through two iterations of equation (5) to get the relative error for the complete method. We then multiply this relative error by x to convert it to the error relative to the most-significant mantissa bit, which allows us to compare it directly to the rounding error for a 24-bit mantissa, which is 2^{-24}. The greatest such error occurs for $b = .5$, $x = \sqrt{2}$ and is $-2^{-27.9}$, which is almost four bits below the rounding error for the final result.

5 CONCLUSIONS

We present a method of computing the inverse square root based on a multiplicative algorithm. We implement the equivalent of two iterations of this algorithm. By exploiting the error equation, we greatly reduce the complexity of our implementation. The first stage provides a minimum number of significant mantissa bits to achieve 24–bit accuracy at the second stage. Our method increases the speed of computations and is very useful in DSP applications. It can also be easily extended to double–precision floating point numbers.

References

[1] F. B. Hildebrand, *Introduction to Numerical Analysis*, New York:Dover, 1987.

[2] A. Ralston and P. Rabinowitz, *A First Course in Numerical Analysis*, New York: McGraw–Hill, 1978.

[3] C. V. Ramamoorthy, J. R. Goodman and K. H. Kim, "Some Properties of Iterative Square–Rooting Methods Using High-Speed Multiplication," *IEEE Transactions on Computers*, vol. C-21, no. 2, pp. 837–847, October 1972.

[4] C. T. Fike, *Computer Evaluation of Mathematical Functions*, Englewood Cliffs, N.J.:Prentice–Hall, 1968.

[5] R. Hashemian, "Square Rooting Algorithms for Integer and Floating–Point Numbers," *IEEE Transactions on Computers*, vol. 39, no. 8, pp. 1025–1029, August 1990.

[6] M. W. Wilson, "Optimal Approximations for Generating Square Root for Slow or No Divide," *Communications of the ACM*, vol. 13, no. 9, pp. 559–560, September 1969.

[7] W. A. Beyer, "A Note on Starting the Newton-Raphson Method," *Communications of the ACM*, vol. 7, no. 7, p. 442, July 1964.

[8] R. F. King and D. L. Phillips, "The Logarithmic Error and Newton's Method for the Square Root," *Communications of the ACM*, vol. 12, no. 2, pp. 87–88, February 1969.

[9] W. James and P. Jarratt, "The Generation of Square Roots On a Computer with Rapid Multiplication compared with Division," *Mathematics of Computation*, 19, pp. 497–500, 1965.

[10] G. Metze, "Minimal Square Rooting," *IEEE Transactions on Electronic Computers*, vol. EC–14, pp. 181–185, April 1965.

[11] S. Majerski, "Square–Rooting Algorithms for High–Speed Digital Circuits," *IEEE Transactions on Computers*, vol. C-34, no. 8, pp. 724–733, August 1985.

[12] V. G. Oklobdzija and M. D. Ercegovac, "An On-Line Square Root Algorithm," *IEEE Transactions on Computers*, vol. C-31, no. 1, pp. 70–75, January 1982.

[13] L. Ciminiera and P. Montuschi, "Higher Radix Square Rooting," *IEEE Transactions on Computers*, vol. 39, no. 10, pp. 1025–1029, October 1990.

[14] M. D. Ercegovac and T. Lang, "Radix–4 Square Root without Initial PLA," *Computer Science Department Technical Report*, UCLA, March 1989.

[15] P. Montuschi and L. Ciminiera, "Simple Radix 2 Division and Square Root with Skipping of Some Addition Steps," *10th IEEE Symposium on Computer Arithmetic*, pp. 202–209, Grenoble, France, June 1991.

[16] P. Montuschi and L. Ciminiera, "On the Efficient Implementation of Higher Radix Square Root Algorithms," *9th IEEE Symposium on Computer Arithmetic*, pp. 154–161, Santa Monica, CA, September 1989.

[17] J. Fandrianto, "Algorithm for High Speed Shared Radix 8 Division and Radix 8 Square Root," *9th IEEE Symposium on Computer Arithmetic*, pp. 68–75, Santa Monica, CA, September 1989.

[18] *DSP96002 IEEE Floating-Point Dual-Port Processor User's Manual*, Motorola Inc., 1989.

[19] K. Hwang, *Computer Arithmetic: Principles, Architecture and Design*, New York:John Wiley & Sons, 1978.

[20] N. Mikami, M. Kobayashi and Y. Yokoyama, "A New DSP–Oriented Algorithm for Calculation of the Square Root Using a Nonlinear Digital Filter," *IEEE Transactions on Signal Processing*, vol. 40, no. 7, pp. 1663–1669, July 1992.

[21] C. S. Wallace, "A Suggestion for a Fast Multiplier," *IEEE Transactions on Electronic Computers*, vol. EC–13, pp. 14–17, February 1964.