Very–High Radix CORDIC Vectoring with Scalings and Selection by Rounding*

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

Tomás Lang
Dept. Electrical and Computer Engineering
Univ. of California at Irvine.
Irvine, CA 92697.
USA
tlang@uci.edu

Javier D. Bruguera
Dept. Electronic and Computer Engineering
Univ. of Santiago
Santiago de Compostela.
SPAIN
bruguera@dec.usc.es

Abstract

A very–high radix algorithm and implementation for circular CORDIC in vectoring mode is presented. As for division, to simplify the selection function, the operands are pre–scaled. However, in the CORDIC algorithm the coordinate x varies during the execution so several scalings might be needed; we show that two scalings are sufficient. Moreover, the compensation of the variable scale factor is done by computing the logarithm of the scale factor and performing the compensation by an exponential. Estimations of the delay for 32–bit precision show a speed up of about two with respect to the radix–4 case with redundant addition. This speed up is obtained at the cost of an increase in the hardware complexity, which is moderate for the pipelined implementation.

1. Introduction

The CORDIC algorithm in circular coordinates permits the calculation of rotations, as well as the trigonometric functions sin, cos, and arctan(b/a), and the modulus of a vector. A large body of work has been reported on variations of the algorithm, on implementations, and on applications. We refer the reader to [15] [19] for an overview of the algorithm, of the previous work, and for additional references. The original algorithm is radix 2 with non–redundant adders. This has been extended to the use of redundant adders and to radix 4 [4] [6] [7] [9] [12] [14] [16] [17] [18]. We consider here the extension to a much higher radix, such as radix 512.

As the radix increases, the number of iterations for a given precision is reduced accordingly, resulting in a potentially faster execution. However, two problems appear: the complexity of the selection function and the compensation of a variable scale factor. The former problem also appears for other digit–recurrence algorithms, such as division. An effective solution that has been applied there is to perform the selection by rounding [8] [11]. To allow this selection the recurrence has to satisfy certain conditions. The following two methods have been used for this:

1. Scaling the recurrence: done for division [11], square root [13], and sqrt(x/d) [2].

2. Performing selection by table in the first iterations until the conditions are satisfied: used for exponential and CORDIC rotation [1].

Very–high radix CORDIC for rotation mode has been considered in [5] [1]. Here we consider circular CORDIC in vectoring mode, which is suitable for computation of arctan(b/a) and the modulus of vector (a,b). In this case, the appropriate method for selection by rounding is scaling of the recurrence. In this sense the algorithm is similar to division. However, there is an additional difficulty due to the variation of the coordinate x in each iteration. This has the effect that several recurrence–scalings might be required. We show that two recurrence–scalings are sufficient.

The variable CORDIC scale factor, as well as the factors of the recurrence–scalings, require that the overall scaling factor be calculated and then used for the scale–factor compensation. To perform this, we calculate the logarithm of the scaling factor (by adding the logarithms of the component factors) and perform the exponential function for the compensation.

In [20] an algorithm for arctan(b/a) is presented which has some similar aspects to the one presented here. Apparently, the selection is done by truncation instead of rounding, so that, for convergence, an over–redundant digit set must be used, resulting in a more complex implementation. In addition, the final steps are done by the computation of a

* E. Antelo and J.D. Bruguera were supported in part by the Ministry of Education and Science (CICYT) of Spain under contract TIC96-1125

polynomial, requiring a full size multiplier. Moreover, the algorithm does not permit the calculation of the modulus of the vector as discussed in Section 7.

Since for the calculation of arctan(b/a) no scale–factor compensation is needed, we present this case in Sections 3 to 5. Then, in Sections 6 and 7, we extend the previous algorithm and implementation. We present pipelined and serial implementations, and have performed a rough estimation of the delay for 32–bit precision showing a substantial speedup with respect to radix–2 and radix–4 cases.

Due to space limitations we omit some details. An extended description can be found in [3].

2. Very–high radix CORDIC vectoring

The algorithm is an extension of the radix–2 algorithm (we use the y recurrence scaled by r^j as in [14] to facilitate the selection). That is, for radix $r = 2^b$, the algorithm is

$$\begin{aligned} x[j+1] &= x[j] + \sigma_j r^{-2j} w[j] \\ w[j+1] &= r(w[j] - \sigma_j x[j]) \\ z[j+1] &= z[j] + \tan^{-1}(\sigma_j r^{-j}) \end{aligned} \quad (1)$$

where $w[j] = r^j y[j]$, σ_j is selected from the set $\{-(r-1), ..., -1, 0, 1, ...(r-1)\}$ so that the following bound for convergence is satisfied

$$|w[j]| \leq r^j \tan(\theta_{max}[j]) x[j] = A[j] x[j]$$

where $A[j] = r^j \tan(\theta_{max}[j])$, and

$$\theta_{max}[j] = \sum_{i=j}^{N} \tan^{-1}((r-1)r^{-i}) + U$$

N is the index of the last microrotation, and U is the maximum angular error. To achieve n bits of precision, the number of iterations is $N = \lceil n/b \rceil$.

Initial Condition: $j = 1$, $x[1] = x_{in}$, $w[1] = r y_{in}$ and $z[0] = 0$. We consider (x_{in}, y_{in}) in first quadrant.

Final values: $x[N+1] = K\sqrt{x_{in}^2 + y_{in}^2}$, $z[N+1] = \tan^{-1}(y_{in}/x_{in})$.

Scale factor: $K = \prod_{j=1}^{N}(1 + \sigma_j^2 r^{-2j})^{1/2}$

For a faster iteration, we utilize carry–save representation for x, w, and z.

2.1. Selection by rounding

The selection is performed by rounding the residual $w[j]$. However, since the residual is in carry–save representation, as done in division, to reduce the iteration time the rounding is performed on an estimate. We consider the case in which this estimate is obtained from a truncation to t fractional bits of the carry–save representation. That is,

$$\sigma_j = \text{round}(\hat{w}) \quad (2)$$

where $\hat{w} = \lfloor w[j] \rfloor_t$ is the carry–save representation of $w[j]$ truncated to t fractional bits.

