

FAST, SIMPLE APPROXIMATION OF FUNCTIONS

Use A Lookup Table And a Digital Multiplier To Implement Newton-Raphson Approximations

by Matt Johnson

Now that low-cost, low-power digital multipliers and multiplier/accumulators are readily available (for example, ADI's ADSP-1000 device-family*) it has become possible to implement many digital signal-processing (DSP) applications with high speed. The most common applications are in performing fast Fourier transforms (FFT) or filtering incoming data from either a real-time source or from stored data off-line. However, their utility is not limited to such conventional applications. DSP components can be used, along with applied mathematical convergence theory, to embody algorithms for evaluating scientific functions at high speed.

When exact solutions are not required, approximation techniques—employing parallel computation using hard-wired components—allow us to bypass the long calculation times normally required by CPUs in evaluating scientific functions. Depending on the specific function and the accuracy of the desired solution, the execution time required by a CPU to effect solutions—using its built-in algorithms—can be bettered by as much as two orders of magnitude. For example, a 16-bit 8086 divide takes some 30 microseconds, compared to about 300 ns for the implementation to be described here, using an 8-bit lookup table and a hardware multiplier. Such increased performance can mean the difference between a successful design and a merely interesting one that never gets off the ground.

As an example of the technique, we show how one might implement an approximation of a common function occurring frequently in signal processing systems, the reciprocal ($y = 1/x$). A fast reciprocal is just one multiplication away from a full-blown fast divider, but redundant architectures do exist for implementing the complete division without the extra multiply cycle. We will examine both the theoretical implications and hardware realizations of the technique.

NEWTON-RAPHSON

We will apply the Newton-Raphson recursion algorithm, which is well known to users of mathematical approximation techniques. It is a very powerful algorithm because it produces quadratic convergence (i.e., each iteration provides a doubling of precision). To begin, a guess is made as to the root of the function (solution for

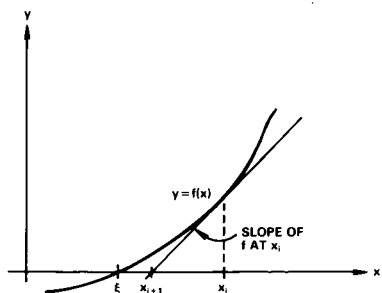


Figure 1. Well-behaved convergence using Newton-Raphson. The first guess is x_i . the second is x_{i+1} .

an unknown value) in the vicinity of the desired root. The derivative (slope) of the function is evaluated at the guessed value, leading in turn to new, successively closer iterations (Figure 1). The method may be applied to approximate any “well-behaved” function, i.e., a function not exhibiting zero derivatives within the region where the function is expected to converge.

The significance of “well-behaved” is apparent in the following cases. Figure 1 traces the algorithm as it converges through two iterations towards a root, ξ , of a monotonic function. As the plot shows, the guess for the i th iteration was x_i ; the slope is determined at x_i , and the point where it intersects the x -axis is the new guess, x_{i+1} . Notice how successive solutions converge quickly. Figure 2, on the other hand, illustrates two examples in which a naughty function (having zero derivatives) prevents convergence. Fortunately, the reciprocal is well-behaved for all values of $x \neq 0$.

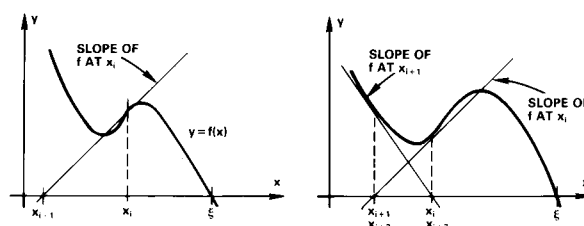


Figure 2. No convergence with zero-derivative functions.

