Interval Sine and Cosine Functions Computation Based on Variable–Precision CORDIC Algorithm

J. Hormigo, J. Villalba and E.L. Zapata
Dept. Computer Architecture, University of Malaga
Complejo tecnologico, campus Teatinos, P.O.Box 4114
E-29080 Malaga (SPAIN)
julio@ac.uma.es

Abstract

In this paper we design a CORDIC architecture for variable–precision, and a new algorithm is proposed to perform the interval sine and cosine functions. This system allows us to specify the precision to perform the sine and cosine functions, and control the accuracy of the result, in such a way that recomputation of inaccurate results can be carried out with higher precision. An important reduction in the number of iterations is obtained by taking advantage of the differential angle, and the number of cycles per iteration is reduced by avoiding the additions of the leading all zero *words. As a consequence, the computation time of the interval function evaluation obtained is close to that of a point function evaluation. The problem of the large table of angles and the scale factor compensation involved in a high precision CORDIC has been solved efficiently.*

1. Introduction

In real scientific computation there are a lot of problems which require numerically intensive calculation. In this type of calculation, roundoff errors and catastrophic cancellation may lead to totally inaccurate results [17][9]. For this reason, a large number of applications, software packages and software libraries have been presented to control this kind of error during the last two decades [2][18][8][6]. These techniques are based on two related ideas: variable–precision and interval arithmetic [20].

The variable–precision technique reduces these errors by increasing the number of data bits and allowing the programmer to choose the required precision. The interval arithmetic technique works over an interval of values instead of fixed values and it ensures that the exact solution is within an interval. This way, if we know the size of the interval we know the precision. Merging both techniques permits working with a selected data size and re–calculate the results with a greater size if the accuracy was not sufficient [24][7].

Software solutions are quite slow, since all variable–precision interval arithmetic is simulated [24][23]. Specifically, this is the case if very high precision (about 1000 bits) and all kinds of operations are required, as in [7]. Because of this, there are some hardware designs that attempt to overcome the large overhead of the software solutions [21][5][3][4][24]. Nevertheless, the problem of variable–precision interval elementary function evaluation has not been confronted directly. In [24], a variation of the polynomial approximation given in [1] are proposed to perform elementary function evaluation using additions and multiplications, but this involve a large amount of operations.

The more usual technique for computing elementary function in hardware is a combined method [26][16] which uses table lookup to reduce the number of multiplications involved in the polynomial approximation. In spite of this reductiond, this number holds very high for the precisions we want to work with. Digit–by–digit methods are based on simple iterative equations, involving only shift and add operations. In this method, when the precision increases, it produces a linear increase in the number of iterations. We propose a digit–by–digit based algorithm to evaluate variable–precision elementary functions and specifically, the CORDIC algorithm to evaluate trigonometric functions.

On the other hand, the published works that deal with interval elementary functions are focused from a software point of view, needing at least the time of two point function evaluation to perform the interval function [22]. In this paper we develop a CORDIC based algorithm (and its hardware support) for the sine and cosine functions that needs slightly more time than one point evaluation for most cases.

The paper is structured as follows: in section 2 we present the basic CORDIC algorithm including the way to obtain the sine and cosine functions. In section 3 we pro-

pose the equations of the CORDIC algorithm for variable–precision. Next, in section 4 we propose an algorithm to perform the interval sine and cosine functions in a time near to that of a point function. In section 5 a detailed study of the error is carried out and some modifications of the algorithm are considered to ensure correct rounding. The architecture to support the variable–precision CORDIC and the interval sine and cosine is presented in section 6. In section 7 we show how to reduce the total number of cycles. Finally in section 8 we provide some conclusions to this work.

2. The CORDIC algorithm

The CORDIC algorithm (COordinate Rotation DIgital Computer) was introduced to compute trigonometric functions and generalized to compute linear and hyperbolic functions [31][32] (for details, see [14]). The basic iteration or *microrotation* in circular coordinates is:

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

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. About $n+1$ iterations are needed to produce n bit precision [15]. The final coordinates are scaled by the factor

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

which is a constant since $\mid \sigma_i \mid = 1$. Several techniques can be used to compensate this scale factor [10] [27] [30]. The addition of scaling iterations [10] is one of the lowest hardware cost solutions and is considered in this paper.