2.2. Conditions for selection by rounding

When σ_j is selected by rounding as indicated above, following a similar reasoning as in [11] for division, the condition for convergence is given by

$$\frac{1}{2} + 2^{-t} + (r-1)|1 - x[j]| < \min(1 - \frac{1}{2r},\ A[j+1]x[j+1]r^{-1}) \quad (3)$$

In [3] it is shown that $A[j+1] > r$, and since $x[j+1] \geq x[j]$, condition (3) is satisfied by

$$\frac{1}{2} + 2^{-t} + (r-1)|1 - x[j]| < \min(1 - \frac{1}{2r},\ x[j])$$

This is similar to division [11] resulting in

$$1 - \frac{1}{2r} + \frac{2^{-t}}{r-1} < x[j] < 1 + \frac{1}{2r} - \frac{2^{-t}}{r-1} \quad (4)$$

3. Pre–scaling

We use pre–scaling of x and w, so that $x[j]$ is close to one and satisfies (4). Since, in contrast to division, the value of $x[j]$ changes because of the iteration, even if the initial x is pre–scaled into the required range, a subsequent $x[j]$ might get out of the range, requiring additional pre–scalings. We now show that it is sufficient to have two pre–scalings, one before the first iteration and another before the second.

Moreover, since the first pre–scaling is only useful for the first iteration and the delay of pre–scaling might depend on the radix of the iteration, we develop the algorithm allowing a radix $R = 2^B$ ($R \leq r$) in the first iteration; we later discuss the effects of the actual value of R.

We proceed as follows:

1. Perform the first iteration in radix R. Before this iteration a pre–scaling is performed. The conditions for this pre–scaling are that the selection of σ_1 can be performed by rounding and that the second iteration (if performed by rounding after the second pre–scaling) produces $|\sigma_2| \leq r - 1$.

2. The remaining iterations are performed in radix r.

3. We determine a second pre–scaling range which allows selection by rounding in all remaining iterations. This pre–scaling interval has to accommodate the variation in $x[j]$ produced by the remaining iterations.

4. From the range of the second pre–scaling determine a lower bound on R.

5. We discuss tradeoffs for the actual value of R.

The first pre–scaling produces

$$d[1] = M_1 x_{in} \quad w[1] = M_1 (R y_{in})$$

since this iteration is radix R.

Then σ_1 is produced by rounding, as indicated in (2) and the first iteration is

$$\begin{aligned} d'[2] &= d[1] + \sigma_1 R^{-2} w[1] \\ w'[2] &= r(w[1] - \sigma_1 d[1]) \\ z[2] &= z[1] + \tan^{-1}(\sigma_1 R^{-1}) \end{aligned} \tag{5}$$

Since $d'[2]$ can be out of the pre–scaling range, we need to perform a second pre–scaling to produce

$$d[2] = M_2 d'[2] \quad w[2] = M_2 w'[2]$$

As we will not perform another pre–scaling, the iterations for $j \geq 2$ are

$$\begin{aligned} d[j+1] &= d[j] + \sigma_j r^{-2J} w[j] \\ w[j+1] &= r(w[j] - \sigma_j d[j]) \\ z[j+1] &= z[j] + \tan^{-1}(\sigma_j r^{-J}) \end{aligned} \tag{6}$$

where $r^{-J} = R^{-1} r^{-j+1}$.

Interval for second pre–scaling and bound on R

We now determine the range of $d[2]$ so that no additional pre–scaling is required. For this, we first compute Δ, a bound on the variation of $d[j]$, with respect to $d[2]$. This bound is calculated using the recurrence for $d[j]$ and the bounds of $|\sigma_j| \leq r-1$ and $w[j] < r(1-(1/2r))$. That is,

$$\begin{aligned} \Delta &= \sum_{j=2}^{\infty} r \left(1 - \frac{1}{2r}\right)(r-1) r^{-2J} \\ &= \left(1 - \frac{1}{2r}\right) r(r-1) \sum_{j=2}^{\infty} r^{-2J} < \frac{1}{R^2} \end{aligned}$$

Consequently, $d[j]$ for $j \geq 2$ will be inside the required range if we pre–scale $d'[2]$ into the range

$$1 - \frac{1}{2r} + \frac{2^{-t}}{r-1} < d[2] = M_2 d'[2] < 1 + \frac{1}{2r} - \frac{2^{-t}}{r-1} - \frac{1}{R^2} \tag{7}$$

From this expression we obtain ($b = \log_2(r)$)

$$R \geq \begin{cases} 2^{\lceil \frac{b}{2} \rceil + 1} & \text{for } t = 2 \text{ and } b \text{ odd} \\ 2^{\lfloor \frac{b}{2} \rfloor + 1} & \text{otherwise} \end{cases}$$

The tradeoffs for the selection of R are

- A small R leads to a simpler pre–scaling for $j = 1$, which might result in a smaller pre–scaling overhead.
- On the other hand, a small value of R results in a more complex pre–scaling in $j = 2$.
- R and r should be selected to minimize the number of cycles with a reasonable hardware complexity.

