Low Latency Digit–Recurrence Reciprocal and Square–Root Reciprocal Algorithm and Architecture[*]

Elisardo Antelo
Dept. Electronic and Computer Eng.
University of Santiago
Santiago de Compostela. SPAIN
elisardo@dec.usc.es

Tomás Lang
Dept. Electrical and Computer Eng.
University of California at Irvine.
Irvine, CA. USA
tlang@uci.edu

Paolo Montuschi
Dept. Automatica e Informatica.
Politecnico di Torino
Torino, ITALY
paolo.montuschi@polito.it

Alberto Nannarelli
Dept. Informatics and Math. Modelling
Technical University of Denmark
Kongens Lyngby, DENMARK
an@imm.dtu.dk

Abstract

The reciprocal and square–root reciprocal operations are important in several applications. For these operations, we present algorithms that combine a digit-by-digit module and one iteration of a quadratic–convergence approximation. The latter is implemented by a digit–recurrence, which uses the digits produced by the digit–by–digit part. In this way, both parts execute in an overlapped manner, so that the total number of cycles is about half of the number that would be required by the digit–by–digit part alone. Because of the approximation, correct rounding of the result cannot be obtained directly in all cases; we propose a variable–time implementation that produces the correctly rounded result with a small average overhead. Radix–4 implementations are described and have been synthesized. They achieve the same cycle time as the standard digit–by–digit implementation, resulting in a speed–up of about 2 and, because of the approximation part, the area factor is also about 2. We also show a combined implementation for both operations that has essentially the same complexity as that for square–root reciprocal alone.

1. Introduction

Reciprocal and square–root reciprocal are among the set of arithmetic operations implemented in hardware in microprocessors. For the implementation of reciprocal and square–root reciprocal there are the following alternatives (see [2] for details): i) Digit–by–digit algorithms, ii) Quadratic convergent algorithm, and iii) Polynomial approximations. Methods ii) and iii) require a dedicated parallel multiplier and additional tables. Method i) is a cost–effective alternative with a low hardware overhead.

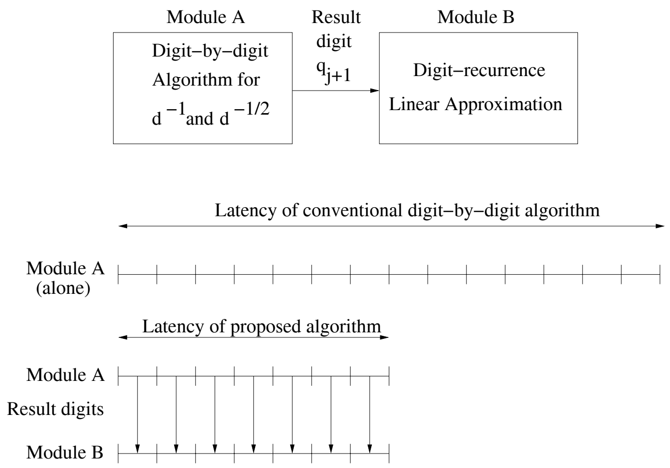
Digit–by–digit algorithms have been implemented in several general–purpose microprocessors and graphics processors. For instance, studies in [3] show that for a geometry processor, a digit–by–digit unit is a suitable choice. A recent highly scalable architecture based on stream processing also uses a digit–by–digit unit as part of the arithmetic clusters [4].

In this work we describe an algorithm for the computation of the functions $1/d$ and $1/\sqrt{d}$ which consists of a digit–by–digit part followed by a linear approximation. The scheme is illustrated in Figure 1. The total delay is reduced by implementing the linear approximation as a digit–recurrence which is performed in an overlapped fashion with the digit–by–digit part. Because the linear approximation has quadratic convergence, we perform roughly half of the iterations as compared to a conventional digit–recurrence algorithm. We require two datapaths operating in parallel.

2. Proposed algorithm

For the reciprocal computation, the digit–by–digit part of the algorithm produces an approximation k of $1/d$. Then, as described by the Newton–Raphson iter-

[*]E. Antelo has been partially supported by Xunta de Galicia under project PGIDT03TIC10502PR.



Module A
Result digit q_{j+1}
Module B
Digit–by–digit Algorithm for d^{-1} and $d^{-1/2}$
Digit–recurrence Linear Approximation
Latency of conventional digit–by–digit algorithm
Module A (alone)
Latency of proposed algorithm
Module A
Result digits
Module B

Figure 1. Proposed scheme.

ation (or equivalently, by a linear approximation based on the Taylor series expansion) a better approximation is given by

$$A = k(2 - kd) \tag{1}$$

The relative order of convergence of this iteration is quadratic, that is, if the relative error of k is δ, then the relative error of A is $\epsilon = \delta^2$.