The sine and cosine functions can be simultaneously obtained by means of the CORDIC algorithm using circular coordinates and in the rotation mode if the initial vector is set to $(\frac{1}{K}, 0)$ [25]. In this case, it is not necessary to compensate for the scale factor.

3. CORDIC Algorithm and variable–precision

In this section we describe the CORDIC algorithm for variable–precision (up to n_{max} bits) supposing a base hardware system with a fixed word size of m bits ($m \leq n_{max}$).

For clarity in the exposition and without any loss of generality, we consider binary fractional representation for the data in the x and y coordinates and one of the initial coordinates $x(0)$ and $y(0)$ is normalized ($0.5 \leq |x(0)| < 1$ or $0.5 \leq |y(0)| < 1$) (the processing that we develop in the next paragraphs is similar to the processing required for the significand in a floating point representation [28]).

The format of the data for variable–precision is as follows: the data is split into a variable set of words having a fixed length m, which coincides with the basic word length of the hardware system [24].

If the selected precision is n, the number of words into which the data is divided is

$$L = \left\lceil \frac{n}{m} \right\rceil \quad (3)$$

Therefore, L cycles are needed to compute a classic CORDIC iteration.

To refer to the different words of a coordinate we employ a subscript that goes from the LSW (Least Significant Word) to the MSW (Most Significant Word).

To make the CORDIC function possible, we must obtain a suitable decomposition of the two basic operations used in CORDIC: addition and shift. This is possible by applying the classic bit–serial techniques to our word–serial system, which are briefly described below.

The addition is quite easy and it can be found in [23]. Basically, we add the different words starting from the least significant word (LSW) using an m bit adder and storing the carry from one word to the next one.

The shift is decomposed into two partial shifts. The first one is a *word level shift* and the second one is a *bit level shift*. For example, to compute $x(i+1)$ we need to shift i bits the value of $y(i)$ (see equations (1)). Let us assume that i is decomposed as $i = d \cdot m + r$ with $r < m$. Therefore, to obtain $[2^{-i} y(i)]$, we have to shift the value $y(i)$ to the right by d full words (*word level shift*) plus r bits (*bit level shift*). Figure 1 shows this situation in the case of $d = 1$.

The computation of $x(i+1)$ is performed by processing the corresponding words of $x(i)$ and $[2^{-i} y(i)]$ sequentially in every cycle, starting from the LSWs. Figure 1 shows also the locations of the corresponding words to be processed.

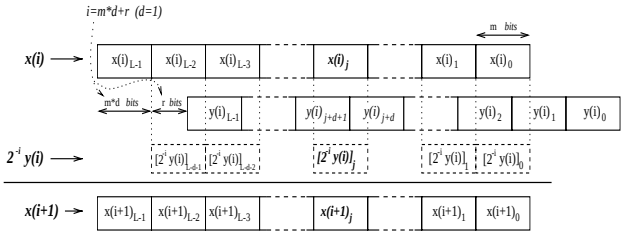

Figure 1. Word level shift and bit level shift.

Therefore, we can express the operation performed in the j-th cycle of the i-th CORDIC iteration as:

$$\begin{aligned} x(i+1)_j &= x(i)_j + \sigma_i \cdot [2^{-i} \cdot y(i)]_j + cx_j \\ y(i+1)_j &= y(i)_j - \sigma_i \cdot [2^{-i} \cdot x(i)]_j + cy_j \\ z(i+1)_j &= z(i)_j - \sigma_i \cdot [\tan^{-1}(2^{-i})]_j + cz_j \end{aligned} \quad (4)$$

where cx_j,cy_j and cz_j are the carries generated in the previous cycle in which $x(i+1)_{j-1}$, $y(i+1)_{j-1}$ and $z(i+1)_{j-1}$ were obtained, and σ_i is the sign of $z(i)$ (rotation mode) or $y(i)$ (vectoring mode).