Interval for the first pre–scaling

To obtain the condition for the first pre–scaling we use

$$-r + \frac{1}{2} + 2^{-t} \leq w[2] < r - \frac{1}{2}$$

Taking into account that $w[2] = M_2 w'[2]$, and that M_2 does not depend on $w'[2]$, we can take the upper bound of M_2, given in (7), to obtain a bound of $w'[2]$ from the above condition, which results in

$$\frac{(-r + \frac{1}{2} + 2^{-t}) d'[2]}{1 + \frac{1}{2r} - \frac{2^{-t}}{r-1} - \frac{1}{R^2}} < w'[2] < \frac{(r - \frac{1}{2}) d'[2]}{1 + \frac{1}{2r} - \frac{2^{-t}}{r-1} - \frac{1}{R^2}}$$

On the other hand, σ_1 is obtained by rounding $w[1]$, so that

$$-\frac{1}{2} + \sigma_1(1 - d[1]) < \frac{w'[2]}{r} < \frac{1}{2} + 2^{-t} + \sigma_1(1 - d[1])$$

Note that, to obtain this expression we have taken into account that $\sigma_1 \geq 0$. Combining both expressions, we derive the following condition on $d[1]$

$$1 - \frac{C - \frac{1}{2} - 2^{-t} + \frac{C\sigma_1^2}{R^2} - \frac{C\sigma_1}{2R^2}}{\sigma_1 + C} < d[1] < 1 + \frac{C - \frac{1}{2} + \frac{\sigma_1^2 C}{R^2} - \frac{\sigma_1 C}{2R^2}}{\sigma_1 - C}$$

where $C = 1 - 1/r$. The bounds for $d[1]$ depend on the value of σ_1. Therefore, we determined the minimum for the upper bound and the maximum for the lower bound, and obtained a simple bound by means of a Taylor's series expansion, resulting in

$$1 - \frac{2\sqrt{1/2 - 2^{-t}}}{R} + \frac{5}{2R^2} + \frac{5}{Rr} < d[1] < 1 + \frac{\sqrt{2}}{R} + \frac{3}{2R^2} - \frac{6}{Rr} \tag{8}$$

Determination of the pre–scaling factors

The pre–scaling factors should assure conditions (7) and (8), that is a x coordinate within an interval close to one after each pre–scaling. Therefore, these factors correspond to an approximation of the reciprocal of x. Since the method used may be constrained by the need to compensate the pre–scaling factors to obtain the modulus, we consider the following two cases:

- **Algorithm used for angle calculation only**. In this case, it is not necessary to compensate the modulus of the vector (neither due to the pre–scaling factors nor to the CORDIC scale factor). Therefore, the method used to compute the pre–scaling factors is not constrained and any of the existing methods to obtain an approximation of the reciprocal may be used. Details of the use of linear interpolation for very–high radix division can be found in [11].

- **Algorithm used for modulus computation**. In this case, it is necessary to compensate the pre–scaling factors, which introduces certain constrains in the method of calculation. This case is discussed in Section 7.1.

4. Range extension and σ–set for first iteration

In this section we show how to extend the range in the angle to $[0, \pi/2]$ to conform to the case of radix 2. In relation to this, we determine the digit set for the first iteration. Since the iterations begin with $j = 1$ and $\sigma_1 \leq D$ (D is the maximum digit value for the first iteration) and the rest are less or equal to $r - 1$, the range of the initial angle is

$$\tan^{-1}\left(\frac{y_{in}}{x_{in}}\right) \leq \tan^{-1}(DR^{-1}) + \sum_{j=2}^{N} \tan^{-1}((r-1)r^{-J}) + U$$

The range covered depends on the value of D. To reach an angle of $\pi/2$ a large value of D could be used, but this would lead to a complex implementation. To avoid this, we have considered the following method: (1) Detect whether the angle of the initial vector is larger than $\pi/4$ ($y_{in} > x_{in}$); (2) if that is the case, exchange x and y and subtract the computed angle from $\pi/2$ (this can be accomplished by making $z[1] = \pi/2$).

For the detection, we need to compare x_{in} and y_{in}. To make a limited comparison we take advantage of the fact that the range of convergence in the angle is larger than $\pi/4$, depending on the value of D. Consequently, to perform a limited comparison we allow that after the interchange

$$y_{in} < x_{in} + 2^{-F}$$

where F is the number of fractional bits compared. We determined

$$D > R + \frac{3}{2} + 2^{-F}R \tag{9}$$

For instance, comparing $F = \log_2(R)$ bits of x and y, results in $D = R + 3$. Therefore, with values of D slightly larger than R it is possible to make a limited comparison.

5. Implementation for ArcTan(b/a)

We now consider the implementation of $\arctan(b/a)$. This uses directly the algorithm presented in previous sections, since no scaling–factor compensation is needed.

5.1. Architecture

CORDIC modules are implemented both in pipelined and sequential (word–serial) fashion. The choice between these two approaches depends on the throughput requirements and on the available area.

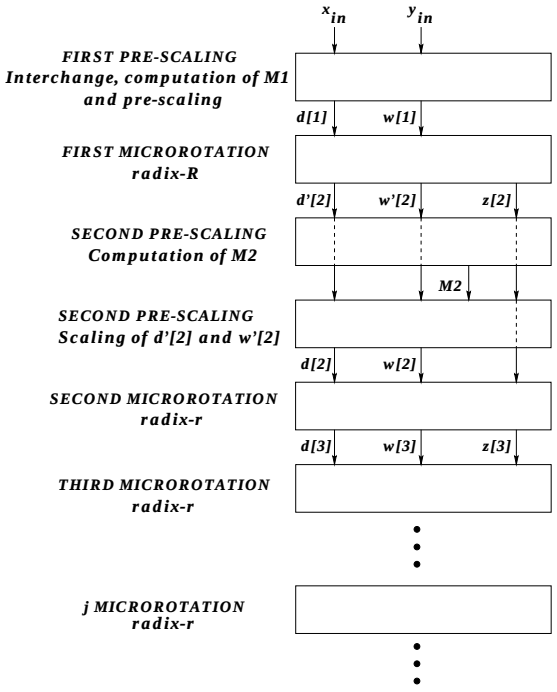

Figure 1. Unfolded algorithm.

The pipelined implementation consists on unfolding the algorithm and placing latches to define the pipeline stages. Figure 1 shows the unfolded algorithm. Details of these parts are given in Figure 2. There is considerable flexibility on where to place these latches, the actual placement depending on the required throughput and the clock characteristics. To allow for these different characteristics, we do not decide on the placement of the latches.

