Journal of VLSI Signal Processing 19, 227–241 (1998)
© 1998 Kluwer Academic Publishers. Manufactured in The Netherlands.

Parallel Compensation of Scale Factor for the CORDIC Algorithm

J. VILLALBA
Department of Computer Architecture, University of Málaga, Málaga, Spain

TOMÁS LANG
Department of Electrical and Computer Engineering, University of California at Irvine, USA

E.L. ZAPATA
Department of Computer Architecture, University of Málaga, Málaga, Spain

Received November 13, 1996; Revised September 15, 1997

Abstract. The compensation of scale factor imposes significant computation overhead on the CORDIC algorithm. In this paper we present two algorithms and the corresponding architectures (one for both rotation and vectoring modes and the other only for rotation mode) to perform the scaling factor compensation in parallel with the classical CORDIC iterations. With these methods, the scale factor compensation overhead is reduced to a couple of iterations for any word length. The architectures presented have been optimized for conventional and redundant arithmetic.

1. Introduction

The CORDIC algorithm (COordinate Rotation DIgital Computer) was introduced to compute trigonometric functions and generalized to compute linear and hyperbolic functions [1, 2]. It is an iterative algorithm suitable for VLSI implementation because it employs only adders and shifters and it has a wide application field. Special attention has been paid by different researchers to the improvement of the algorithm in the last few years, as referenced in [3].

By means of the CORDIC algorithm, a vector (x, y) is rotated an angle (rotation mode) or it is taken to the coordinate axis (vectoring mode). The algorithm is based on rotations over prefixed known elementary angles. These operations can be performed in linear, circular and hyperbolic coordinate systems. For clarity in the exposition, the hyperbolic case is not considered here, but the procedure proposed in this paper is valid also for the hyperbolic case. Therefore, in what follows we consider circular coordinates, in which basic iteration or *microrotation* is

$$\begin{aligned} x(i+1) &= x(i) - \sigma_i \cdot 2^{-i} \cdot y(i), \\ y(i+1) &= y(i) + \sigma_i \cdot 2^{-i} \cdot x(i), \\ z(i+1) &= z(i) - \sigma_i \cdot \tan^{-1}(2^{-i}), \end{aligned} \qquad (1)$$

where $(x(0), y(0))$ are the initial coordinates of the vector and the z coordinate accumulates the angle. The coefficient $\sigma_i \in \{-1, +1\}$ specifies the direction of each microrotation. $n+1$ iterations are needed to maintain n bit of precision.

Each iteration introduces a scaling over both coordinates given by the expression $K_i = \sqrt{1+\sigma_i^2 \cdot 2^{-2i}}$, and thus, after $n+1$ iteration the final x and y coordinates are scaled by the factor

$$K = \prod_{i=0}^{n} \sqrt{1+\sigma_i^2 \cdot 2^{-2i}} \qquad (2)$$

Therefore, it is necessary to compensate the scale factor by multiplying $x(n)$ and $y(n)$ with K^{-1}, and this imposes significant overhead whose minimization has been attempted by different researchers [4–8]. As discussed further in Section 4, these methods produce an increase in the latency of the algorithm that is dependent on n and, in some cases, increase the complexity of the CORDIC iteration. In this paper, we present two methods to reduce this overhead.

A preliminary version of parallel compensation of scale factor can be found in [9]. The present paper is an improvement and an extension of that work, where the hardware requirements have been significantly decreased and the use of redundant arithmetic has been incorporated. We present two algorithms to perform the parallel compensation of the scale factor: the carry-analysis method, which is applicable to rotation and vectoring modes, and the double-rotation method, which is only applicable to rotation mode. The associated architectures are optimized for conventional and redundant arithmetic. This way, the scale factor compensation overhead is only one or two iterations.

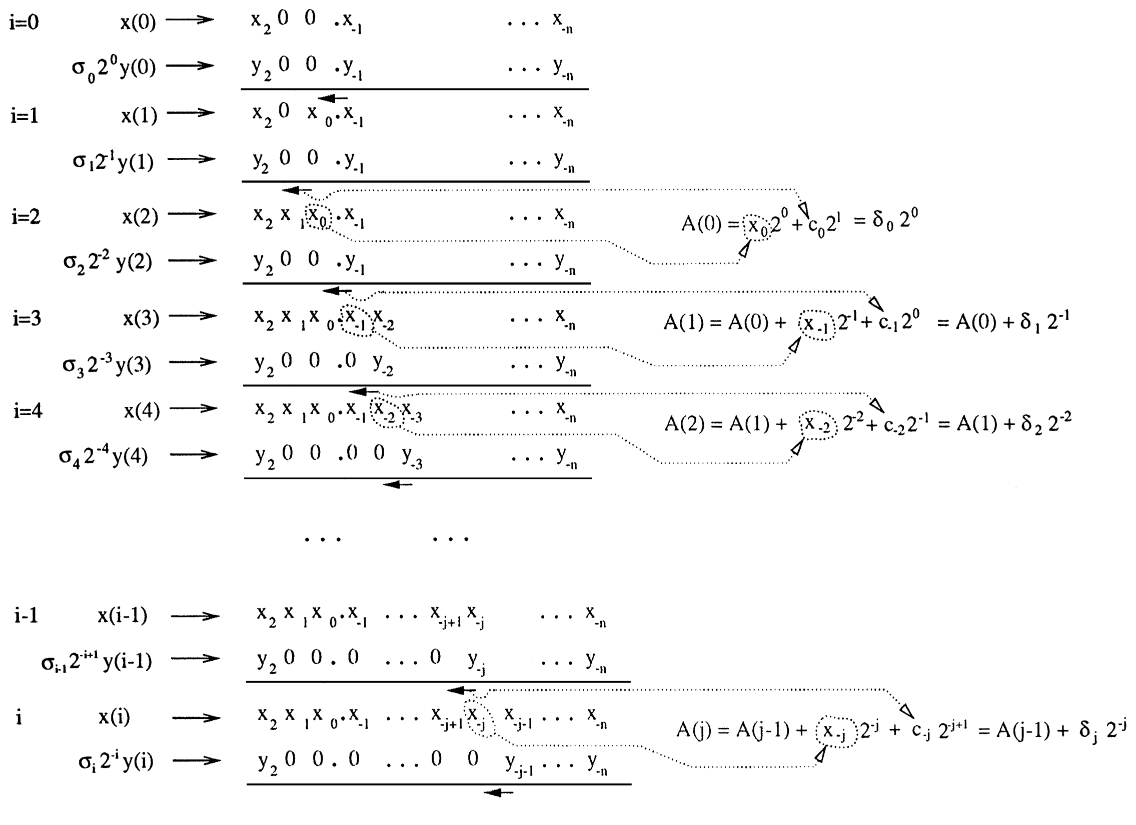
2. Parallel Compensation by Means of the Carry-Analysis Method

To perform the *scale factor compensation* we can multiply the values obtained in iteration n by K^{-1}. The *carry-analysis method* is based on obtaining enough information in each iteration to perform the multiplication in a distributed way

$$x(n+1)\cdot K^{-1} = \sum_{j=0}^{n} \delta_j \cdot 2^{-j} K^{-1}, \qquad (3)$$

where the value of the digit δ_j is determined from a few bits of $x(i)$, with $i = j + 2$, as explained later.

To make the understanding of the method easier, Fig. 1 shows the evolution of coordinate x along the CORDIC iterations, where the sign-and-magnitude representation was selected, although the procedure may be extended to any representation, as discussed later. The initial values of $x(0)$ and $y(0)$ are fractions and two bits are used for the integer part (x_1x_0 bits)



i=0
x(0)
σ0 2^0 y(0)
i=1
x(1)
σ1 2^-1 y(1)
i=2
x(2)
σ2 2^-2 y(2)
i=3
x(3)
σ3 2^-3 y(3)
i=4
x(4)
σ4 2^-4 y(4)
i-1
x(i-1)
σi-1 2^-i+1 y(i-1)
i
x(i)
σi 2^-i y(i)
A(0) = x0 2^0 + c0 2^1 = δ0 2^0
A(1) = A(0) + x-1 2^-1 + c-1 2^0 = A(0) + δ1 2^-1
A(2) = A(1) + x-2 2^-2 + c-2 2^-1 = A(1) + δ2 2^-2
A(j) = A(j-1) + x-j 2^-j + c-j 2^-j+1 = A(j-1) + δj 2^-j

Figure 1. Evolution of the coordinate x along the CORDIC iterations.

because of the scaling introduced by the CORDIC algorithm. x_2 and y_2 are the sign bits.