Similarly to the reciprocal case, if the digit–recurrence part produces an approximation k of $1/\sqrt{d}$, a Newton-Raphson iteration produces a better approximation B, as follows:

$$B = \frac{k}{2}\left(3 - dk^2\right) \tag{2}$$

Again, in this case the relative order of convergence is quadratic. If δ and ϵ are the relative errors of k and B respectively, then

$$\delta = 1 - k\sqrt{d} \quad \text{and} \quad \epsilon = 1 - B\sqrt{d} \tag{3}$$

Therefore,

$$k\sqrt{d} = 1 - \delta \quad \text{and} \quad \epsilon = 1 - \frac{k\sqrt{d}}{2}\left(3 - dk^2\right) \tag{4}$$

Consequently,

$$\epsilon = 1 - \frac{1-\delta}{2}\left(3 - (1-\delta)^2\right) = ... = \frac{\delta^2}{2}(3 - \delta) \tag{5}$$

Then the relative error of B has a complexity of $O(\delta^2)$.

Floating–point representation is used with n–bit significands, and the input significand is shifted (with the corresponding exponent adjustment) to have the result in the interval $[1, 2)$. This is achieved by producing a shifted input significand d within the interval $[1/2, 1)$ for reciprocal and $[1/4, 1)$ for square–root reciprocal[1].

The proposed algorithm can be summarized as follows:

- Obtain the approximation k using a digit–recurrence algorithm and performing g iterations, roughly half of final required precision.

- Since k is obtained digit–by–digit in most–significant digit first mode, perform the computation of $A = k(2 - d\,k)$ and $B = (k/2)(3 - d\,k^2)$ by means of a digit recurrence. In this way, we have a full overlap between the computation of k and the computation of the approximation (see Figure 1).

The details of the digit–by–digit algorithms to compute k can be found for instance in [2]. In summary,

- Reciprocal: For the computation of $1/d$ a residual is defined as $w[j] = r^j(1 - d\;Q[j])$ where $Q[j]$ is the partial result up to iteration j, defined as

$$Q[j] = Q[0] + \sum_{i=1}^{j} q_i r^{-i}$$

  Then the algorithm proceeds as follows:

  1. Initializations: initialize $w[0]$, q_1 and $Q[0]$ to assure convergence. Several alternatives are possible [2].
  2. Recurrence[2]: for $j = 0$ to $g - 1$

$$\begin{aligned} w[j+1] &= rw[j] - q_{j+1}d \\ q_{j+2} &= SEL(rw[j+1], d) \end{aligned} \tag{6}$$

     The digits q_{j+1} take values in the set $\{-\rho(r-1), \ldots, 0, \ldots, \rho(r-1)\}$, where ρ is the redundancy factor ($\rho > 1/2$). The digit–selection function SEL assures an absolute error of $Q[j]$ bounded by $\pm\rho\; r^{-j}$.

- Square–root reciprocal [6] [7] [8]: for the computation of $1/\sqrt{d}$ a residual is defined as $w[j] = r^j(1/2)(1 - d\,P[j]^2)$ where $P[j]$ is the partial result up to iteration j, defined as

$$P[j] = P[0] + \sum_{i=1}^{j} p_i r^{-i}$$

  Moreover, to obtain a simple implementation two variables are introduced: $D[j] = dP[j]$ and $C[j] = (1/2)r^{-(j+1)}d$. The algorithm proceeds as follows:

[1]If the input significand is 1.0 the algorithm provides directly the trivial result.

[2]We use a retimed recurrence as in [5].

1. Initializations: initialize $w[0]$, p_1, $Q[0]$, $D[0]$ and $C[0]$ to assure convergence. Several alternatives are possible [7].
2. Recurrence: for $j = 0$ to $g-1$

$$\begin{aligned} w[j+1] &= rw[j] - p_{j+1}D[j] - p_{j+1}^2 C[j] \\ D[j+1] &= D[j] + 2p_{j+1}C[j] \qquad (7) \\ C[j+1] &= r^{-1}C[j] \\ p_{j+2} &= SEL(rw[j+1], D[j+1]) \end{aligned}$$

   Similarly, the digits p_{j+1} take values in the set $\{-\rho(r-1), \ldots, 0, \ldots, \rho(r-1)\}$. The digit–selection function assures an absolute error of $P[j]$ bounded by $\pm\rho\ r^{-j}$.

After g iterations (we will show that g is roughly $\lceil n/(2\log_2(r))\rceil$), we obtain the approximation $k = Q[g]$ for reciprocal and $k = P[g]$ for square–root reciprocal.