On the other hand, the word–serial implementation reuses the same data–path for all parts. The diagram is shown in Figure 3. The cycles are

- Cycle 1: Comparison and exchange. Calculation of $M1$ and pre–scaling. This can be done in one cycle because the radix R is small, so that the calculation of $M1$ is fast.
- Cycle 2 : First CORDIC iteration.
- Cycle 3: Calculation of $M2$.
- Cycle 4: Second pre–scaling.
- Cycle 5 to $N = \lceil \frac{n-B}{b} \rceil + 4$ (n is the precision): Other CORDIC iterations.

5.2. Evaluation and comparison

We now make a rough estimation of the execution time and of the area and compare with other algorithms. To make

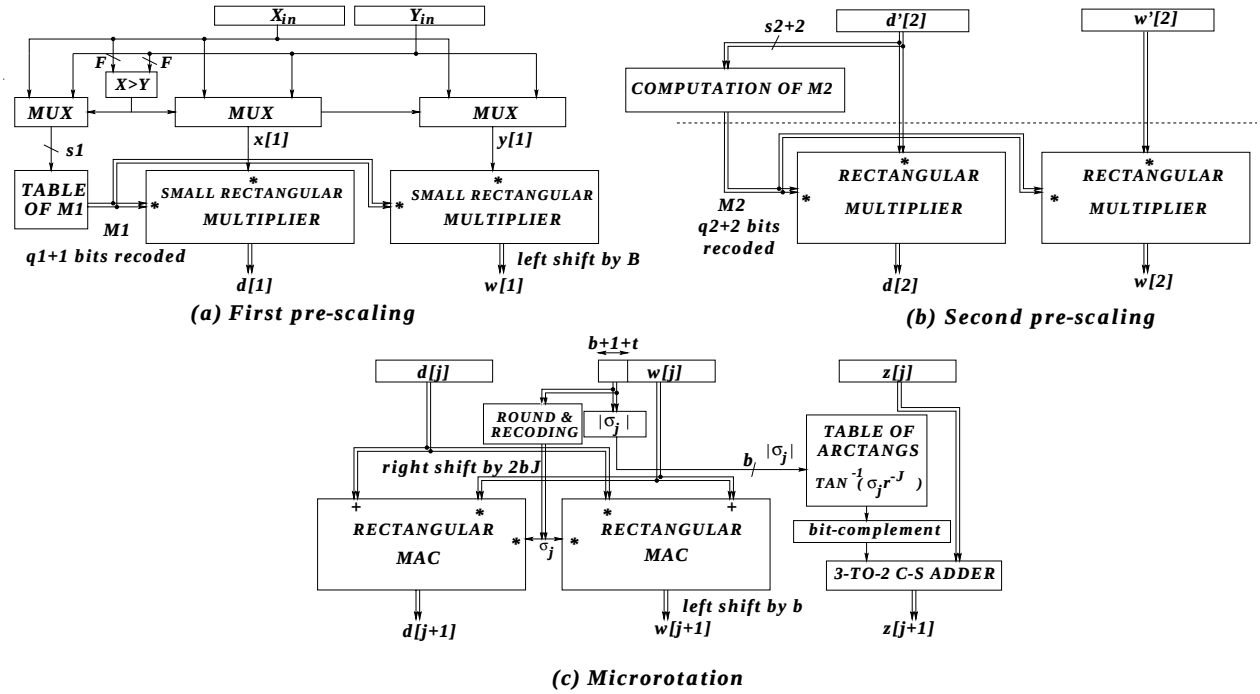

Figure 2. Implementation components.

Table 1. Comparison of pipelined implementations.

Scheme	selection	buf+mux	adder	iter. time	no. iter.	delay	speedup of vhr
radix–2 nred	0	1.5	4.0	5.5	33	180	3.6
radix–2 red	1.5	1.5	1.5	4.5	33	150	3.0
radix–4 red	4.0	1.5	1.5	7.0	17	120	2.4

this evaluation more specific, we consider the case of 32 bits precision and an implementation for $r = 512$ and $R = 32$. We use as unit of delay the delay of a full adder (t_{fa}).

We compare with the following schemes: radix–2 with non–redundant (fast carry–propagate) addition, radix–2 with redundant addition [9] [4], and radix–4 with redundant addition [18] [12].

Pipelined implementation

We first give a rough estimation of the critical path of the CORDIC iteration. We have not included the delay of the latches, since their number depends on the desired throughput. This path is composed of the following parts:

- Selection by rounding and recoding: $1.5\ t_{fa}$
- Rectangular MAC:
  - buffer + mux : $1.5\ t_{fa}$
  - adder tree (2x5+2 inputs): $4.0\ t_{fa}$

Consequently, the critical path is $7.0\ t_{fa}$.

Now we estimate the delay of the other parts as follows:

- First pre–scaling: same as CORDIC iteration.
- Second pre–scaling: two times CORDIC iteration.

Since there are four microrotations (one radix 32 and three radix 512) the total delay is $7.0 \times (1 + 1 + 2 + 3) \approx 50\ t_{fa}$.

Table 1 shows the delays of the compared schemes, indicating the corresponding speedup obtained by the very–high radix implementation.

With respect to area, for redundant representation, the part of the CORDIC iterations has essentially the same area independent of the radix, except for the tables of the angles whose size is proportional to the radix. In addition, the very–high radix scheme requires the area devoted to the pre–scaling. On the other hand, the area devoted to the wired shifts (which is substantial for low–radix implementations) is significantly reduced in the very–high radix implementation.