Note in the figure that the nonzero MSB (Most Significant Bit) of the y expressions shifts one bit to the right in every iteration starting from $i = 2$ (i.e., for $i = 3$ the nonzero MSB is $y_{-2}(3)$, and for $i = 4$ it is $y_{-3}(4)$; the first two iteration have a special bound that will be studied later). Therefore, we will see that it is possible to obtain information about the value of the MSBs of $x(i+1)$ if we know the value of the MSBs of $x(i)$ and the *carry* or *borrow* generated in the position of the nonzero MSB of y.

We now call $A(j)$ the sum of the weighted MSBs of $x(i)$ up to the bit 2^{-j}, with $j = i - 2$:

$$A(j) = \sum_{k=1}^{k=-(j)} x_k(i)2^k \tag{4}$$

For example, in Fig. 1 we have $A(\mathbf{2}) = x_1(4)2^1 + x_0(4)2^0 + x_{-1}(4)2^{-1} + x_{-2}(4)2^{-\mathbf{2}}$. Then, we can write that

$$\begin{aligned} A(2) &= [x_1(4)2^1 + x_0(4)2^0 + x_{-1}(4)2^{-1}] \\ &\quad + x_{-2}(4)2^{-2} \\ &= [x_1(3)2^1 + x_0(3)2^0 + x_{-1}(3)2^{-1}] \\ &\quad + c_{-2}2^{-1} + x_{-2}(4)2^{-2} \\ &= A(1) + c_{-2}2^{-1} + x_{-2}(4)2^{-2}, \end{aligned} \tag{5}$$

where c_{-2} is the outgoing *carry* or *borrow* of the bit of weight 2^{-2} when the operation $x(4) = x(3) + \sigma_3 2^{-3} y(3)$ is performed (see Fig. 1).

Taking into account the development of the variable A in Fig. 1 we can deduce the following recurrence:

$$A(j) = A(j-1) + c_{-j}2^{-j+1} + x_{-j}(i)2^{-j}. \tag{6}$$

Expression (6) can be written as

$$A(j) = A(j-1) + \delta_j 2^{-j}, \tag{7}$$

where

$$\delta_j = 2c_{-j} + x_{-j}(i). \tag{8}$$

These expressions have been obtained for a sign-and-magnitude representation, where $x_{-j}(i) = \{0, 1\}$ and c_j is the *carry* (for additions) or *borrow* (for subtractions) and then $c_{-j} = \{0, \pm 1\}$, and therefore $\delta_j = \{0, \pm 1, \pm 2, 3\}$. However, we can follow the same reasoning if we use signed-digit representation in such a way that we obtain the same expressions. Since each digit has its own sign, the allowed values for $x_{-j}(i)$ and c_{-j} are $\{0, \pm 1\}$ and therefore $\delta_j = \{0, \pm 1, \pm 2, \pm 3\}$.

In true-and-complement representation we can follow a similar reasoning, but the final expression needs to be slightly modified due to the particular way of representing negative numbers. If $\sigma_{i-1}2^{-(i-1)}y(i-1)$ is negative it has all *ones* from the MSB up to the bit of weight 2^{-j+1}. Then, we can write

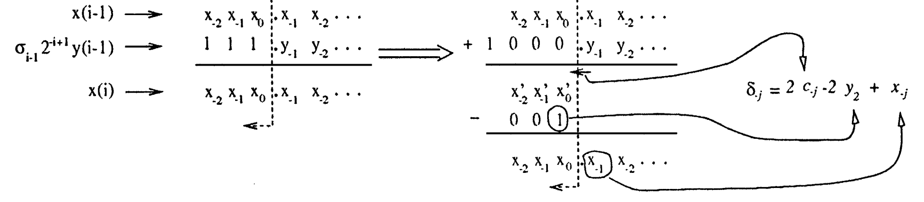
$$\sigma_{i-1}2^{-(i-1)}y(i-1) = -2^{-j+1} + \sum_{m=j}^{n} y_{-m}(i-1)2^{-m}. \tag{9}$$

Figure 2 shows this for $i = 3$. Following this procedure for the negative numbers we deduce the expression

$$\delta_j = 2(c_{-j} - y_2) + x_{-j}(i), \tag{10}$$

where c_{-j} is now the outgoing *carry* of bit $x_{-j}(i)$ and y_2 is the sign bit of $\sigma_{i-1}2^{-(i-1)}y(i-1)$ (see Fig. 2). Because the allowed values of $x_{-j}(i)$, c_{-j} and y_2 are $\{0, 1\}$, the range of δ_j is $\{0, \pm 1, \pm 2, 3\}$ (note that $c_j \in \{0, 1\}$ since we use the complement representation for the negative number).

The initial value of the variable A is 0 except if $x(1) < 0$ and true-and-complement representation of numbers is used. Because the first iteration of the recurrence (7) is carried out when $i = 2$, we must take into account the value of the MSB of $x(1)$. In true-and-complement representation this bit represents the value -2 if $x(1)$



x(i-1)
$\sigma_{i-1}2^{-i+1}y(i-1)$
x(i)
$x_{-2}\ x_{-1}\ x_0\ .x_{-1}\ x_{-2}\dots$
$1\ 1\ 1\ .y_{-1}\ y_{-2}\dots$
$+\ 1\ 0\ 0\ 0\ .y_{-1}\ y_{-2}\dots$
$x'_{-2}\ x'_{-1}\ x'_0$
$-\ 0\ 0\ 1$
$\delta_j = 2\,c_{-j} - 2\,y_2 + x_{-j}$

Figure 2. Addition of a negative number in true-and-complement representation ($i = 3$).

is a negative number, and therefore, we must initialize the variable A to -2. Therefore, $A(-1) = -2x_2$.

If we write the expression (7) from $j = 0$, we have

$$A(n) = \sum_{j=0}^{n} \delta_j \cdot 2^{-j} \tag{11}$$

Because $A(n)$ has taken into account the contribution up to the bit of weight 2^{-n}, which is the LSB (Least Significant Bit) of $x(n+1)$, we can write

$$A(n) = x(n+1). \tag{12}$$

We now take advantage of the recurrence of $A(i)$ to carry out the compensation of the scale factor. The value of the compensated coordinate x is

$$x(n+1) \cdot K^{-1} = A(n) \cdot K^{-1} = \sum_{j=0}^{n} \delta_j \cdot 2^{-j} \cdot K^{-1}, \tag{13}$$

which corresponds to (3).

Next we find the bound of every iteration. In what follows, we call *null* the bits of value 0 if a number is positive or 1 if a number is negative and complement representation is used.

First we find a bound for $\sigma_i 2^{-i} y(i)$:

(a) Iteration $i = 0$. Because $x(0)$ and $y(0)$ are fractions, the integer part bits are all null. This justifies the three zeroes of the y expression in the first iteration in Fig. 1.

(b) Iteration $i = 1$. From Eqs. (1) we have $y(1) = y(0) - \sigma_0 x(0)$. From $|x(0)| < 1$ and $|y(0)| < 1$ we obtain that $|y(1)| < 2$. Therefore $|\sigma_1 2^{-1} y(1)| < 1$ and consequently, it is a fractional number. That is the reason why the integer bits in the expression $|\sigma_1 2^{-1} y(1)|$ in the Fig. 1 are all zeroes.

(c) Iterations $i \geq 2$. The value of the scale factor that the CORDIC algorithm introduces is $K = 1.6467$. Any value $y(i)$ is less than the module of the largest possible initial vector ($\sqrt{2} - 2^{-n}$) scaled by 1.6467. Therefore, we have the following bound for $y(i)$:

$$|y(i)| < K\sqrt{2} = 2.32\ldots \tag{14}$$

Multiplying both terms of this expression by 2^{-i} we can write that

$$|2^{-i} y(i)| < 2^{-i} 2.32 < 2^{-i+2}. \tag{15}$$

As a consequence of (15), in every iteration we add or subtract to each coordinate a bounded amount, and this bound is reduced by a factor of 2^{-1} from one iteration to the next. This justifies that the most significant non-null bit possible for $\sigma_i 2^{-i} y(i)$ is shifted one bit to the right in every iteration. For example, for $i = 2$ we have $|\sigma_2 2^{-1} x(2)| < 1$, and then, the bits of the integer part of the expression $|\sigma_2 2^{-1} x(2)|$ in Fig. 1 are all null; for $i = 3$ we have $|\sigma_2 2^{-1} x(2)| < 0.5$, and then, the 3 MSB are null, etc.