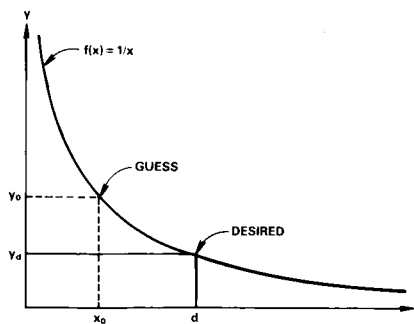


Figure 3. Functional relationship, $f(x) = 1/x$.

The Newton-Raphson recursion is derived using a first-order Taylor series expansion in the following manner (figure 3). Assume we wish to find the reciprocal of a value of x we will call d (thus, $y_d = 1/d$), knowing only the functional dependence, $f(x) = 1/x$, and the value of y at an arbitrary point, x_0 : $f(x_0) = y_0$. We define an error function, $F(y) = 1/y - d$, whose root occurs at $F(y_d)$, i.e.,

$$F(y_d) = 1/y_d - d (= 0) \quad (1)$$

When it is solved for y , we will have the desired value of the reciprocal at d , y_d .

We write a first-order Taylor series expansion of this error function about the point, y_d , giving us a linear approximation to the desired solution. Hence we have, for the n th iteration, y_n ,

$$F(y_d) \approx F(y_n) + (y_d - y_n)F'(y_n) \quad (2)$$

Equating this expansion to zero (to minimize the error) and generalizing y_d as y_{n+1} (the improved guess), equation 2 becomes:

$$0 = F(y_n) + (y_{n+1} - y_n)F'(y_n) \quad (3)$$

Solving (3) for the value of y_{n+1} (which we are seeking),

$$y_{n+1} = y_n - F(y_n)/F'(y_n) \quad (4)$$

This is the general form of the Newton-Raphson recursion. For the specific case of the reciprocal, we substitute the error function (from equation (1)) and its derivative ($F'(y_n) = -1/(y_n)^2$):

$$\begin{aligned} y_{n+1} &= y_n + y_n^2(1/y_n - d) \\ &= y_n(2 - d * y_n) \end{aligned} \quad (5)$$

PREPARATIONS

How do we make our first guess, y_0 , to start the Newton-Raphson iteration? The most effective way is to address a lookup table (ROM), having a relatively small number of fixed points, with a nearby value of x . Although the hardware could be simplified (eliminating ROM and associated circuitry) by always starting at a fixed arbitrary value, we would find that it is a somewhat inefficient approach, because the resulting higher initial error leads to more iterations than we want (albeit still faster than an exact solution executed by a CPU).

When lookup tables and finite-precision hardware are used, an important aspect of approximation techniques is data normalization. We would like to match the full operating range of a device to the full range of the function. In the interest of restricting the range of the function's argument to fully utilize the limited precision of hardware multipliers (or conversely, to maximize the dynamic range of the result), it is common practice to shift the argument (viz., successively multiply or divide by 2) until it is in the range within which a sufficiently correct answer can be obtained most efficiently. After the approximation technique has been applied, the result is denormalized (i.e., restored to its appropriate range).

Restricting discussion to positive values (a rather trivial stipulation), the range of the reciprocal is from 0 to +infinity. When using a lookup approach for generating the first guess: for a given table size, the smaller we limit the range of the argument, the closer the first guess will be. In the case of the reciprocal, we normalize the data to the range $1/2$ to 1^- (the superscript used in 1^- denotes all numbers less than but not including 1)*, i.e., data greater than or equal to 1 is right-shifted, and data less than $1/2$ is left-shifted, until the most significant bit appears at the 2^{-1} position. For example, 101.11010 would be shifted right until it became 0.1011101, and 0.0001111 would be shifted left until it became 0.1111000. The resulting truncated reciprocal function is the curve sector shown in figure 4.

To provide a uniformly accurate first guess for the Newton-Raphson recursion, we divide up the normalized dividend range, x , into a convenient number of intervals and assign the midpoint of each interval as our guess for $1/d$ for all values of d falling within that interval. Effectively, the first guess, $y_0(s)$, becomes a stepwise

*In fractional binary, this range would be 0.1000...0 to 0.1111...1.