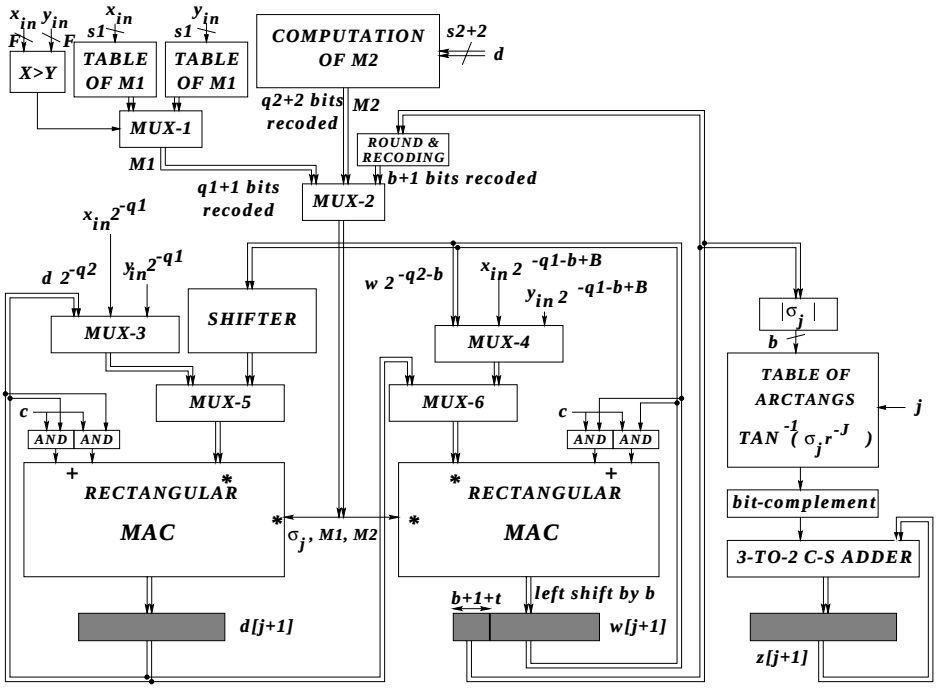

Figure 3. Word–serial architecture.

Table 2. Comparison of word–serial implementations.

Scheme	selection+buf+mux (buf+shifter+mux)	adder	reg	cycle time	no. iter.	delay	speedup of vhr
radix–2 nred	0+1.0+0.5 (1.0+1.5+0.5)	4.0	1.0	8.0	33	265	4.2
radix–2 red	1.5+1.0+0.5 (1.0+1.5+0.5)	1.5	1.0	5.5	33	180	2.9
radix–4 red	4.0+1.0+0.5 (1.0+1.5+0.5)	1.5	1.0	8.0	17	135	2.2

Word–serial implementation

We estimate the delay of one cycle as that of a CORDIC iteration and estimate that the rest of the cycles can be implemented in this cycle time. Note that to achieve this in the first cycle the table of M1 is duplicated. The delay is

(1.5) recoding + (0.5) mux + (6) MAC + (1.0) reg = 9.0 t_{fa}

In this case the size of the MAC is determined by the second pre–scaling factor, so that the MAC has larger latency than that for the pipelined implementation. Since there are 7 cycles the execution time is $63\ t_{fa}$.

We compare with the same schemes as for the pipelined implementation. The results are given in Table 2.

In the word–serial implementation the very–high radix scheme occupies a larger area mainly because of the MACs, the tables for the angles, and the calculation of the pre–scaling factors. The tradeoff between speed up and area can be varied by choosing the radix.

6. Scale–factor compensation for the modulus

The algorithm described in the above sections introduces a scale factor in the modulus of the vector. This factor is the result of the pre–scaling factors, used to make selection by rounding, and the scale factor introduced by the CORDIC algorithm. Note that all of these factors depend on the input data and, therefore, are not constant.

In this section we present an algorithm to compensate all the scale factors (pre–scaling factors and CORDIC scale factor). This is needed only for modulus computation.

To perform the compensation, we use a scheme that con-

sist in computing the logarithm of the total scale factor to be compensated and perform the exponential function. This scheme was first used in [5] and later used in [1]. In [4] a similar scheme was used for a radix–2 algorithm.

In this paper we improve the computation of the exponential function. In the following sections we describe how to perform these two steps.

We do not consider other methods to compensate a variable scale factor (for instance square–root and division [9], or look–up tables and multiplications [18]) since we estimate that they increase the latency and/or the area [3].

6.1 Calculation of logarithm of scale factor

The expression of the scale factor to compensate is

$$T = M_1 M_2 \prod_{j=1}^{N} (1 + {\sigma_j}^2 r^{-2J})^{1/2}$$

where $r^{-J} = R^{-1} r^{-j+1}$.

Therefore, the computation of the logarithm of the scale factor results in

$$\ln(T) = \ln(M_1) + \ln(M_2) + \frac{1}{2} \sum_{j=1}^{N} \ln(1 + {\sigma_j}^2 r^{-2J}) \quad (10)$$

This involves the computation of the logarithm of each factor and addition of all the logarithms. We have determined that the range of T is within $(1/2, 5/2)$. Therefore, the range of the logarithm is within $(-\ln(2), \ln(5) - \ln(2))$. Since the argument of the exponential function is $\ln(1/T)$, its range should be $(-\ln(5) + \ln(2), \ln(2))$.

6.2. Compensation by exponential

In this section we propose an algorithm for the computation of $He^{\ln(1/T)}$, where H is the x coordinate after the vectoring operation. This algorithm corresponds to improvements over the algorithms proposed in [5] and [1].

The basic iteration to compute the exponential is [5]

$$\begin{aligned} v[i+1] &= v[i] + e_i \, r^{-i} v[i] \\ c[i+1] &= r(c[i] - B_i[e_i]) \end{aligned} \quad (11)$$

where $i \geq 1$, $v[1] = H$, $c[1] = -\ln(T)$, $-(r-1) \leq e_i \leq r-1$ and

$$B_i[e_i] = r^i \ln(1 + e_i r^{-i}) \quad (12)$$

The selection of e_i is performed so that $r^{-i} c[i]$ tends to 0. The result is

$$v[S+1] = He^{\ln(1/T)} = \frac{H}{T}$$

where S is the number of iterations of the exponential algorithm. Note that we use r for the radix, but this does not imply the use of the same radix as in the vectoring.