As consequence of (a) and (b) items, the value of bit x_1 can only be modified if a carry is propagated to it in the second iteration. Therefore, we can begin the application of the Eq. (3) just after we know $x(2)$ (that is, the bit x_0 and the carry c_0 of $x(2)$, see Fig. 1).

In Appendix A we justify from a mathematical point of view the procedure that we have just described, doing it independently from the kind of representation selected. In this appendix we prove the relationship between the value of the weighted MSBs of $x(i+1)$ up to the bit $x_j(i)$ ($j = i-2$) and the same bits of $x(i)$, which is the base of the previous procedure.

Based on this reasoning we give the CORDIC algorithm with parallel compensation of the scale factor.

Algorithm

```
For $i = 0$ to $i = n+1$ do
{
    ($i \leq n$)
    if (rotation mode) then $\sigma_i = \mathrm{sign}(z(i))$ else $\sigma_i = \mathrm{sign}(y(i))$;
    $x(i+1) = x(i) + \sigma_i 2^{-i} y(i)$
    $y(i+1) = y(i) - \sigma_i 2^{-i} x(i)$
    $z(i+1) = z(i) - \sigma_i \tan^{-1}(2^{-i})$

    (if $i \geq 2 \quad j = i-2$)
    $\delta_j = x_{-j}(i) + 2(c_{-j} - y_2*),\ \mu_j = y_{-j}(i) + 2(c'_{-j} - x_2*)$    (16)
    $xcomp(j) = xcomp(j-1) + \delta_j 2^{-j} K^{-1}$    (17)
    $ycomp(j) = ycomp(j-1) + \mu_j 2^{-j} K^{-1}$    (18)
}
```

where $x_{-j}(i), y_{-j}(i)$ are the bits of weight 2^{-j} of $x(i)$ and $y(i)$, c_{-j}, c'_{-j} are the carries at position 2^{-j} and y_2*, x_2* are zeroes if sign-and-magnitude or signed-digit number representation is selected or they correspond to the sign bits of the expressions $\sigma_i 2^{-i} y(i)$ and $\sigma_i 2^{-i} x(i)$, respectively, if complement representation is used. The initial value of $xcomp$ is 0 except if true-and-complement representation is used, in which case it is $-2x_2(1)$. Similary for *ycomp*.

2.1. Word-Serial Architecture

Figure 3 shows the word-serial architecture for the implementation of the CORDIC algorithm with parallel compensation of the scale factor. In what follows we will explain the operation of coordinate x; operation on coordinate y is similar. In this figure the data paths for the compensation of the scale factor are pointed out in dotted line and the hardware elements added to the conventional CORDIC are shadowed, whereas the multiplexors for the classical scaling iterations were eliminated.

In order to obtain coefficient δ_j it is necessary to reach bit $x_{-j}(i)$ and carry c_{-j} in each iteration (see expression (16)). To access these bits it would be interesting for them to stay in fixed positions in the corresponding registers. In order to do this, we carry out a *rotation* of one bit to the left of $x(i)$ in every iteration, in such a way that bit $x_{-j}(i)$ is always in the leftmost bit position of the register Rx, and the carry c_{-j} is always in the same position in the adder (see Fig. 3) Consequently, the particular position of the bits in the adder requires the outgoing carry to be connected to the incoming carry, as Fig. 3 shows, and therefore, a specific number representation must be used, as we will see in the Sections 2.1.2 and 2.1.3.

The shift to the right operation implicit in $\sigma_i 2^{-i} y(i)$ (see Eq. (26)) is carried out by means of a *rotation* of i bits to the right (module $\vec{i}$ *Rotator*) plus a sign-extension of i bits (module Z and the shift register attached). This split of functions allows the use of a single hardware to apply the sign extension to other data. Basically, the shift register is initialized to “1” in every bit and in every iteration a “0” is introduced from the right side. This shift register is used by module Z as a reference to know how many bits the sign extension

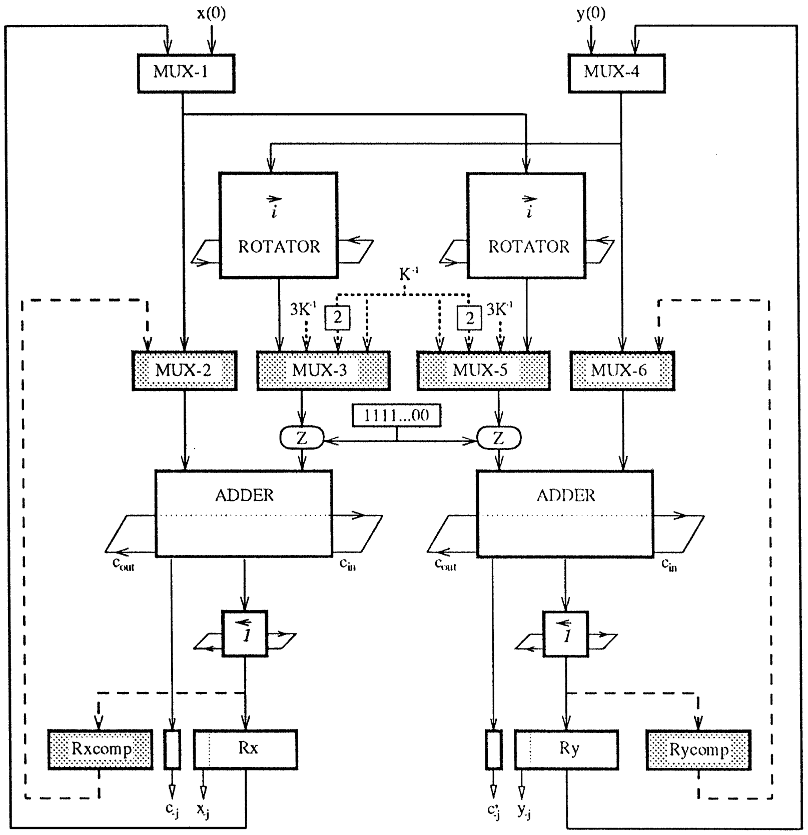


Figure 3. CORDIC architecture with parallel compensation.

has (the hardware of module Z will be analyzed in the next subsections).

The parallel compensation of the scale factor is supported by the registers *Rxcomp* and *Rycomp*. To perform the Eq. (17) we need the values $2^{-j}K^{-1}$, $2 \cdot 2^{-j}K^{-1}$ and $3 \cdot 2^{-j}K^{-1}$, because $|\delta_j| = \{0, 1, 2, 3\}$. Because, like coordinate x, the value of the variable $xcomp$ is rotated one bit to the left in every iteration we do not need to shift the initial values K^{-1}, $2K^{-1}$ and $3K^{-1}$ in every iteration, and it is sufficient to make the sign extension over the suitable bits. This last operation is carried out by means of the Z module and the shift register attached.

2.1.1. Timing. A CORDIC iteration is composed of two sequential suboperations: a shift (with associated delay "S") and an addition (with delay "A"). On the other hand, a scale factor compensation iteration is composed of one addition only (see Eq. (17); the shift operation over K^{-1} is not necessary as we have just analyzed in the previous paragraph). We take advantage of this fact and carry out the addition associated to the scale factor compensation while the shift operation of the classic CORDIC iteration is performed. Therefore, in a rough approximation, the delay "D" of an iteration is given by

$$D = \max\{S, A\} + A \tag{19}$$

Since the classical CORDIC delay is $D = S + A$, to keep the same delay in the architecture of Fig. 3 it is necessary that "S" and "A" be similar. This condition can be accomplished if redundant arithmetic or conventional arithmetic with fast adders (CLAs) is used.

Figure 4 shows the dependency graph and the timing of a couple of iterations for coordinate x (coordinate y is similar) when $S \approx A$. In the classic CORDIC approach, every iteration is performed in one cycle. We have split the classic CORDIC cycle into two subcycles: in the first one we carry out the shift suboperation, and in the second one we carry out the addition. We perform the first subcycle at the same time as the scale factor compensation cycle is carried out. Observe that in the first falling slope register *Rx* is loaded and Mux-3 selects the suitable input whereas in the second falling slope register *Rxcomp* is loaded and Mux-3 selects the leftmost input. This way, every one of the two subcycles which every CORDIC iteration is now composed has a time approximately half of the classic CORDIC cycle, and consequently, the time of one iteration is not modified with regard to the classic approach.