3. Recurrence for the approximation

Since k is computed digit–by–digit, to avoid multipliers and to speedup the computation, we obtain $A = k(2 - d\ k)$ and $B = (k/2)(3 - d\ k^2)$ also by means of a digit–recurrence which is overlapped with the determination of k.

3.1. Reciprocal ($1/d$)

To obtain a recurrence we define

$$A[j] = Q[j](2 - d\ Q[j])$$

so that the final reciprocal approximation is $A = A[g]$ with $A[0] = Q[0](2 - dQ[0])$.

We now obtain a recurrence for $A[j]$. To eliminate variable shifts, we compute

$$E[j] = r^{2j}A[j].$$

Then,

$$\begin{aligned} E[j+1] - r^2E[j] = r^{2(j+1)}(&Q[j+1](2 - d\ Q[j+1]) \\ &-Q[j](2 - d\ Q[j])) \end{aligned}$$

$$\begin{aligned} E[j+1] - r^2E[j] = r^{2(j+1)}(&(Q[j] + q_{j+1}r^{-(j+1)}) \\ &(2 - d\ (Q[j+1]) - Q[j](2 - d\ Q[j])) \end{aligned}$$

which simplifies to

$$E[j+1] - r^2E[j] = r^{j+1}q_{j+1}(2 - dQ[j+1] - dQ[j])$$

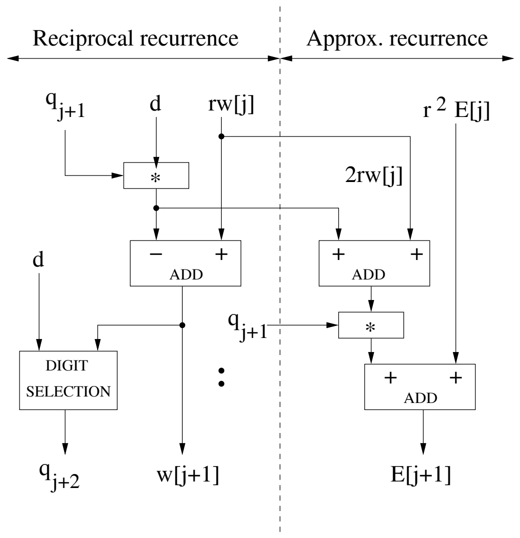


Figure 2. Recurrences for the proposed reciprocal computation.

Since $w[j] = r^j(1 - d\ Q[j])$ and $Q[j+1] = Q[j] + q_{j+1}r^{-(j+1)}$, we obtain

$$E[j+1] = r^2E[j] + q_{j+1}(2rw[j] - q_{j+1}d) \qquad (8)$$

Note that $w[j]$ and $-q_{j+1}d$ are computed as part of the w recurrence.

An alternative to the above expression is

$$E[j+1] = r^2E[j] + q_{j+1}(rw[j] + w[j+1]) \qquad (9)$$

The preferred expression will depend on the implementation details.

Figure 2 shows the overlap between recurrences w and E in the computation of the reciprocal (in this case we use expression (8)). The actual implementations depend on the radix and on the representation (for instance two's complement non redundant or redundant representation) of the different variables.

The number of iterations g is related to the final error as follows:

- The modulus of the absolute error of $k = Q[g]$ is less than $\rho\ r^{-g}$. Therefore the modulus of the relative error of k is bounded by $|\delta| \leq \rho\ r^{-g}d$.
- Since the approximation squares the relative error, we obtain a bound for final relative error

$$\epsilon \leq \rho^2 r^{-2g} d^2$$

Consequently, for a final absolute error ϵ/d not larger than 2^{-p},

$$\frac{\epsilon}{d} \leq \rho^2 r^{-2g} d \leq \rho^2 r^{-2g} < 2^{-p} \qquad (10)$$

resulting in

$$g \geq \frac{p}{2b} - \frac{\log_2(1/\rho)}{b} \qquad (11)$$

where $r = 2^b$. Since $1/2 \leq \rho < 1$, then g is bounded by

$$\left\lceil \frac{p}{2b} - 1 \right\rceil \leq g \leq \left\lceil \frac{p}{2b} \right\rceil$$

For a specific implementation, the exact value of g should be determined using (11) with the actual value of ρ. Note that since g has to be an integer, the actual absolute error is $2^{-p'}$ with $p' \geq p$ (the value of p' is obtained from (10) using the actual value of g). Therefore we distinguish between the desired absolute error (2^{-p}) and the achieved absolute error with g iterations ($2^{-p'}$).