As in [1], we consider selection by rounding. However, in this case we perform the selection as follows

$$e_i = \text{round}(\hat{c}) \quad (13)$$

where $\hat{c} = \lceil c[i] \rceil_t$ is the carry–save representation of $c[i] + 2^{-t}$ truncated to t fractional bits. Note that, in contrast to the vectoring case, for this algorithm, to perform the rounding we add always one in position t. This selection leads to a reduction in the maximum residual for the next iteration.

In [3] we obtain the conditions for convergence using selection by rounding. Using similar conditions of convergence as in the vectoring case, we determined that

- For iterations with $i \geq 3$ convergence is assured by using selection by rounding and $-(r-1) \leq e_i \leq r-1$.
- For iteration $i = 2$, we achieve convergence with selection by rounding if $-(r-2) \leq e_2 \leq r-1$.
- Iteration $i = 1$ does not converge with selection by rounding.

For the first iteration, we propose to use a table for selection. The following two aspects determine the parameters of the table:

- As we have seen in Section 6.1, the domain of the exponential function has a positive and negative part. Since the first selection is by table, negative arguments lead to a large selection table due to the form of the logarithm function for argument less than one. To have always a positive argument we propose to compute

  $$H \, e^{\ln(1/T)} \, 2^k \, 2^{-k} = H \, e^{\ln(1/T) + k \ln(2)} \, 2^{-k}$$

  where $k = \lceil \log_2(T_{max}) \rceil = 2$.

  This assures a positive argument for the exponential function, by just performing a right shift by two positions. The range to be covered by the exponential is now $(3\ln(2) - \ln(5), 3\ln(2))$.

- Since the selection of the second iteration is performed by rounding with $-(r-2) \leq e_2 \leq r-1$, the residual $c[2]$ should verify $-(r-2) - 1/2 + 2^{-t} \leq c[2] < r - 1/2$. This assures convergence in the remaining iterations.

We report in Section 7.1 the results we have obtained for this table.

7. Calculation of ArcTan(b/a) and Modulus

We now combine the implementation of Section 5 with the scale–factor compensation of Section 6 to obtain a module for the calculation of both arctan(b/a) and the modulus of the vector (a,b).

7.1. Architecture

We first discuss the size of the required tables for the compensation and then we consider a pipelined and a word–serial implementation.

Size of the required tables

In this section we discuss the size for the required tables, for the compensation of the total scale–factor. The tables correspond to the calculation of the logarithm of the scale factor, to the selection in the first iteration of the exponential, and to the constants for the exponential iterations.

Tables for the computation of the logarithm. As discussed in Section 6.1, we include the compensation of the pre–scaling factors as a part of the CORDIC scale–factor compensation using the logarithm–exponential method. This requires the computation of the logarithm of the pre–scaling factors with a precision of about n bits. Since the pre–scaling factors have about $b = \log_2(r)$ bits, which is much smaller than n, an effective method for the computation of the logarithms is a table–lookup. We address the table directly with the bits of x. Moreover, since the error in the computation of the pre-scaling factor is minimum when a table look-up is used, this method reduces the number of address bits of the tables.

In summary, in this case we use two tables for each pre–scaling factor, one for the pre–scaling factor and one for the logarithm. In [3], we determined the effect of the algorithm parameters (such as R, r and t) on the number of bits involved. There are various solutions to the number of input bits of these tables, the choice depending on a trade-off among many characteristics. In any case the number is about $B + 1$ bits for $\ln(M_1)$, and $b + 2$ bits for $\ln(M2)$.

Moreover, the computation of the logarithm requires the terms $\ln(1 + \sigma_j^2 r^{-2J})$, which are obtained from a table of b input bits for each j.

Table for selection in the first iteration of the exponential. We have determined in [3] the size of the table for the first selection in the exponential algorithm. This size is of about (there are different possibilities) $b + 2$ input bits and $b + 3$ output bits.

Tables of constants for the exponential. Iteration c needs constants of the form $\ln(1 + e_i r^{-i})$. Since the logarithm function is not symmetrical, it requires a table of $b+1$ input bits for each iteration.

Pipelined implementation

As shown in Figure 4(a), the pipeline of Section 5.1 is extended to include the scale–factor compensation of x. This extension consists of the following parts:

- The calculation of the logarithm of the scale factor. As described by expression (10) this calculation is performed by the sum of the component terms. As indicated previously, these component terms are obtained from tables. In Figure 4(b) we show the implementation of one iteration of this calculation.
- The recurrence to obtain the digits of the decomposition of the logarithm according to expression (11). The selection is done by rounding, except for the first digit, which is obtained from a table (with $b + 2$ input bits). The implementation of one of these iterations is shown in Figure 4(c).
- The recurrence using the digits above to perform the compensation. This is not shown since it is similar to the x CORDIC data–path.

Note that since the scale factor of the required precision depends only on about the first half of the σ_j digits, the calculation of the logarithm is performed only in those iterations. Moreover, in the last iterations x does not change, making it possible to overlap the compensation with the rest of the CORDIC iterations. Specifically, we have determined in [3] that the last iteration in x to obtain H (the value of the scaled modulus within the precision) is

$$h = \left\lceil \frac{n - 2B}{2b} \right\rceil + 1$$

To avoid the delay of the table for selection of e_1, this selection is performed using an estimate of $\ln(1/T)$. Finally, the last portion of the compensation can be performed by a rectangular multiplier (termination by linear approximation of the exponential).

Word–serial implementation

As shown in Figure 4(d), to the implementation for the calculation of arctan(b/a) it is necessary to add a data–path for the summation of the logarithms. This same data–path is then used to obtain the digits e_i. The actual multiplication by $(1 + e_i r^{-i})$ is done using the x data–path. Therefore, the shifter should be extended to perform also shifts by r^{-i}. The same considerations as for the pipelined implementation reduce the amount of tables required and allow the overlapping of the CORDIC iterations and the scale–factor compensation.