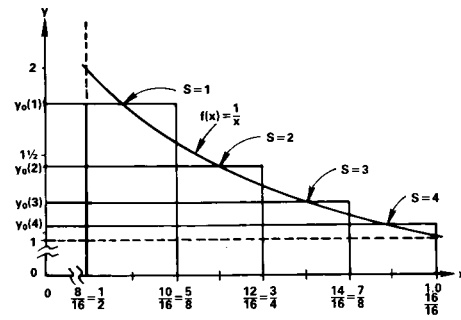


Figure 4. Normalized reciprocal $1/x$ and stepwise approximation $y_0(s)$ with $j = 2$.

approximation to $f(x) = 1/x$, being constant in each of 2^j equal segments between $1/2$ and 1^- . The true reciprocal at the midpoint of each segment, $s(x)$, may be calculated using (6).

$$y_0(s) = \frac{2^{j+2}}{2^{j+1} + 2s - 1} \quad (6)$$

Such a scheme is seen in Figure 4; the number of segments has been limited to 4 ($j = 2$) for illustrative purposes. For example, if $s = 3$, the value of $y_0(3) = 16/(8 + 6 - 1) = 16/13$. This is the exact reciprocal of $13/16$, the midpoint between $3/4$ and $7/8$.

It will prove useful to make a few observations about the normalized argument. Consider, for instance, the format of 16-bit binary numbers ranging from $1/2$ to 1^- :

$$d = 0.1 \text{ i i i i i i i i j j j j j j } \quad (7)$$

In (7), d is referred to as having a "1.15" format (one bit to the left of the decimal, and 15 to the right). This useful convention makes it easy to locate the proper decimal position of the product formed by two arbitrarily formatted numbers. We simply add the number of bits to the left and to the right of both operands, giving us the correct format of the resulting product. For example, $1.01 \times 1.001 = 01.01101$; we can predict the format of the product, 2.5, from the 1.2, 1.3 formats of the operands. If either operand is less than 1.0, the leftmost bit of the product will always be 0.

By virtue of the normalization, we know that the integer portion is 0 and the MSB of the fractional portion of the number will always be 1 (i.e., 2^{-1}); the rest of the number (2^{-2} and on...) remains arbitrary. The trick is to use the most significant part of the arbitrary portion of the number (say, the first eight bits: 2^{-2} through 2^{-9} , or i i i i i i i i) as an address into a lookup table whose contents are simply the reciprocal of the address (eq. 6).

Since $1/2$ to 1^- is the range of the normalized d , its actual reciprocal lies in the range, 2 to 1^+ , and we must reserve the 2^1 bit position for the result. But, because the approximation is computed to be the *midpoint* of each interval, the table contents will only have a range of 2^- to 1^+ ; i.e., the 2^1 bit is not needed in the table. Furthermore, since we know that the lookup value will always have the 2^0 bit = 1, we need only store the fractional portion of the guess in the table, manually setting the 2^0 bit at the input to the multiplier. Thus, we obtain a 9-bit initial guess with only an 8-bit lookup.

IMPLEMENTATION

Figure 5 is a block diagram of a setup to compute the result of equation (5). The procedure is to compute $d * y_n$, subtract it from 2, then perform the second multiplication (by y_n). For speed and

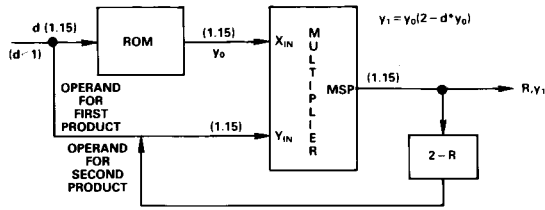


Figure 5. Multiplier layout for reciprocal calculation.

simplicity, the multiplications are 16-bit multiplications, in which only the 16-bit most-significant product (MSP) is used.