Therefore $E[g]$ corresponds to the scaled value of $1/d$ within a precision of 2^{-p} ($A[g] = r^{-2g}E[j]$). In this way, the number of iterations (g) has been roughly halved with respect to a conventional digit–by–digit radix–r algorithm. In terms of the number of iterations, this is roughly equivalent to using a radix r^2, although the latter should lead to an implementation with a higher cycle time due to a more complex iteration.

3.2. Square–root reciprocal ($1/\sqrt{d}$)

The linear approximation for the square–root reciprocal results in

$$B = k(\frac{3}{2} - \frac{1}{2}dk^2) \qquad (12)$$

Since $k = P[g]$, similarly to the case for reciprocal we define

$$H[j] = r^{2j}P[j](\frac{3}{2} - \frac{1}{2}dP^2[j]) = r^{2j}P[j](1 + r^{-j}w[j])$$

so that $B = r^{-2g}H[g]$. We now obtain a recurrence for $H[j]$.

$$H[j+1] = r^2H[j] + r^{2(j+1)} \Big[(P[j] + p_{j+1}r^{-(j+1)}) \cdot (1 + r^{-(j+1)}w[j+1]) - P[j](1 + r^{-j}w[j])\Big]$$

Using the recurrence given in (7) results in

$$H[j+1] = r^2H[j] + p_{j+1}(2rw[j] + w[j+1] - \frac{1}{2}p_{j+1}D[j]) \qquad (13)$$

The initial condition is

$$H[0] = P[0](\frac{3}{2} - \frac{1}{2}dP^2[0])$$

As shown in Figure 3, there is an overlap between recurrences $w[j]$ and $H[j]$.The actual implementations

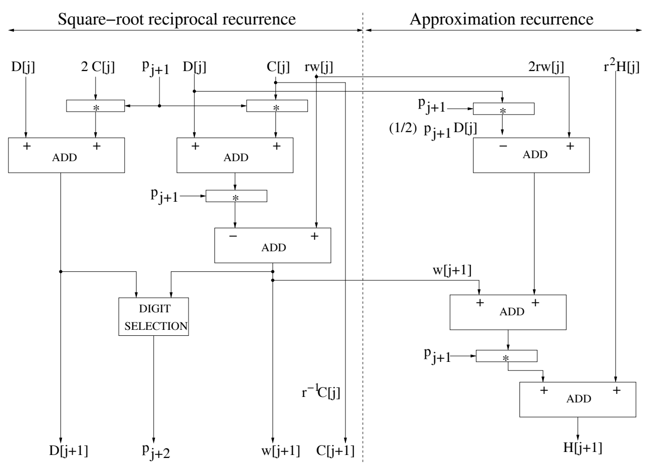


Figure 3. Proposed recurrences for square–root reciprocal computation.

depend on the radix and on the representation of the different variables. The total number of iterations is again g.

Again as for the reciprocal the number of iterations g is related to the final error as follows:

- The modulus of the absolute error of $k = P[g]$ is less or equal to $\rho\ r^{-g}$. Therefore the modulus of the relative error of k is bounded by $|\delta| \leq \rho\, r^{-g}\sqrt{d}$.
- Therefore, the relative error of B is bounded by

$$\epsilon \leq \frac{\rho^2\ r^{-2g}\ d}{2}(3 + \rho r^{-g}\sqrt{d})$$

Consequently, for a final absolute error $\epsilon/\sqrt{d}$ not larger than 2^{-p},

$$\frac{\epsilon}{\sqrt{d}} \leq \frac{\rho^2\ r^{-2g}\ \sqrt{d}}{2}(3+\rho r^{-g}\sqrt{d}) < \frac{\rho^2\ r^{-2g}}{2}(3+\rho r^{-g}) < 2^{-p}$$

Then, the condition to solve is

$$r^{-2g} < \frac{2 \cdot 2^{-p}}{\rho^2(3 + \rho r^{-g})} \qquad (14)$$

resulting in the following condition for g

$$g > \frac{p}{2b} + \frac{1}{2b}\log_2\left(\frac{\rho^2(3 + \rho\ r^{-g})}{2}\right) \qquad (15)$$

Since $1/2 \leq \rho < 1$, $r \geq 2$ and $g \geq 1$, the logarithm term is bounded by

$$-1 < -\frac{1}{b} < \frac{1}{2b}\log_2\left(\frac{\rho^2(3 + \rho r^{-g})}{2}\right) < \frac{\log_2(7/4)}{2b} < 1$$

Therefore g is bounded by

$$\left\lceil \frac{p}{2b} - 1 \right\rceil \leq g \leq \left\lceil \frac{p}{2b} + 1 \right\rceil$$

For a specific implementation, the exact value of g should be determined using (14) with the actual value of ρ. Again the achieved absolute error ($2^{-p'}$) should be less than equal to the desired absolute error (2^{-p}) since g is an integer.