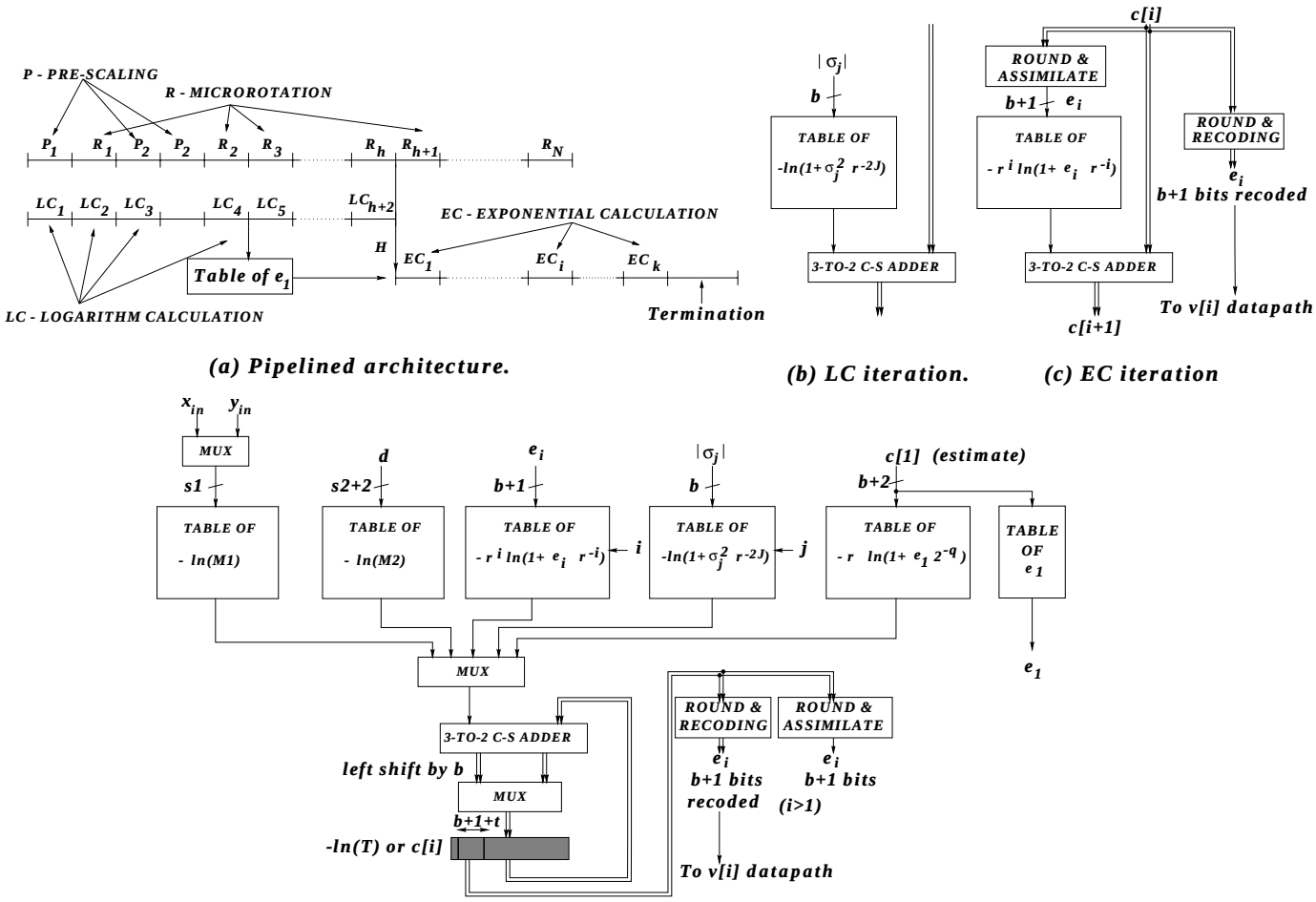

Figure 4. Implementations.

7.2. Evaluation and Comparison

We now extend the evaluation performed for the arctan case to include also the modulus. As there, we consider the case for 32 bits precision, $r = 512$, and $R = 32$.

For the pipelined implementation, after the third CORDIC iteration it is possible to initiate the compensation. Moreover, after the second iteration of the exponential the compensation is terminated by a multiplier (linear approximation). We assume that the delay of an iteration of the compensation as well as the delay of the multiplier is the same as that of the CORDIC iteration. Consequently, the total delay of the calculation of the modulus is equivalent to the delay of 9 CORDIC iterations, which corresponds to $9 \times 7.0 = 63\ t_{fa}$. The comparison of delays is as follows:

- For the radix–2 implementation with non–redundant adder, the compensation can be done totally in parallel with the angle calculation. Therefore, the delay remains in $180\ t_{fa}$, with a speed up of 2.9.
- For the radix–2 implementation with redundant adder, to simplify the compensation, it is convenient to use an implementation with a constant–scaling factor, such as that of [14]. For 32 bits, this requires 6 additional CORDIC iterations, for a total of 39 iterations and the compensation can be performed in parallel with these iterations. The total delay is $39 \times 4.5 = 175\ t_{fa}$, resulting in a speedup of 2.8.
- For radix 4 (with redundant adder) [18], the scaling factor is variable. The scale factor is computed by an initial table and linear approximations, and the compensation is performed by a multiplier. Both operations are performed in parallel with the angle calculation. Therefore the total delay is $17 \times 7.0 \simeq 120\ t_{fa}$, resulting in a speedup of 1.9.

The word–serial implementation has about the same crit-

ical path as the architecture for angle calculation of Section 5.1. However in this case we have 10 iterations (for the word–serial we do not terminate the exponential with a multiplier, resulting in one additional iteration, as compared with the pipeline architecture). Therefore, this implementation presents a total delay of about $10 \times 9.0 = 90\ t_{fa}$.