In Fig. 5 we can see the complete sequence of CORDIC iterations with parallel compensation of scale factor. As we can see in this figure, the total latency is $2n + 5$ cycles, which corresponds approximately to the time of $n + 2.5$ standard CORDIC iterations. In contrast, the number of iterations of the CORDIC algorithm without compensation of scale factor is $n + 1$, so that the overhead for compensation in the carry-analysis method is only of about 1.5 iterations.

If $A > S$ the CORDIC iteration delay is given by $D = 2A$. Nevertheless, it is possible to reduce the cycle time to $D = S + A$ if we use a separate parallel hardware to carry out the compensation of the

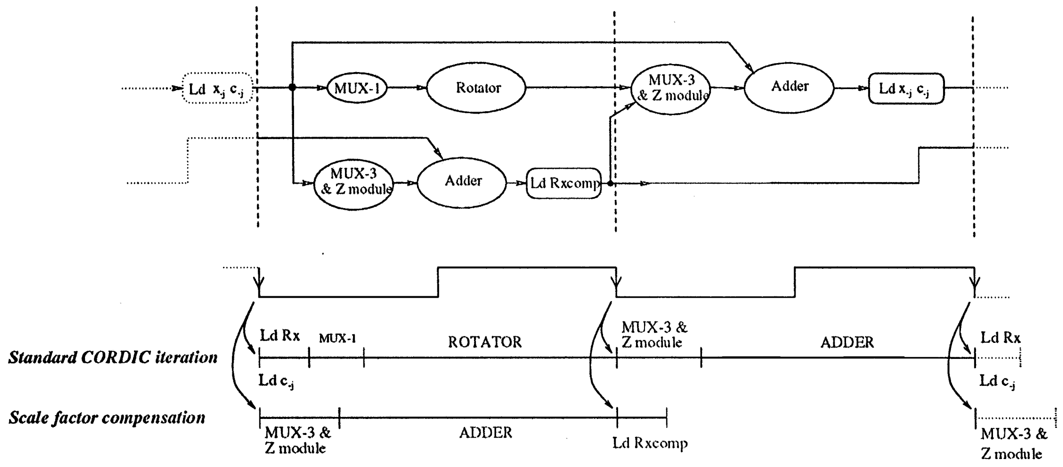


Figure 4. Dependency graph and timing.

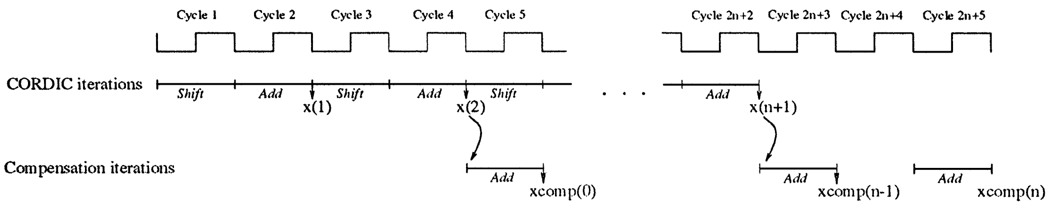


Figure 5. Global timing for the CORDIC with parallel compensation.

scale factor. In this case, the intermediate multiplexors MUX-2,3,5,6 in Fig. 3 are eliminated.

2.1.2. Implementation in Conventional Arithmetic. We now look for a number representation that is suitable for the architecture shown in Fig. 3. In this architecture we have connected the outgoing carry to the incoming carry in the adders, in such a way that the outgoing carry of the MSB and the incoming carry of the LSB are always connected. In two's complement representation an outgoing carry can corrupt the true value of the addition and therefore, it is not valid. On the other hand, in the addition in one's complement a nonzero outgoing carry requires a correction of the result obtained. This correction consists of an addition of the value 1 to the current result, which can be carried out connecting the outgoing carry to the incoming carry [10]. Therefore, a suitable number representation to perform the CORDIC algorithm with parallel compensation of the scale factor using conventional arithmetic is *one's complement.*

If this representation is chosen and the input data is in two's complement, it is not necessary to convert the negative two's complement data into negative one's complement data because the error is less than the precision we work with since we should subtract the value 1 at the LSB of the guard bit, which is out of precision. The same happens with the output data.

Next, we analyze the sign extension needed in the shift of coordinate y (see Eq. (26)). The module $\vec{i}$ *Rotator* is in charge of producing a rotation of i bits to the right of coordinate y. But, since coordinate y stays rotated i bits to the left in register RY (see second paragraph of the beginning of this section), coordinate y appears with the bits placed in the conventional order at the output of the rotator (that is, the MSB on the leftmost bit and the LSB on the rightmost bit). For example, assume $y(4) = \hat{0}01.1010111\underline{1}$ (we have pointed out the MSB with symbol ^ and the LSB with the symbol _). Then, register *RY* holds the value $010111\underline{1}\hat{0}01.1$, at the output of the rotator we have the value $\hat{0}01.1010111\underline{1}$, and the value of $2^{-3}y(3)$ is found after the Z module as $001101\underline{0}\hat{0}00.0$.

In Fig. 6 we can see the order of the bits at the output of the rotator module and the structure of the *Z* module. This module is composed of $n - 1$ 2-to-1 multiplexors

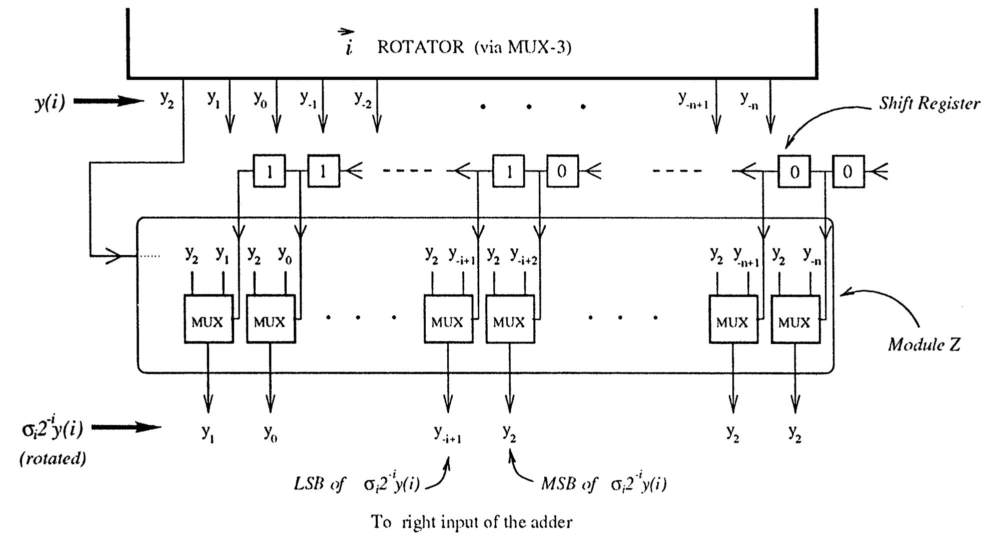

$\vec{i}$ ROTATOR (via MUX-3)
y(i)
y_2 y_1 y_0 y_{-1} y_{-2} y_{-n+1} y_{-n}
Shift Register
1 1 1 0 0 0
y_2 y_1 y_2 y_0 y_2 y_{-i+1} y_2 y_{-i+2} y_2 y_{-n+1} y_2 y_{-n}
MUX
Module Z
$\sigma_i 2^{-i} y(i)$
(rotated)
y_1 y_0 y_{-i+1} y_2 y_2 y_2
LSB of $\sigma_i 2^{-i} y(i)$
MSB of $\sigma_i 2^{-i} y(i)$
To right input of the adder

Figure 6. Sign extension in one's complement.

controlled by the shift register, that allows to place the sign bit y_2 over the suitable bits.

2.1.3. Implementation in Signed-Digit Arithmetic. We have several possibilities to choose a redundant number representation. Since the architecture that we propose has the outgoing carry of the adder connected to the incoming carry, we have to avoid the propagation of a nonzero carry of the MSB of $x(i)$ to the LSB. If we choose carry-save representation we must use one's complement to avoid the problem of the outgoing carry, as explained in the previous subsection.

In signed-digit representation there are always zeroes over the integer part of $\sigma_i 2^{-i} y(i)$ (see Appendix B), and therefore, an outgoing carry will never occur. Nevertheless, the expansion of a number represented in signed-digit forces us to use an additional bit to avoid an outgoing carry. We prove this in Appendix C. In this case, the sign extension needed in the shifts of coordinate y (see Eq. (26)) is simpler than the one's complement case, because we only have to force to zero instead of to the sign. Therefore, the architecture of module Z is similar to Fig. 6 but replacing the 2-to-1 multiplexors by two input AND gates. Consequently, the signed-digit representation is better than the carry-save one.