The final result is $B = B[g] = r^{-2g}H[g]$, which corresponds to $1/\sqrt{d}$ computed within a precision of 2^{-p}. As before, the number of iterations has been roughly halved with respect to a conventional digit–by–digit radix–r implementation. In terms of the number of iterations, this is roughly equivalent to using a radix r^2, although the latter should lead to an implementation with a higher cycle time due to a more complex iteration.

3.3. On–the–fly–conversion of the result

The results $E[g]$ and $H[g]$ are obtained in carry–save form. As done in the corresponding digit–by–digit implementations, it is possible to convert them on–the–fly to conventional representation [2]. Since only g iterations are performed, it is necessary to obtain one additional radix–r^2 digit each iteration. Because of space limitations we do not give the detailed description of the conversion algorithm. In addition to the result, it uses two conditional forms and performs the conversion with a delay of one cycle.

4. Rounding

Although in several applications correct rounding is not necessary, it might be convenient to include this possibility. In this section we present a method to obtain correctly rounded results. This method consists of the following steps:

1. Determine whether the result obtained by the basic algorithm described in the previous section can be used directly to perform the rounding. We would compute the result with some additional accuracy so that the probability of being able to do this rounding is high.

2. In the few instances in which this rounding cannot be performed, we continue with the digit recurrence until we produce the rounding bit and the corresponding final residual. The rounding of this result is then straightforward [2].

Although this scheme leads to a variable latency operation (the higher latency case with very low probability) this should not be a hard constraint for a dynamically scheduled processor.

The delay overhead for correct rounding is composed of the time to produce the additional bits to reduce the probability of not being able to round directly and of the time to finish the iterations of the digit–by–digit part in the few cases in the result cannot be rounded directly. Moreover, little additional hardware is needed since we use the already available digit–by–digit hardware.

It can be easily shown that, since the approximation produces always a positive error, there are only three patterns that cannot be directly rounded (if we include all rounding modes). Consequently, if h additional bits are computed, the probability of not being able to round directly is $3/2^h$. Moreover, depending on the radix it might be the case that some additional bits are already obtained as part of the last iteration. For example, for radix–4, double precision (i.e. $p = 53$), it is necessary to perform 14 iterations, which produce $p' = 56$ bits. Consequently, one additional iteration would produce a pattern of $h' = p' - p + 4 = 7$ bits and a probability 0.024 of not being able to round directly. Actually, if after performing the last normal iteration we detect whether we can round we would need the additional cycle in only $3/8$ of the cases. Moreover, by checking also after the additional cycle and performing another iteration if needed, we would reduce the probability by $1/16$, with an average overhead of less than half of a cycle.

5. Radix–4 Implementation

In this section we describe the radix–4 implementation of the proposed algorithm with the digit–set $\{-2, -1, 0, 1, 2\}$. We first show the separate implementations for reciprocal and square–root reciprocal and then we consider their combination. To have a faster implementation we use a carry–save representation for the residual and for the digit–recurrence of the approximation.

5.1. Reciprocal

Figure 4 shows the radix–4 implementation for the reciprocal. The digit–selection function requires an estimation with the seven leading bits of rw and three bits of d (see [2] for details of the selection function). The digit multipliers are implemented as decoded 4–1 multiplexers (with an implicit zero output when all control inputs are zero). The result of the approximation E, represented in carry–save, is converted on–the–fly to non–redundant representation. The Figure

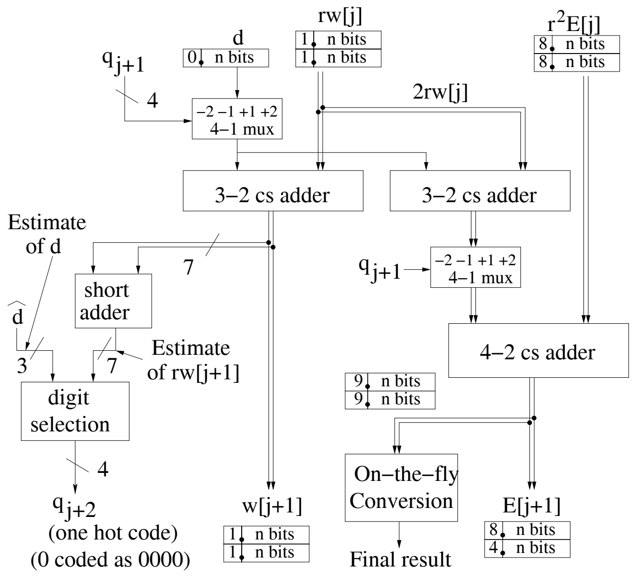


Figure 4. Radix–4 implementation of the reciprocal.

also shows the number of integer and fractional bits required for the operands.