The first term formed is the $d * y_0$ term; its upper byte (most significant product—MSP—from the multiplier) is a 16-bit number in the 2.14 format. A single bit arithmetic left-shift (utilizing the format-adjust output option of the multiplier, FA = 0), dropping the leading 0 (we know that the first digit is 0, because the first digit of d is 0), is used to reformat the result back to the 1.15 format in preparation for the second multiply of the iteration, $y_0 (2 - d * y_0)$. Following (5), the shifted result is subtracted from 2 (in the appropriate format) and written back into the multiplier over the former Y register contents of d in preparation to take the product of y_0 and the bracketed expression. The second multiply completes the first recursion; the result may be read directly in the 2.14 format, or shifted (again using FA = 0) and read in the normalized 1.15 format if desired.

ERRORS

Besides the error of fitting the function with a limited number of recursions (which turns out to be negligible after only a single trial), there are also errors of computation. The two principal sources of computational errors are the limited ROM resolution (only 256 trial points) and the multiplication scheme.

The error analysis of this hardware reciprocal scheme is fairly straightforward. For the product, $y_0 * d$, we know that y_0 is in error by an amount less than 2^{-8} (since we assumed an 8-bit lookup table); call this error e_0 . That is, $y_0 = 1/d + e_0$. Due to feeding back only a single-precision intermediate result (i.e., the MSP of the multiplier, left-shifted by 1 bit) as an operand to the second multiply, we introduce a known computational error, e' , such that $y_1 = y_0 [(2 - d * y_0) + e']$.

We may now substitute the error terms into eq. 5 for the first recursion (y_1 ideally = $y_0 [2 - d * y_0]$):

$$y_1 = (1/d + e_0)[2 - d * (1/d + e_0) + e'] \quad (8)$$

$$= 1/d + e'/d - (e_0)^2 d + e_0 e' \quad (9)$$

We can combine the three right-hand terms as a single additive error term, e_1 ,

$$y_1 = 1/d + e_1 \quad (10)$$

Thus, the worst-case error associated with the first iteration may be expressed as:

$$|e_1| = |e'/d - (e_0)^2 d + e_0 e'| \quad (11)$$

$$\leq |e'/d| + (e_0)^2 |d| + |e_0| |e'| \quad (12)$$

This approach may be generalized to describe the error associated with any iteration as:

$$y_n = 1/d + e_n \quad (13)$$

$$y_{n+1} = 1/d + e_{n+1} \quad (14)$$

in which:

$$|e_{n+1}| \leq |e'/d| + (e_n)^2 |d| + |e_n| |e'| \quad (15)$$

Knowing that the maximum computational error, e' ($\leq 2^{-15}$), and the range of d (i.e., $1/2$ to 1^-), are both constants, we may substitute their worst case values into eq. 15 to determine the errors introduced by the implementation of the algorithm:

$$|e_{n+1}| \leq 2^{-14} + (e_n)^2 + |e_n| 2^{-15} \quad (16)$$

Since the second and third terms are second-order, with values which can be neglected, the maximum computational error contributed by each iteration is 2^{-14} .

Figure 6 is a plot showing the errors of 1000 equally spaced points over the normalized denominator range from $1/2$ to 1, for a *single iteration* of the Newton-Raphson recursion using an 8-bit lookup and single-precision feedback, as shown in Figure 5. The rms error is about 39 parts per million, slightly more than 2^{-15} , while the maximum error for the set of points tested is 150 ppm. Since $2^{-14} = 61$ ppm, and there are relatively few points with greater errors, we may conclude that the main source of error is the computational error introduced by the single-precision feedback term, e' .