Since the same adder is used in the CORDIC iteration and in the compensation, the result of the compensation multiplication is obtained in signed-digit representation. A standard conversion to conventional representation would require a carry-propagate addition. However, this compensation can be viewed as a "distributed" left-to-right multiplication, so that the on-the-fly conversion [11] can be used.

2.2. Pipelined Architecture

It is possible to perform the *carry-analysis method* in a pipelined architecture. In this case, the rotator modules of Fig. 3 are implemented in a hardwired fashion and a parallel data path to carry out the parallel compensation of the scale factor is needed. In this case, the pipeline is composed by $n+1$ stages to perform the CORDIC iterations plus $n+1$ parallel stages to carry out the compensation of the scale factor, with a delay of two cycles between the starting of both pipelines. Therefore, the throughput is one full CORDIC operation per cycle (the same that the standard CORDIC, with the same cycle time) and the latency is $n+3$ cycles. The increase in latency due to the compensation is only of two cycles, which suposses a minimal latency solution for a scale factor compensated CORDIC.

2.3. Computation of the Modulus of the Vector

If we are only interested in the computation of the modulus of the vector, we have to select the vectoring mode, and carry out $(n/2)+1$ iterations. Nevertheless, the parallel scale factor compensation proposed needs n iterations. We can reduce the number of these iterations if we modify slightly the algorithm of parallel compensation. To do this, we analyze 2 bits per iteration instead of 1 bit. This is due to the fact that the vectoring mode has the following bound:

$$|2^{-i} y(i)| < 2^{-i} 2.32 < 2^{-2i+3} \tag{20}$$

Therefore, the amount that is added or subtracted in every iteration is bounded, and this bound is reduced by a factor of 2^{-2} instead of 2^{-1} as we saw in expression (15). Consequently, after iteration $i=(n+1)/2$, coordinate x keeps the scaled module of the vector, and it is not necessary to perform more iterations. Then, applying the *carry analysis method* it is possible to obtain the module of the vector in only $\frac{n+1}{2}+2$ iterations. We only need to slightly modify the hardware because now expression (8) becomes $\delta_j = 4c_{-j} + 2x_{-j}(i) + x_{-j-1}(i)$ and therefore $|\delta_j|$ can take the new values 4 and 5. Consequently, in Fig. 3 six-input multiplexors instead of four-input multiplexors are needed.

3. Double Rotation Method

This method is valid only in rotation mode, and it can be applied in mixed radix 2-4 CORDIC systems and radix-4 CORDIC. First, we give a brief explanation of the double rotation method that can be found in [9]. After this, we design the new architecture that implements this method.

Let (x, y) be the coordinates of a vector to which we apply a rotation of θ. The coordinates obtained after $n+1$ CORDIC iterations can be expressed as follows:

$$\begin{bmatrix} x(n+1) \\ y(n+1) \end{bmatrix} = K \cdot \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \cdot \begin{bmatrix} x(0) \\ y(0) \end{bmatrix} \tag{21}$$

Let us now define a new angle β:

$$\beta = \cos^{-1}(K^{-1}) \tag{22}$$

We will now call $x^{+}(n+1)$, $y^{+}(n+1)$ and $x^{-}(n+1)$, $y^{-}(n+1)$ the coordinates obtained after $n+1$ CORDIC iterations when we apply a rotation of $(\theta+\beta)$ and $(\theta-\beta)$, respectively.

$$\begin{bmatrix} x^{+}(n+1) \\ y^{+}(n+1) \end{bmatrix} = K \cdot \begin{bmatrix} \cos(\theta+\beta) & \sin(\theta+\beta) \\ -\sin(\theta+\beta) & \cos(\theta+\beta) \end{bmatrix} \cdot \begin{bmatrix} x(0) \\ y(0) \end{bmatrix} \tag{23}$$

$$\begin{bmatrix} x^{-}(n+1) \\ y^{-}(n+1) \end{bmatrix} = K \cdot \begin{bmatrix} \cos(\theta-\beta) & \sin(\theta-\beta) \\ -\sin(\theta-\beta) & \cos(\theta-\beta) \end{bmatrix} \cdot \begin{bmatrix} x(0) \\ y(0) \end{bmatrix} \tag{24}$$

If we carry out the semi-sum of $x^{+}(n+1)$ with $x^{-}(n+1)$ and the semi-sum of $y^{+}(n+1)$ with $y^{-}(n+1)$ and take into account $\cos\beta = K^{-1}$, we have

$$\frac{x^{+}(n+1)+x^{-}(n+1)}{2} = x\cdot\cos\theta + y\cdot\sin\theta$$

$$\frac{y^{+}(n+1)+y^{-}(n+1)}{2} = -x\cdot\sin\theta + y\cdot\cos\theta$$

The second terms of these expressions are the values of the compensated coordinates $x(n+1)$ and $y(n+1)$, that is $x(n+1)K^{-1}$ and $y(n+1)K^{-1}$.

Summarizing, this method has three steps:

1. obtain $(\theta+\beta)$ and $(\theta-\beta)$,
2. rotate the vector $(x(0), y(0))$ an angle $(\theta+\beta)$ and $(\theta-\beta)$,
3. perform the final semi-sums.

3.1. Architecture

To carry out the second step of this method two CORDIC rotators working in parallel were used in [9]. That solution is valid for a pipeline architecture as well as a word-serial architecture. In this section we present a new word-serial architecture that avoids the hardware resource doubling of the aforementioned previous work.

A CORDIC iteration is composed by two sequential operations: one shift plus one addition which are performed in one cycle in the classic word-serial solution. In our design, we have pipelined the CORDIC iteration into two stages: the shift stage and the addition stage, and therefore, every CORDIC iteration is now composed by two subcycles instead of one cycle. This *word-serial two stages pipelined architecture* is presented in Fig. 7.

Based on the architecture of Fig. 7 it is possible that in every subcycle the first stage is performing the shift of the coordinates related to one of the rotations (i.e., $2^{-i}y^{-}(i)$), whereas the second stage is performing the addition related to the other rotation $(x^{+}(i) + \sigma_i^{+}2^{-i}y^{+}(i))$. In the next cycle, the roles are interchanged. In this way, we avoid the use of two CORDIC rotators, and use only one CORDIC rotator (modified as Fig. 7 shows) with the same functionality.

If we call S, A and R to the delay of the Shifter, Adder and the loading of a Register respectively, the delay of a CORDIC iteration in a classic word-serial architecture is $S+A+R$, whereas the delay of a CORDIC iteration for the architecture of Fig. 7 corresponds to the delay of two consecutive subcycles: $2\cdot(\max\{S, A\}+R)$. In order to obtain an efficient two stages pipelined design it is necessary that the delay of each stage is similar. This can be achieved in implementations using redundant arithmetic or conventional arithmetic with fast adders (i.e., CLA) since the delay of the shifter can be similar to the delay of the Adder ($S \approx A$). Thus, there is a good balance between both stages, and considering that $R \ll S$, the subcycle time ($\approx S + R$) can be slightly longer than half the classic CORDIC cycle time ($\approx 2S + R$).

The sequence of operations can be seen in Table 1. We show the contents of the registers of Fig. 7 and the value of the coefficients σ_i at the end of each cycle. The total number of subcycles needed to obtain the final compensated coordinates is $2n + 4$, which corresponds approximately with the equivalent time to $n + 2$ cycles of the classic approach if we take into account that the time of a classic CORDIC cycle corresponds approximately with the time of two subcycles. This results in an overhead time for the compensation of the scale factor corresponding to only one Classic CORDIC cycle time (overhead in the classic CORDIC: about $n/3$).

Table 1. Sequence of the double rotation method.