Summarizing, every CORDIC iteration is made up of L cycles in which one word of the final result is obtained. To obtain the j–th word $x(i+1)_j$ we need $x(i)_j$, the carry produced in the previous cycle cx_j and $[2^{-i}y(i)]_j$; This last value is obtained from the pair of words $y(i)_{j+d+1}$, $y(i)_{j+d}$, with $d = \lfloor \frac{i}{m} \rfloor$ (see Figure 1).

3.1. Scale factor compensation

The compensation for the scale factor for the CORDIC algorithm imposes a significant overhead whose minimization has been attempted by different researchers [27], [30]. The solution that we propose in this subsection is based on that with least hardware cost which is the addition of scaling iterations [10] [14]. Nevertheless, if we are only interested to compute the sine and cosine functions it is not necessary to compensate for the scale factor since it is implicit in the initial coordinates (see section 2).

One scaling iteration has the form:

$$x(i+1) = x(i)(1 + \mu_i 2^{-\psi_i}) \tag{5}$$

where $\mu_i = \pm 1$ and $\psi_i \in \{0, 1, ..., n\}$.

The number of scaling factors and their parameters (μ_i and ψ_i in equation (5)) is a function of the precision n, in such a way that the shortest sequence of scaling iterations is found by means of an intensive search (between $n/4$ in the best cases to $n/2$) [27]. This means that for two different values of n the shortest sequences can be completely different, having no common factor between them. This represents a serious problem for variable–precision since a large amount of memory is required to store every set of parameters of the scaling factors.

Nevertheless, we have found a sequence of approximately $n/3$ factors in such a way that the factors are maintained from one precision to a higher precision. This allows storing a reduced number of factors for any precision. In other words, it is only necessary to store the factors corresponding to the maximum precision, since the sequence of factors for lower precisions are taken from that.

For example, Table 1 shows the parameters of the scaling factor and the total number of them corresponding to precisions up to $n = 512$ and $m = 32$. The parameter ψ have been decomposed as a word level shift (d) plus a bit level shift(r, the sign of r represents the sign of μ). Only the bit level shift has to be stored since the word level shift is implicit.

n	d	r	total number scaling
32	0	$-1,+2,-5,+8,-10,+16,+17,-19,-23,-26,+29,+31$	12
64	1	$+2,-6,+10,+13,-16,-17,+19,-21,-25,+27,-31$	23
96	2	$-1,-5,-11,+14,-16,+19,+20,-22,+24,-26,-29$	34
128	3	$-0,-5,-7,+9,-11,-16,+25,+27$	42
160	4	$-2,+5,-9,-10,+12,-21,-23,-30$	50
192	5	$-0,+2,-6,-10,-13,+16,+24,+25,-27,-30$	60
224	6	$-0,+3,-5,-9,+11,+17,+18,-20,-27,+30$	70
256	7	$-0,+3,+10,+12,+14,-17,+19,-21,-24,+26,-29,-31$	82
288	8	$+1,-8,-10,-13,+16,-19,-21,+23,-25,+28$	92
320	9	$-0,-2,+4,+8,+12,+15,-18,-23,-28,+30$	102
352	10	$-1,+7,+10,+13,+18,+21,+22,-24,-29$	111
384	11	$-4,+7,-11,+14,+17,+18,-20,+22,-26,+29,+31$	122
416	12	$+0,-2,-6,+8,-13,+15,-18,-20,-23,+27,+30$	133
448	13	$+1,+3,+6,+8,-11,+14,-19,+22,-25,-31$	143
480	14	$-2,-4,-5,+7,-10,+13,-16,+21,+26,+27,-29$	154
512	15	$-4,+6,+10,+13,+16,-19,+21,-24,+28,-30$	164

Table 1. Parameters of scaling factors.

4. Variable–precision interval sine and cosine function

The extension of the CORDIC algorithm presented in the previous section allowed us to perform all of the operations that the classic CORDIC is able to support, such as the sine and cosine function, but using variable–precision. Now we extend the algorithm to perform variable–precision interval sine and cosine. Related to this, a preliminary step within the field of fixed precision can be found in [12].