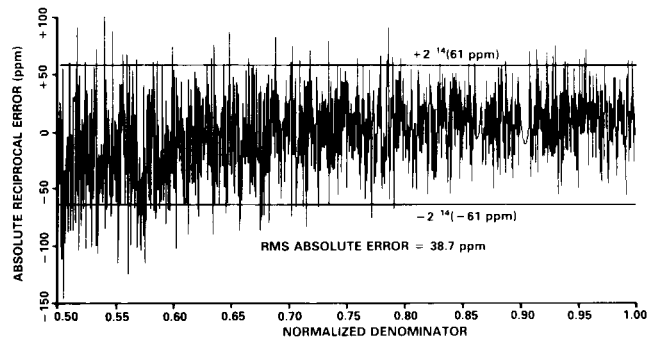


Figure 6. Absolute error for 16-bit reciprocal approximation.

We may also conclude that one recursion is generally adequate to obtain accuracies to better than 14 bits, and that the single-precision computational error, which degrades accuracy to the 14-bit level, insures that more than one iteration would not increase the accuracy. Although 14-bit accuracy is rather modest for many applications, remember that this whole computation is performed within 320 nanoseconds ($2 * 145$ ns for the multiplications + 30 ns for an ECL lookup table).

A SIMPLER IMPLEMENTATION

Until now, we have been assuming that all the multiplications have been performed in the unsigned arithmetic mode. A fortuitous identity in twos complement arithmetic allows us to simplify the hardware requirements of the reciprocal calculation:

A twos complement number is formed by taking the ones complement of a number and adding one LSB; it is equivalent to subtracting the number from the next higher power of 2. This suggests that perhaps the subtraction from 2 in (5) could be eliminated if the output of the multiplier were coded in 2s complement.

It may not be intuitively obvious, but we can eliminate the subtraction if we use the twos complement of the initial guess (call it y'_0) in forming the first product ($d * y'_0$)—and perform the multiplication in the “mixed” arithmetic mode (y'_0 in 2s complement and

d in unsigned), again using the format-adjust option of the multiplier (FA = 0). We get a single-signed, twos-complement product in the 1.15 format, which, if interpreted as an *unsigned* result, will in fact be the term $(2 - d * y_0)$. That is, when this result is fed back for the second multiplication, $y_0 (2 - d * y_0)$, we just clock it in as an unsigned number.*

It might appear that using this technique would require a double-size lookup table and an extra addressing bit for selecting between the twos complement and the unsigned values in the table of initial guesses. However, it turns out that we can avoid this.

Since there is strong evidence that the 8-bit ROM is more than adequate for the first guess, a 9-bit word to address a double sized, 8-bit lookup table needed to store both y_0 and y'_0 (which is not an available ROM configuration) is unnecessary. An initial guess, y_0 , with greater accuracy than the computational error, avails us nothing. Thus we can reduce the number of segments used to derive the first guess by half, relegating the 8th bit to a bank-select function. With this technique, the error due to the initial guess (the quadratic term) will still be smaller than the computational error. Figure 7 shows a suggested hardware diagram.

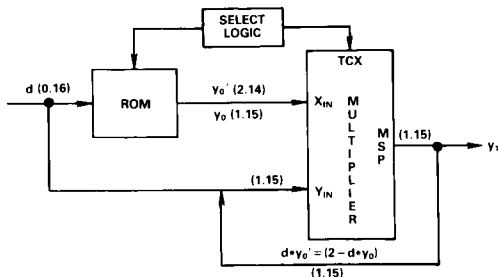


Figure 7. $(2 - d * y_0)$ in 1 operation with less hardware.

FURTHER SIMPLICITY WITH A SINGLE-PORT MAC

The ADSP-1110, a single-port, 16-bit multiplier/accumulator (MAC)—and much more—in a 28-pin package, is an unusually versatile new product from the DSP Division of Analog Devices. The ADSP-1110 has a 6-bit instruction set offering the user no fewer than 32 separate operations, while reducing the pin count for a 16-bit MAC function from 64 to only 28! Its salient features (Figure 8) include: a 40-bit accumulator, a pipelined input register, independent rounding control at bits 14 and 15, a left-shift output option, byte-swapping capability, an overflow flag, and saturation logic. The instruction set allows the user to form and accumulate products using any combination of addition, subtraction, and negation.