Reg.	c0	c1	c2	c3	c4		$2n+1$	$2n+2$	$2n+3$
Rα	β	α_0	α_0	α_1	α_1	...	α_n	α_n	α_n
Rz1	$z^+(0)$	$z^-(0)$	$z^+(1)$	$z^-(1)$	$z^+(2)$	...	$z^-(n)$	$z^+(n+1)$	$z^-(n+1)$
Rz2	—	$z^+(0)$	$z^-(0)$	$z^+(1)$	$z^-(1)$	...	$z^+(n)$	$z^-(n)$	$z^+(n+1)$
σ_i	σ_0^+	σ_0^-	σ_1^+	σ_1^-	σ_2^+	...	σ_n^-	—	—
Rx$'$	$x(0)$	$x(0)$	$x^+(1)$	$x^-(1)$	$x^+(2)$	...	$x^-(n)$	$x^+(n+1)$	—
Rx$''$	$y(0)$	$y(0)2^0$	$y^+(1)2^{-1}$	$y^-(1)2^{-1}$	$y^+(2)2^{-2}$	...	$y^-(n)2^{-n}$	—	—
Rx	—	$x^+(1)$	$x^-(1)$	$x^+(2)$	$x^-(2)$	...	$x^+(n+1)$	$x^-(n+1)$	$2x(n+1)K^{-1}$
Ry$'$	$y(0)$	$y(0)$	$y^+(1)$	$y^-(1)$	$y^+(2)$	...	$y^-(n)$	$y^+(n+1)$	—
Ry$''$	$x(0)$	$x(0)2^0$	$x^+(1)2^{-1}$	$x^-(1)2^{-1}$	$x^+(2)2^{-2}$	...	$x^-(n)2^{-n}$	—	—
Ry	—	$y^+(1)$	$y^-(1)$	$y^+(2)$	$y^-(2)$	...	$y^+(n+1)$	$y^-(n+1)$	$2y(n+1)K^{-1}$

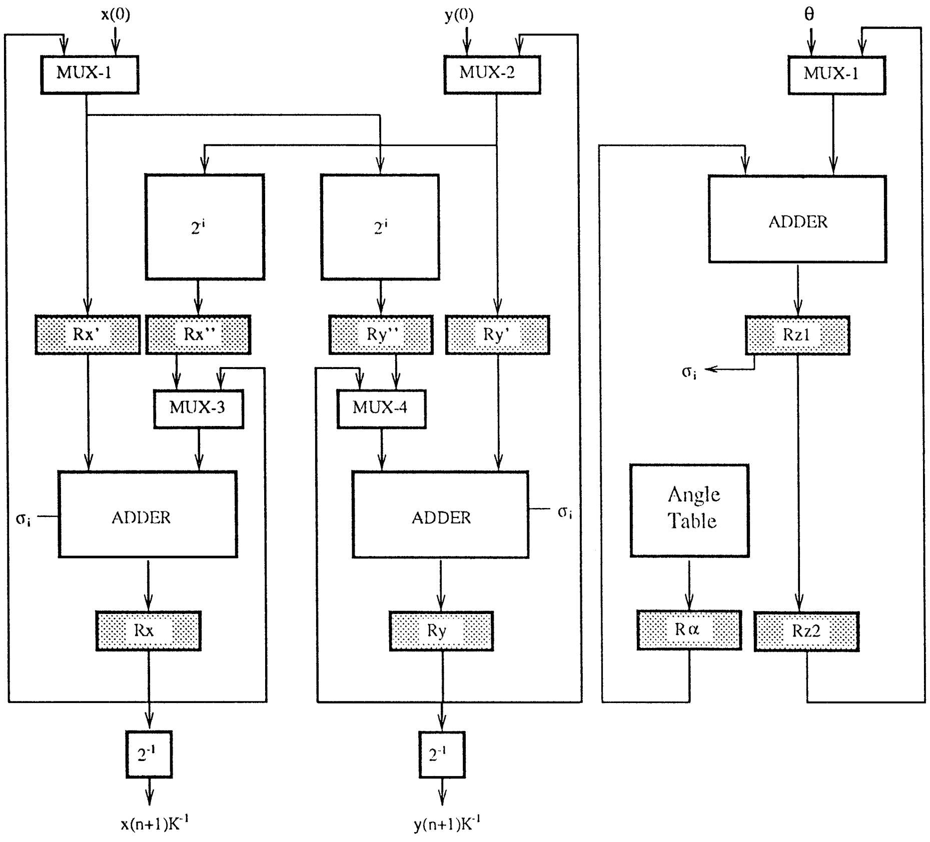


Figure 7. Architecture of the double rotation method.

The *double rotation method* can be applied directly in mixed radix 2-4 CORDIC systems [12]. In this kind of CORDIC systems the first $(n+1)/2$ iterations are carried out in radix-2 whereas the remaining iterations are performed in radix-4, only needing $n/4$ radix-4 iterations instead of $n/2$ radix-2 iterations of the classic approach, but the number of scaling iterations required for the compensation of the scale factor continues being the same. In this case, the delay associated to the scale factor compensation has a relatively greater weight than in a full radix-2 CORDIC (about 45% overhead in mixed radix and 30% overhead in radix-2), and therefore the alternative of using the double rotation method proposed in this paper seems specially interesting.

3.2. *Rotations by a Set of Known Angles*

Parallel compensation of the scale factor can be specially interesting when radix-4 CORDIC is used in systems where we want to perform rotations of a set of angles that are known beforehand. In radix-4 CORDIC the elementary angles are $\tan^{-1} 4^{-i}$ and the coefficient $\sigma_i = \{0, \pm 1, \pm 2\}$, which reduces to half the number of microrotations [13]. The scale factor depends on the sequence of σ_i's in which each angle is decomposed (see Eq. (2)). Unlike in the radix-2 CORDIC, in radix-4 this sequence changes from one angle to another, and consequently the scale factor is not constant.

However, in radix-4 the angle decomposition is redundant (different sequences of σ_i''s correspond to the same angle), and thus we can select the decompositions in such a way that the number of scale factors is minimized [14]. To apply the double rotation method, we can keep the different angles β (see expression (22)) in a table and select the suitable one when we perform the corresponding rotation. This system results in a low latency: about $\frac{n+1}{2} + 2$ iterations on average, including the scale factor compensation (in contrast, the standard radix-2 CORDIC requires about $4n/3$ iterations and the standard radix-4 CORDIC about $n + 1$ iterations).

4. Comparison

In this paper we propose two algorithms and the corresponding architectures to carry out the parallel compensation of the scale factor. In systems where only rotation mode is needed, the best choice is the *double-rotation method* since we can apply mixed radix resulting in a significant reduction of the total number of iterations. If vectoring mode or both vectoring and rotation modes are required, we should use the *carry-analysis method*.

The word-serial architecture for the double rotation method proposed in this paper reduces by about half the hardware cost of the solution proposed in [9]. The version of the carry-analysis method (called the bit-analysis method) that can be found in [9] needs a couple of separate adders to carry out the compensation of the scale factor, and the coefficient δ_i depends on six bits, whereas in the algorithm proposed in this paper the coefficients depend on only three bits and no extra adders are needed to compensate the scale factor. Furthermore, the bit-analysis method is only valid for conventional arithmetic, whereas the carry-analysis method is valid both in redundant and conventional arithmetic.

The number of cycles to obtain a rotation by means of the CORDIC algorithm including the compensation of the scale factor is about $n + \frac{1}{2}n$ if we use the technique of adding scaling iterations. Nevertheless, the repetition of some iterations may modify the value of the scale factor in such a way that the total number of iterations may become $n + \frac{1}{4}n$ in the best case [15]. In any case, the scale factor compensation overhead increases linearly with the word length, n. With the algorithms and architectures proposed in this paper the scale factor overhead is fixed to a couple of iterations, and it is not dependent on n.

The carry-analysis method is related to the on-line CORDIC algorithm presented in [16], where $n + 6$ steps are needed including the compensation of the nonconstant scale factor. In contrast, the carry-analysis method uses the standard CORDIC algorithm and has a constant scale factor. Its relation to the on-line algorithm is that it can be considered as an application of the digitalization to the traditional CORDIC, with a left-to-right multiplier for the compensation. As mentioned before, when redundant addition is used in this multiplier, an on-the-fly conversion can be used to avoid the conventional addition in the conversion.

In [5] possibilities for the minimization of the scale factor compensation overhead are studied. The best strategies proposed are different for pipelined and word-serial implementation. Two strategies are proposed for the pipelined implementation. The first consists in the canonic signed-digit encoding of the scale factor with a minimum number of 1's in such a way that an additive decomposition of K^{-1} is obtained. If p is the number of nonzero bits of K^{-1}, then the multiplication of K^{-1} with $x(n + 1)$ can be carried out by means of a tree of $\log_2 p$ levels. The second one is based on a multiplicative representation of K^{-1}, which produces a more area economical solution but more latency. In any case, the latency of the resulting pipeline is greater than the latency of the pipeline architecture proposed in this paper (see Section 2.2). While the overhead for scaling factor compensation is of $\log_2 p$ stages with the additive strategy proposed in [5], the overhead is fixed to one stage for any value of n in the architecture proposed in this paper.