To compute the sine or cosine functions of the interval $[\beta_1, \beta_2]$ and without any loss of generality, we first reduce the argument to the range $[\frac{-\pi}{2}, \frac{\pi}{2}]$ [1][22]. At first, to find the correct bounds, it is necessary to perform two full CORDIC operations (rotations), one for each angle ($2n+2$ iterations).

Since in this work interval arithmetic is focused to bound error in numerical computation, the interval argument is usually narrow. We take advantage of this in such a way that we perform the first rotation through β_1 ($n+1$ iterations), and the second is obtained from the first one by rotation through the *incremental* angle $\Delta\beta = (\beta_2 - \beta_1)$. If the incremental angle $\Delta\beta$ is narrow enough, it is possible to perform the last operation in a few iterations instead of $n+1$, as we probe next.

Let us assume that $\Delta\beta = \beta_2 - \beta_1 < 2^{-j}$ and $j > n/3$. In this case the elementary angles fulfill $\alpha_i = \tan^{-1}(2^{-i}) \simeq 2^{-i}$ ($\forall i \geq j$) [29][33] and therefore $\Delta\beta < \alpha_j$. From the convergence condition of the CORDIC algorithm $\alpha_i \leq \sum_{p=i+1}^{n} \alpha_p + \alpha_n$. Consequently, to achieve the convergence it is not necessary to begin at iteration 0 but at iteration $j+1$, and the rotation through the incremental angle can be carried out in only $n-j$ iterations. Hence, the total number of iterations is $n+1+n-j$ instead of $2n+2$. Nevertheless, to prevent problems with the scale factor, we consider $j > n/2$.

5. Error analysis

The CORDIC algorithm presents two basic sources of errors [15][19]: the approximation error and the rounding error. The size of these errors can be controlled by adding iterations for the first type of error, and adding some extra bits (guard bits) for the second type of error [15].

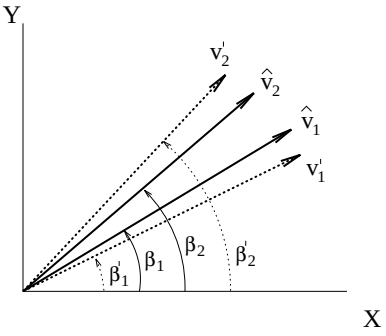

Figure 2. Correct upper and lower bounds.

For interval arithmetic, we must ensure that all interval endpoints are rounded in the proper direction. Next, we analyze the influence of both source of errors and the modifications of the algorithm which is necessary to introduce.

The approximation error: In this case, we must obtain a final interval $[\beta_1', \beta_2']$ which includes the argument interval $[\beta_1, \beta_2]$, as figure 2 shows ($[\beta_1, \beta_2] \subseteq [\beta_1', \beta_2']$).

In Figure 3 we have an example of how to find β_1' from the vector obtained after the first rotation v_1 (n+1 CORDIC microrotations). In this figure, $\hat{v}_1$ is the ideal value (infinite microrotations). In this case v_1 is over $\hat{v}_1$ and therefore $z(n+1) > 0$. To find a lower bound, we perform an extra iteration through a microrotation angle of α_n in such a way that the vector v is placed beyond the vector $\hat{v}$ (v_1'). This is possible because the maximum deviation from $\hat{v}_1$ is up to $\alpha_n = \tan^{-1}(2^{-n})$ (from CORDIC convergence).

To perform the incremental rotation $\Delta\beta$ we start from vector v_1' which is always below the ideal value. Nevertheless, to ensure a correct upper bound of v_2 the initial vector should be placed over $\hat{v}_1$. This forces the addition of an extra initial microrotation through α_n. Similar considerations are required to place the final vector v_2 over the ideal value.

Consequently, to find the two correct bounds a total of one extra iteration is required in the best case, and up to three iterations in the worst case.