To have the result within the interval [1,2) we use the following initializations: $Q[0] = 1$ if $d \geq 3/4$ else 2; $w[0] = 1 - d\ Q[0]$ and $E[0] = Q[0](2 - d\ Q[0])$. The algorithm requires $g + 1$ cycles (see expression (11) for the value of g): In the first cycle the initializations and the computation of q_1 are performed. Cycles 2 to g correspond to normal iterations and cycle $g + 1$ is required to finish the conversion and rounding. This results in 7 cycles ($g = 6$) for single precision and 14 cycles ($g = 13$) for double precision. In contrast, a conventional digit–by–digit algorithm with the same initializations requires 13 cycles for single precision and 27 for double precision. The algorithm was verified using numerical simulations.

5.2. Square–root reciprocal

We use an implementation of the digit–by–digit algorithm as described in [7] (see left side of Figure 5). As before, for radix–4 the digit multipliers are implemented as decoded 4–1 multiplexers. Moreover in this case the multiplication by p_{j+1}^2 reduces to a selection between the multiples 0, 1 or 4. We use also a 4–1 multiplexer in this case to control the selection directly by p_{j+1}.

In this implementation $rw[j]$ is represented in carry–save but $D[j]$ and $C[j]$ are in conventional two's complement representation. Therefore a 4–to-2 carry–save adder is required to update the residual, and a fast carry–propagate adder is used to update the value of D.

For the selection function an estimate of $rw[j+1]$ and of $D[j+1]$ is required. A short adder assimilates the more significant bits of $rw[j+1]$. The estimate of $D[j+1]$ is obtained by the addition of the most significant bits of $D[j]$ and of $2\ p_{j+1}C[j]$.

For the implementation of the approximation part (right side of Figure 5) we arranged the computation (see Equation (13)) to have a similar critical path as the one of the digit–by-digit algorithm and to allow the combined implementation with the reciprocal (see below). Moreover the result of the approximation H is converted on–the–fly. The number of integer bits of the operands would be similar to the reciprocal case. The number of fractional bits depends on the required type of rounding (see [6] and [7] for details).

To have the result within the interval [1,2) we use the following initializations: $P[0] = 1$ if $d \geq 1/2$ else 2; $w[0] = (1/2)(1 - d\,P[0]^2)$, $D[0] = d\ P[0]$, $C[0] = (1/8)d$ and $H[0] = (P[0]/2)\ (3 - d\ P[0]^2)$. The algorithm requires $g + 1$ cycles (see Expression (15) for the value of g): In the first cycle the initializations and the computation of p_1 are performed. Cycles 2 to g correspond to normal iterations and cycle $g + 1$ is required to finish the conversion and rounding. This results in 7 cycles ($g = 6$) for single precision and 15 cycles ($g = 14$) for double precision. In contrast, a conventional digit–by–digit algorithm with the same initializations requires 13 cycles for single precision and 27 for double precision. The algorithm was verified using numerical simulations.

5.3. Combined unit

The digit-by-digit algorithm for combined reciprocal and square–root reciprocal was developed in [7]. The resultant implementation is basically the square–root reciprocal implementation with the following modifications

- Single selection function for reciprocal and square–root reciprocal as obtained in [7], with the same complexity as in the case of the square–root reciprocal.
- Initialize $C[0] = 0$ for reciprocal computation. This implies that $D[j] = d$, that is, constant along the iterations.

Figure 5 shows the combined reciprocal and square–root reciprocal. For the combination of the digit–recurrence (E and H) of the linear approximation part we perform the following scheme:

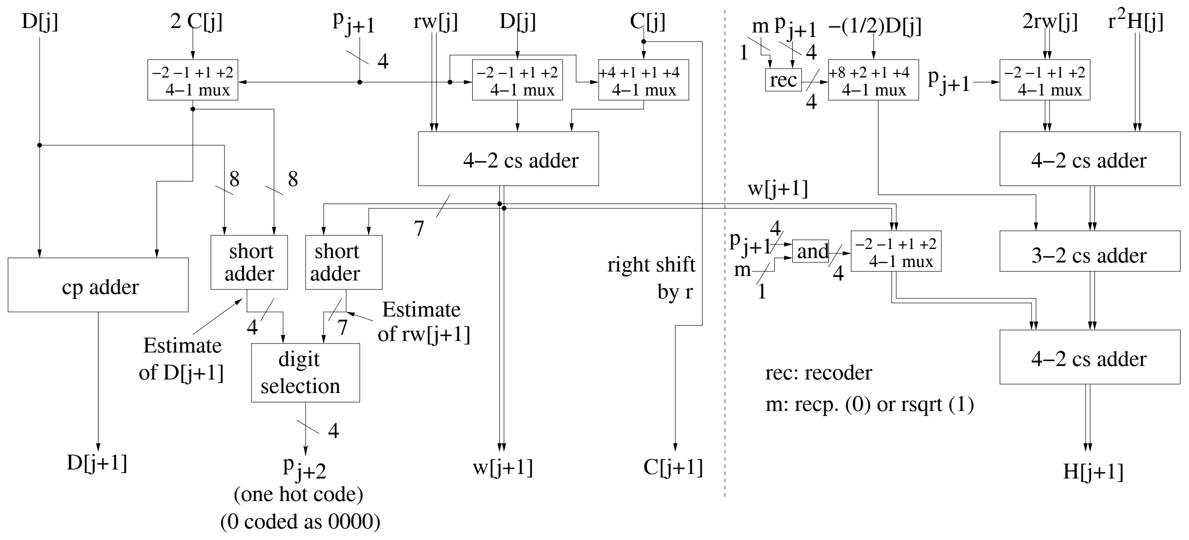


Figure 5. Radix–4 unified implementation of reciprocal and square–root reciprocal.

- Recurrence E does not add the term $w[j+1]$. Therefore we make $p_{j+1} = 0$ before the multiplexer that performs $p_{j+1}w[j+1]$, so that an output zero is produced by the multiplexer when the reciprocal operation is to be performed.

- Recurrence H requires the term $(-1/2)p_{j+1}^2 D[j]$, while for E we need $-p_{j+1}^2 D[j]$ (note that in the combined implementation p_{j+1} is used for digits of both reciprocal and reciprocal square–root, and that for reciprocal $D[j] = d$). For the combined implementation we use a recoder which inputs are p_{j+1} and a bit that indicates the operation, and which outputs select the suitable values from a 4–1 multiplexer.

Therefore the combination of both algorithms requires minor changes in the square–root reciprocal architecture, and does not affect its critical path.

6. Evaluation

In this section we present the results of the evaluation of the proposed designs and a comparison with existing digit–recurrence alternatives. We performed a synthesis of both a radix–4 reciprocal unit and a radix–4 square–root reciprocal unit using a 0.18 μm CMOS library and Synopsys. From the synthesis we estimated the critical path (including estimations at netlist level of wire load) and the gate count.

Table 1 shows the results obtained for double precision (not exactly rounded). In both cases the critical path corresponds to the digit–by-digit part, which means that apart from slight load variations, the cycle times of the proposed schemes are the same as the corresponding digit–by–digit schemes without the approximation part.

Regarding the area, since the proposed conversion has roughly the same complexity as the standard conversion, the proposed schemes increase the hardware complexity by 1.9 (reciprocal) and 1.7 (square–root reciprocal) with respect to the corresponding digit–by–digit schemes.

For reciprocal, the area–time ratios of higher radix schemes with respect to the radix–4 scheme are well known [2], and therefore it is possible to compare our design with more instances in an area–time space. Figure 6 shows the area–time ratios for digit–by–digit radix–4, radix–16 using overlapped radix–4 and very–high radix (radix 512 with prescaling and selection by rounding) schemes in comparison with the proposed design, for double precision.

For digit–by–digit square–root reciprocal we are aware of the implementations proposed in [7] [8] (radix 4) and in [6] (very–high radix). A radix–16 implementation with overlapped radix–4 iterations would be significantly more complex, because of the many conditional forms required, so this implementation was not considered. It was shown in [7] that the scheme proposed presents better area–delay figures than the proposal of [8]. Therefore we take [7] as the radix–4 design of reference. For the very–high radix implementation we assume a cost in area 1.5 times the cost of a very–high radix reciprocal unit with the same cycle time (this is a very conservative figure since the square–root reciprocal unit requires two rectangular multipliers in-

Table 1. Summary of results for the synthesized radix–4 units.