The conclusion obtained in [5] for word-serial implementation is that the design of [8] is the lowest latency solution. This work is based on a previous one [7] where an iteration combines a standard CORDIC iteration and a scaling iteration. The resulting system is more complex, with four additions in each coordinate. To simplify the hardware, the least significant-term is eliminated, but this yields restrictions in the convergence that are solved by the introduction of the repetition of some iterations. Therefore, the resulting architecture uses three-input adders and incorporates some additional multiplexors, as compared to the standard CORDIC architecture. In contrast, we use a couple of additional registers, two-input adders and four-input multiplexors. Therefore, there is not a clear gain in area or iteration delay in either architecture. Nevertheless, the total computation time is smaller in our architecture since repetitions are needed in [8] to maintain convergence whereas our design does not need any repetition. Furthermore, a specific design for calculating the modulus of the vector based on [8] requires several scaling iterations after iteration $i = (n+1)/2$, whereas a design based on the algorithm we propose requires only two iterations.

Appendix

Appendix A: Relationship Between the Value of the Weighted MSBs of x(i + 1) and x(i) up to the Bit $x_j(i)$ ($j = i - 2$)

Theorem 1. *Let E and F the functions integer part and fractional part*[1] *respectively*; *along the CORDIC iterations the following equalities are kept*:

$$\begin{aligned} &E[x(i+1)2^j] \\ &\quad = E[x(i)2^j] + E[F[x(i)2^{i-2}] + \sigma_i 2^{-2} y(i)] \\ &E[y(i+1)2^j] \\ &\quad = E[y(i)2^j] + E[F[y(i)2^{i-2}] + \sigma_i 2^{-2} x(i)] \end{aligned} \tag{25}$$

where $j = i - 2$

Proof: We prove it for $x(i)$, being similar for $y(i)$. The CORDIC equation for the coordinate x is

$$x(i+1) = x(i) + \sigma_i \cdot 2^{-i} \cdot y(i). \tag{26}$$

Multiplying both terms of this expression by 2^{i-2} and applying the integer part function we can write

$$\begin{aligned} E[x(i+1)2^{i-2}] &= E[x(i)2^{i-2} + \sigma_i 2^{-2} y(i)] \\ &= E[x(i)2^{i-2}] + E[\sigma_i 2^{-2} y(i)] \\ &\quad + E[F[x(i)2^{i-2}] + F[\sigma_i 2^{-2} y(i)]] \end{aligned} \tag{27}$$

Moreover, multiplying expression (15) by 2^{i-2} we have $|\sigma_i 2^{-2} y(i)| < 1$. We consider two cases:

(a) $\sigma_i 2^{-2} y(i) \geq 0$. In this case $E[\sigma_i 2^{-2} y(i)] = 0$ and $F[\sigma_i 2^{-2} y(i)] = \sigma_i 2^{-2} y(i)$. Therefore, the expression (27) is

$$\begin{aligned} &E[x(i+1)2^{i-2}] \\ &\quad = E[x(i)2^{i-2}] + E[F[x(i)2^{i-2}] + \sigma_i 2^{-2} y(i)] \end{aligned} \tag{28}$$

(b) $\sigma_i 2^{-2} y(i) < 0$. In this case $E[\sigma_i 2^{-2} y(i)] = -1$ and $F[\sigma_i 2^{-2} y(i)] = \sigma_i 2^{-2} y(i) + 1$. Therefore,

$$\begin{aligned} &E[x(i+1)2^{i-2}] \\ &\quad = E[x(i)2^{i-2}] - 1 \\ &\qquad + E[F[x(i)2^{i-2}] + \sigma_i 2^{-2} y(i) + 1] \end{aligned} \tag{29}$$

Since $E[a+1] = E[a] + 1$ in the expression (29), we obtain, for any case of $\sigma_i 2^{-2} y(i)$

$$\begin{aligned} &E[x(i+1)2^{i-2}] \\ &\quad = E[x(i)2^{i-2}] + E[F[x(i)2^{i-2}] + \sigma_i 2^{-2} y(i)] \end{aligned} \tag{30}$$

□

Note that the last two terms of this expression represent the amount that we add or subtract to obtain the MSBs of $x(i+1)$ from the MSBs of $x(i)$, and it is the base of the algorithm of parallel compensation of the scale factor.

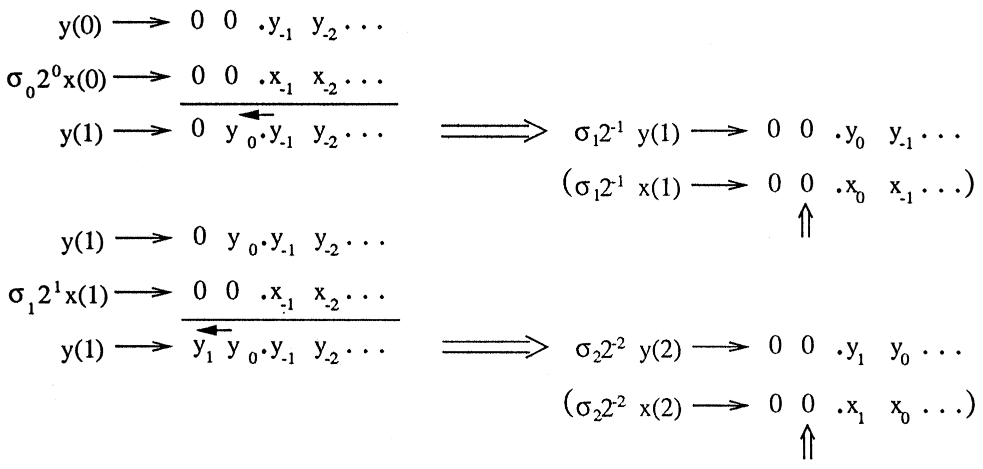

y(0) → 0 0 .y-1 y-2 . . .
σ0 2^0 x(0) → 0 0 .x-1 x-2 . . .
y(1) → 0 y0 .y-1 y-2 . . .
σ1 2^-1 y(1) → 0 0 .y0 y-1 . . .
(σ1 2^-1 x(1) → 0 0 .x0 x-1 . . .)
y(1) → 0 y0 .y-1 y-2 . . .
σ1 2^1 x(1) → 0 0 .x-1 x-2 . . .
y(1) → y1 y0 .y-1 y-2 . . .
σ2 2^-2 y(2) → 0 0 .y1 y0 . . .
(σ2 2^-2 x(2) → 0 0 .x1 x0 . . .)

Figure 8. First two CORDIC iterations.

Appendix B: Integer Part Bits of $2^{-i}y(i)$ are All 0 in Signed-Digit

Because $|x(0)| < 1$ and $|y(0)| < 1$, the trivial representation in signed-digit has the digit 0 in the integer part. In Fig. 8 we can see the operation to obtain $y(1)$. According to the modified rules for adding binary SD number [10], the only possibility for the sum bit in the position of weight 2^1 is 0. Nevertheless, the position of weight 2^0 can be -1, 0 or 1 because a carry can be propagated from the bits of weight 2^{-1}. Expression $2^{-1}y(1)$ means that we have to shift the value of $y(1)$ one bit to the right. Therefore, after this shift operation, the bit of weight 2^0 of $2^{-1}y(1)$ is 0, as we can see in Fig. 8.

The same situation takes place for $y(2)$, as shown in Fig. 8. We can see that in every iteration the size of the word can increase by one bit and a right shift is carried out. Therefore, the integer bits are always zeroes.

Appendix C: Expansion of a Signed-Digit Number

Due to the bound of the coordinates $x(i)$ and $y(i)$, a minimum of two bits are needed to represent the integer part (see expression (14)). Nevertheless, a number in signed-digit representation can expand one bit to the left after an addition. Therefore, we use *three* bits for the integer part, and we are going to prove that coordinates x and y do not expand more than one bit.