Making similar considerations for the radix–2 algorithms, results in an speedup of about 3.0 with respect to the radix–2 algorithm with non–redundant adder, and 2.4 with respect to the radix–2 algorithm with redundant adder. For the radix–4 algorithm, the compensation is carried out by a radix–4 division algorithm, that begins when the scaled module is obtained within the precision, so that certain overlap exists between the microrotations and the compensation. The resulting total number of iterations is 33 (25 microrotations and 8 iterations of the division used for compensation, which do not overlap with the microrotations), and a total delay of 200. The speedup in this case is of about 2.2.

8. Conclusions

We have presented a very–high radix algorithm and implementation for the circular CORDIC in vectoring mode, which is used to compute arctan(b/a) and the modulus of vector (a, b). In order to use the simple selection by rounding, two pre–scalings are performed, one before the first iteration and another after it. Moreover, to reduce the overhead of the pre–scaling and adapt to the required precision, the first iteration is done using a smaller radix than the rest. The main requirement for this algorithm, as compared with radix 2 and radix 4, corresponds to an increase in the size of the tables for storing the elementary angles, the need for tables for the pre–scaling factors as well as the need of rectangular multipliers instead of adders in the serial implementation (for the pipelined implementation we estimate that the total cost of the rectangular multipliers and of the adders is comparable). Moreover, when the modulus is required, the scale–factor compensation is performed by a logarithm–exponential approach, so that additional tables are needed. The number of input bits to each of these tables is about the logarithm of the radix, so that we estimate that the implementation is reasonable up to a radix around 1024.

We have performed a rough evaluation for 32–bit precision of both a pipelined and a word–serial implementation and have shown a substantial speed up with respect to radix–2 and radix–4 implementations. This module complements the unit for CORDIC rotation presented in [1].

References

[1] E. Antelo, J. Bruguera, T. Lang, J. Villalba and E. Zapata, "High–radix CORDIC rotation based on selection by rounding", Lecture Notes in Computer Science (EUROPAR-96), pp. 155–164, 1996.

[2] 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 Transactions on Computers, vol. 47, no. 2, pp. 152–161, Feb. 1998.

[3] E. Antelo, T. Lang, J.D. Bruguera, "Very high–radix CORDIC vectoring with pre–scaling and selection by rounding – Extended version ", Internal Report, Dept. Electrical and Computer Eng., University of California at Irvine, USA, September 1998 (available in the web at http://www.eng.uci.edu/numlab/archive).

[4] J.C. Bajard, S. Kla and J.M. Muller, "BKM: A new hardware algorithm for complex elementary functions", IEEE Transactions on Computers, pp. 955–964, August 1994.

[5] P. W. Baker, "Parallel multiplicative algorithms for some elementary functions", IEEE Trans. on Computers, no. 3, pp. 322–325, 1975.

[6] H. Dawid and H. Meyr, "The differential CORDIC algorithm: constant scale factor redundant implementation without correcting iterations", IEEE Trans. on Comput., vol. 45, no. 3, pp. 307–318, 1996.

[7] J. Duprat and J.M. Muller, "The CORDIC algorithm: New results for fast VLSI implementation", IEEE Transactions on Computers, vol. 42, no 2, pp. 168-178, Feb. 1993.

[8] M.D. Ercegovac, "A higher radix division with simple selection of quotient digits", In Proc. 6th IEEE Symposium on Computer Arithmetic, pp. 94–98, 1983.

[9] M.D. Ercegovac and T. Lang, "Redundant and on–line CORDIC: Application to matrix triangularization and SVD", IEEE Transactions on Computers, vol. 39, no 6, pp. 725-740, June 1990.

[10] M.D. Ercegovac and T. Lang, "Division and square–root: Digit recurrence algorithms and implementations", Kluwer Academic Publishers, Norwell, MA, 1994.

[11] M.D. Ercegovac, T. Lang and P. Montuschi, "Very–High radix division with pre–scaling and selection by rounding", IEEE Transactions on Computers, vol. 43, no. 8, pp. 909–918, August 1994.

[12] Y.H. Hu and H.H.M. Chern, "A novel implementation of CORDIC algorithm using backward angle recoding (BAR)", IEEE Trans. on Computers, vol. 45, no. 12, pp. 1370–1378, December 1996.

[13] T. Lang and P. Montuschi, "Very–High radix combined division and square–root with pre–scaling and selection by rounding", Proc. 12th IEEE Symposium on Computer Arithmetic, pp. 124–131, 1995.

[14] J.-A. Lee and T. Lang, "Constant–Factor redundant CORDIC for angle calculation and rotation", IEEE Transactions on Computers, vol. 41, no 8, pp. 1016-1025, Aug. 1992.

[15] J.M. Muller, "Elementary functions: algorithms and implementations", Birkhauser Pub., 1997.

[16] N. Takagi, T. Asada and S. Yajima, "Redundant CORDIC methods with a constant scale factor for sine and cosine computation", IEEE Transactions on Computers, vol. 40, no. 9 pp. 989-995, Sept. 1991.

[17] D.Timmermann, H. Hahn and B. J. Hosticka, "Low latency time CORDIC algorithms", IEEE Transactions on Computers, vol. 41, no 8, pp. 1010-1015, Aug. 1992.

[18] J. Villalba, E. Antelo, J.D. Bruguera and E.L. Zapata, "Radix–4 Vectoring CORDIC Algorithm and Architectures", J. of VLSI Signal Processing for Signal, Image and Video Tech., June 1997.

[19] Web site http://devil.ece.utexas.edu/cordic.html. References to CORDIC arithmetic.

[20] W.F. Wong y E. Goto, "Fast hardware–based algorithms for elementary function computations using rectangular multipliers", IEEE Transactions on Computers, Vol. 43, no. 3, March 1994.