Design	Critical path buf: buffer; mux: 4-to-1 mux; reg: register; 4-2: 4-to-2 csa; sel: digit sel.	area (# nand-2) (A+B+C)*
Reciprocal	$t_{buf}(0.11) + t_{mux}(0.20) + t_{xor3}(0.16) + t_{sel}(0.61) + t_{reg}(0.28) = 1.36\ ns$	11650 (2700+4650+3500)
Square–root reciprocal	$t_{buf}(0.05) + t_{mux}(0.20) + t_{4-2}(0.27) + t_{sel}(0.97) + t_{reg}(0.28) = 1.77\ ns$	19500 (8200+7700+3500)

* A: digit–by–digit recurrence; B: approximation; C: conversion.

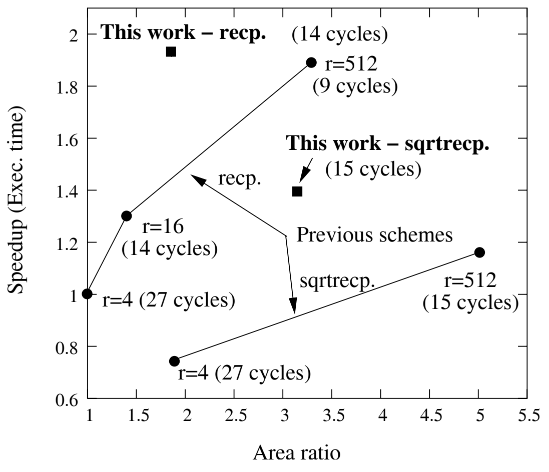


Figure 6. Area-time space.

stead of one for reciprocal, and additional adders, multiplexers, registers and a barrel shifter). We also show in Figure 6 the ratios corresponding to the square–root reciprocal, using the area and delay of the conventional radix–4 reciprocal unit as a reference. From our estimations, we conclude that the proposed designs introduce attractive points in the area–time space.

7. Conclusions

We have presented an algorithm and implementation for the computation of the reciprocal and square–root reciprocal operations. These are based on the combination of the corresponding digit–by–digit algorithm and an approximation with quadratic convergence. The latter is performed by a digit recurrence using as input the digits produced by the digit–by–digit part, allowing in this way the overlap of both parts and eliminating the need of a multiplier. As a result, the execution time is almost one half of that required when using only the digit–by–digit portion.

In contrast with the implementations that use only a digit–by–digit algorithm, because of the approximation, the result obtained cannot be correctly rounded directly (in all cases). Although this rounding is not required in some applications, such as graphics, we propose a variable–time execution that produces the correctly rounded result, with a small average overhead.

We have performed a synthesis of the implementation for radix 4 and have shown that the resulting cycle time is the same as that of the digit–by–digit unit and that, as a consequence, the execution time is almost halved. On the other hand, the addition of the approximation part almost doubles the required area. Since the resulting implementation produces four bits of the result per iteration, for reciprocal we show that it is about 50% faster than a radix–16 implementation with overlapped radix–4 stages with an increase of 30% in area. On the other hand, for square–root reciprocal the radix–16 implementation would be significantly more complex, because of the many conditional forms required.

References

[1] N. Ide et al., “2.44–GFLOPS 300–MHz Floating–Point Vector–Processing Unit for High–Performance 3–D Computer Graphics Computing”, IEEE J. of Solid–State Circuits, Vol. 35, No. 7, pp. 1025–1033, July 2000.

[2] M. Ercegovac and T. Lang, ”Digital Arithmetic”, Morgan Kaufmann Publishers, 2003.

[3] C.H. Jeong, et al. “Cost/performance Trade-off in Floating-point Unit Design for 3D Geometry Processor”, in Proc. 1st IEEE Asia Pacific Conference on ASICs, pp. 104-107, 1999.

[4] W.J. Dally, P. Hanrahan, M. Erez and T.J. Knight, “Merrimac: Supercomputing with Streams”, Supercomputing Conference, November 2003, Phoenix, Arizona (USA).

[5] A. Nannarelli and T. Lang, ”Low-Power Divider”, IEEE Trans. on Computers, vol. 48, no. 1, Jan. 1999, pp. 2-14.

[6] E. Antelo, T. Lang and J.D. Bruguera, “Computation of $\sqrt{x/d}$ in a Very–high Radix Combined Division/Square–Root Unit with Scaling and Selection by Rounding”, IEEE Trans. on Computers, vol. 47, no. 2, Feb. 1998, pp. 152–161.

[7] T. Lang and E. Antelo, “Radix–4 Reciprocal Square–root and Its Combination with Division and Square Root”, IEEE Trans. on Comput., vol. 52, no 9, Sept. 2003, pp. 1100–1114.

[8] N. Takagi, “A Hardware Algorithm for Computing Reciprocal Square Root”, in Proc. 15th IEEE Symposium on Computer Arithmetic, pp. 94–100, 2001.