Because the integer bits of $2^{-i}y(i)$ are all zeroes (see Appendix B), coordinate x is obtained in every iteration after an addition/subtraction operation in the following way:

$$\begin{array}{r} x_1\ x_0\ x_0\ .\ x_{-1}\ x_{-2}\ \ldots \\ \pm\ \underline{0\ \ 0\ \ 0\ .\ y_{-1}\ y_{-2}\ \ldots} \end{array}$$

According to the modified rules for adding binary SD numbers [10], the only possibilities to obtain a carry in the MSB are:

$$\text{(a)}\quad \begin{array}{l} 1\ 1\ x_0\ .\ x_{-1}\ x_{-2}\ \ldots \\ \underline{0\ 0\ 0\ \ .\ y_{-1}\ y_{-2}\ \ldots} \end{array}$$

$$\text{(b)}\quad \begin{array}{l} 1\ 0\ x_0\ .\ x_{-1}\ x_{-2}\ \ldots \\ \underline{0\ 0\ 0\ \ .\ y_{-1}\ y_{-2}\ \ldots} \end{array}$$

$$\text{(c)}\quad \begin{array}{l} \bar{1}\ \bar{1}\ x_0\ .\ x_{-1}\ x_{-2}\ \ldots \\ \underline{0\ 0\ 0\ \ .\ y_{-1}\ y_{-2}\ \ldots} \end{array}$$

We analyze each case in the following:

(a) Due to the bound of $x(i)$, the value $11x_0.x_{-1}x_{-2}\ldots$ is not possible. We analyze the worst case: $11\bar{1}.\bar{1}\ldots$. The decimal value of this expression is $11\bar{1}.\bar{1}\ldots = 4+2-1-2^{-1}-\cdots > 4 > |x(i)|_{\max}$, which is an impossible situation.

(b) The bound in signed-digit for this value is $2 < 10x_0.x_{-1}x_{-2}\ldots < |x(i)|_{\max} = 2.32\ldots$. Taking into account the weight of each bit, the only possibility to fulfill this bound is $x_0 = \bar{1}, x_{-1} = \bar{1}$. There are two possibilities to obtain a number like $10\bar{1}.\bar{1}\ldots$ in a CORDIC algorithm, according to the modified rules for adding binary SD numbers [10]

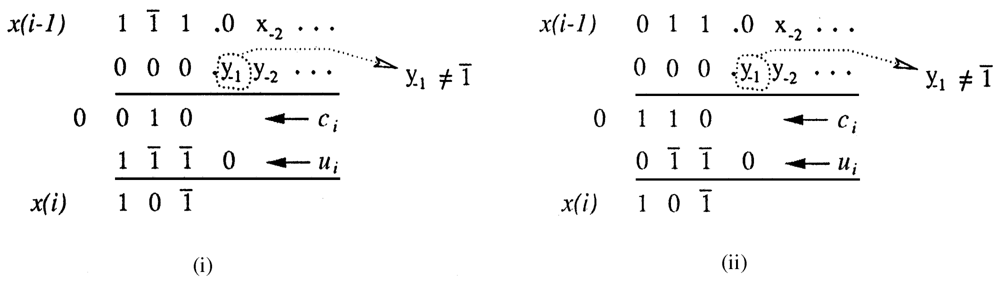


Figure 9. Ways to obtain $10\bar{1}.\bar{1}\ldots$.

and we present them in Fig. 9:

(i) In this case the value $x(i-1) = 1\bar{1}1.0x_{-2}\ldots > |x(i)|_{\max}$ for all $x_{-2}x_{-3}\ldots$, and therefore, it is an impossible value.

(ii) As in the previous case, the initial value $x(i) = 011.0x_{-2}\ldots > |x(i)|_{\max}$ is an impossible value.

Therefore, the value $10x_0.x_{-1}x_{-2}\ldots$ is impossible to obtain.

(c) This case is similar to case (a)

Note

1. $E[a] \in Z : E[a] \leq a < E[a] + 1$; $F[a] = a - E[a]$.

References

1. J.E. Volder, “The CORDIC trigonometric computing technique,” *IRE Trans. Elect. Comput.*, EC, No. 8, pp. 330–334, 1959.
2. J.S. Walther, “A unified algorithm for elementary functions,” *Proc. Spring. Joint Comput. Conf.*, pp. 379–385, 1971.
3. Y.H. Hu, “CORDIC-based VLSI architectures for digital signal processing,” *IEEE Signal Processing Magazine*, No. 7, pp. 16–35, July 1992.
4. G.L. Haviland and A.A. Tuszynski, “A CORDIC arithmetic processor chip,” *IEEE Trans. Comput.*, Vol. C-29, No. 2, pp. 68–79, Feb. 1980.
5. D. Timmermann, H. Hahn, B.J. Hosticka, and B. Rix, “A new addition scheme and fast scaling factor compensation mehods for cordic algorithms,” *The VLSI Journal of Integration*, No. 11, pp. 85–100, 1991.
6. H.M. Amhed, “Signal processing algorithms and architectures,” Ph.D. dissertation, Standford University, June 1982.
7. A.M. Despain, “Fourier transform computers using cordic iterations,” *IEEE Trans. Comput.*, Vol. C-23, No. 10, pp. 993–1001, Oct. 1974.
8. R. Meyer and R. Mehling, “Architecture and performance of a new arithmetic unit for the computation of elementary functions,” *Proc. ICASSP’90*, pp. 1783–1786, 1990.
9. J. Villalba, J.A. Hidalgo, E. Antelo, J.D. Bruguera, and E.L. Zapata, “CORDIC architecture with parallel compensation of the scale factor,” *Proc. Int. Conf. on Application Specific Array Processors (ASAP’95)*, pp. 258–269, July 1995.
10. I. Koren, *Computer Arithmetic Algorithm*, Prentice Hall, p. 25, 1993.
11. M. Ercegovac and T. Lang, “On-the-fly conversion of redundant into conventional representations,” *IEEE Trans. Comput.*, Vol. C-36, No. 7, pp. 895–897, 1987.
12. J. Lee and T. Lang, “Constant-factor redundant cordic for angle calculation and rotation,” *IEEE Trans. Compt.*, Vol. 41, No. 8, pp. 1016–1025, Aug. 1992.
13. E. Antelo, J. Villalba, J.D. Bruguera, and E.L. Zapata, “High performance rotation architectures based on radix-4 cordic algorithm, *IEEE Trans. Comput.*, Vol. 46, No. 8, pp. 855–870, 1996.
14. J.D. Bruguera, N. Guil, T. Lang, J. Villalba, and E.L. Zapata, “CORDIC based parallel/pipelined architecture for the Hough transform,” *J. of VLSI of Signal Processing*, Vol. 12, No. 3, 1996.
15. J.M. Delosme, “VLSI implementation of rotations in pseudo-euclidean spaces,” *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, Vol. 2, pp. 927–930, 1983.
16. H.X. Lin and H.J. Sips, “On line cordic algorithm,” *IEEE Trans. Comput.*, Vol. 39, pp. 1038–1052, Aug. 1990.

Julio Villalba was born in Granada (Spain) in 1962. He received the B.S. degree in Physics in 1986 from the University of Granada and the Ph.D. degree in Computer Engineering in 1995 from the University of Malaga, Spain. During 1986–1991 he worked in the R&D Department of Fujitsu Spain and was Assistant Professor at the University of Malaga. From 1992 he is an Associate Professor at the Department of Computer Architecture. Villalba’s primary research interest is in computer arithmetic, variable precision, VLSI design and high performance architectures.
julio@ac.uma.es

Tomás Lang is a Professor in the Department of Electrical and Computer Engineering at the University of California, Irvine. Previously he was a Professor in the Computer Architecture Department of the Polytechnic University of Catalonia, Spain, and a faculty member of the Computer Science Department at the University of California, Los Angeles. He received an Electrical Engineering degree from the Universidad de Chile in 1965, a M.S. from the University of California (Berkeley) in 1966 and the Ph.D. from Stanford University in 1974. Lang's primary research and teaching interests are in digital design and computer architecture with current emphasis on high-speed and low-power numerical processors and multiprocessors. He is coauthor of two textbooks on digital systems, two research monographs, one IEEE Tutorial, and author or coauthor of research contributions to scholarly publications and technical conferences.

Emilio L. Zapata received the B.S. degree in Physics from the University of Granada in 1978 and the Ph.D. degree in Physics from the University of Santiago de Compostela in 1983. During 1978–1981, he was an Assistant Professor in the University of Granada, and during 1982–1991, he successively was Assistant, Associate, and Full Professor in the University of Santiago de Compostela. Currently he is a Professor with the Department of Computer Architecture at University of Malaga. His research interests are in the area of parallel computer architecture, parallel algorithms, numerical algorithms for dense and sparse matrices and VLSI digital signal processing. In these areas, he has published more than 70 papers in refereed International Journal and about 100 refereed International Conference Proceeding.