The Newton-Raphson reciprocal approximation can be implemented using the ADSP-1110 with simplified hardware. Although it was designed to excel in the multiply/accumulate (sum of products) mode, the device's internal bussing and left-shift option make it an attractive choice for recursive algorithms too.

By using the ADSP-1110 to implement the reciprocal, *either* approach $(2 - d * y_0$ or $d * y'_0)$ could be used in forming the first term. But again, the more expedient choice is the $(d * y'_0)$ which saves the implied overhead of preloading the accumulator with 2 in forming the difference term, $(2 - d * y_0)$.

In terms of layout, the 8th address bit will still be used to select either y_0 or y'_0 , which may be provided by an extra instruction bit

*The proof of this, beyond the scope of this article, is available from the author upon request.

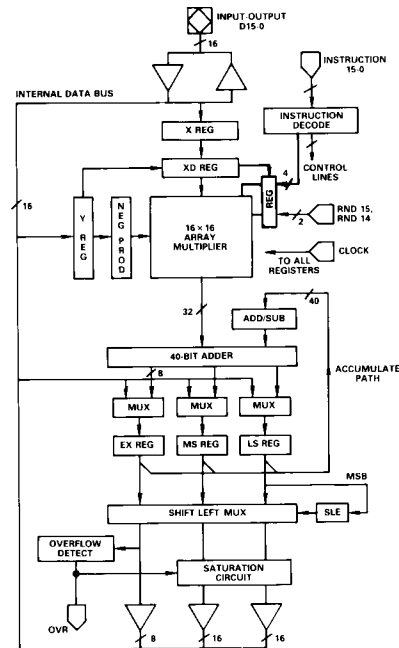


Figure 8. ADSP-1110 Single-Port MAC.

in the algorithm microprogram, but, except for that, the ADSP-1110 will do everything else inherently.

Consider the code sequence of figure 9. The first instruction loads the pipeline register with y'_0 . The second simultaneously shifts the pipeline register, loads d , and starts the product in the mixed mode. Halfway through this multiply, the pipeline register is loaded with the first operand of the second multiply, after which the first product is latched. The fifth instruction outputs the result, which is then loaded into the Y register, starting the 2nd multiply of the MS and the shifted pipeline register. There being no more arithmetic operations to perform, the last 3 instructions are required to finish the recursion and output the reciprocal.

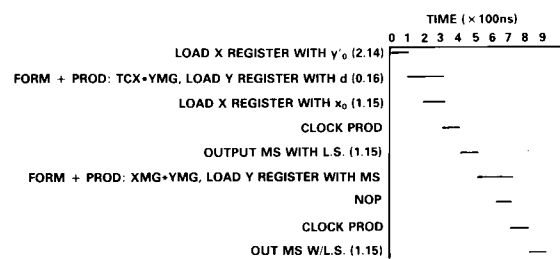


Figure 9. Code sequence to implement the algorithm with ADSP-1110.

Execution time (room temperature) for the discrete-multiplier implementation is $2 * 145\text{ns} + 30\text{ns} = 320\text{ns}$ (assuming 30ns for ECL ROM), or roughly $1/3 \mu\text{s}$. The single-port MAC offers a simpler design, but it is almost three times as slow (900ns).

In these pages, we have investigated approximation technique (the Newton-Raphson recursion); we have learned how lookup tables are used in conjunction with normalized numbers for greatest dynamic range; and we have seen the use of 2s complement to combine a product and a difference term in one multiply—and its use in hardware implementation of reciprocals with high-speed multiplying devices. These approximation techniques, applicable to other scientific functions as well, for example logarithms, exponentials, linearizing functions, etc., serve to illustrate some unconventional—but useful—applications of digital multipliers. ▀