The rounding error: The rounding error accumulated through the successive microrotations ε have to be taken into account before the final rounding to guarantee correct endpoints. The maximum value of the accumulated rounding error ε_{max} is added before rounding up to ensure that the ideal value is less than the value used to round [1]. Similarly, it is subtracted before rounding down. The

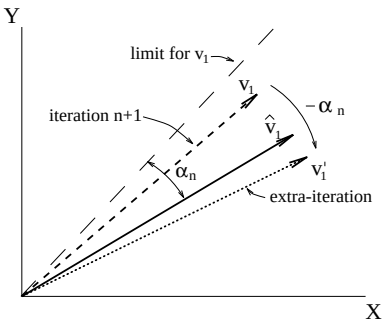

Figure 3. Addition of one extra iteration.

value ε_{max} depends on the number of guard bits such as $\varepsilon_{max} < 1.5n2^{-p}$, where p is the total number of bits including the guard bits [19] [15]. Due to the fact that the incremental rotation starts from a value with error and the number of microrotations is up to $n/2$, the accumulated rounding error for this rotation is less than $1.5\varepsilon_{max}$ instead of ε_{max}. In what follows, and without any loss of generality, we consider one extra word to support the guard bits. Therefore, the number of cycles for every CORDIC iteration is now $L+1$ instead of L.

Taking into account these modifications of the CORDIC algorithm, the worst case error is different for each rotation. For approximation error, it is still α_n for the first rotation but $2\alpha_n$ for the incremental rotation. If these errors are moved to the x and y coordinates, we obtain 2^{-n} and 2^{-n+1} respectively. For the rounding error, since ε_{max} is added, this error is double in the worst case. Hence, it is $2\varepsilon_{max}$ for the first rotation and $3\varepsilon_{max}$ for the second one. Taking into account that the final rounding up or rounding down produce an extra error of one unit in the last place (ULP), the total the worst case error is $2^{-n} + 2\varepsilon_{max} + 1ULP$ for the lower bound and $2^{-n+1} + 3\varepsilon_{max} + 1ULP$ for the upper bound.

6. Architecture

In this section we propose a basic architecture to support CORDIC with variable–precision arithmetic and interval sine and cosine functions. For the sake of simplicity we do not included the standard hardware needed to round. This architecture works with one word as its basic length, needing several cycles to perform every one of the n CORDIC iterations.

Figure 6 shows the architecture for the x, y and z data paths. The shading elements correspond to the hardware added to the variable–precision CORDIC architecture to support the interval sine and cosine functions efficiently. We analyze the x, y data paths and the z data path separately.

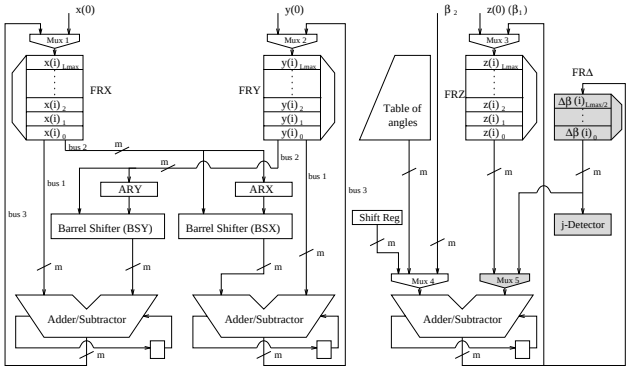

Figure 4. Variable–Precision CORDIC architecture.

6.1. The x, y data path

In what follows, and without any loss of generality, we normally refer to the x coordinate as being similar to the y coordinate. Basically, we can see three parts in the x data path of the architecture of Figure 6: a section to store the coordinate (FRX), a section to support the shift operation ($ARY \& BSY$) and one adder/subtractor.

One of the main differences that can be observed in the x data path between a standard CORDIC architecture and the proposed one is the FRX file register. This element substitutes the classic register which supports the data in every iteration. This is because the coordinates now consist of several words instead of a single word (a similar solution can be find in [23]). The file register has two output ports and one input port.

During every cycle, the word–serial architecture of Figure 6 carries out equations (4). To perform the equation in x of (4) we read the word $x(i)_j$ from the file register FRX (bus 1) and the word $[2^{-i}y(i)]_j$ is supplied by the barrel shifter BSY. The inputs to this barrel shifter are the value $y(i)_{j+d+1}$ (coming from the file register FRY) and the value $y(i)_{j+d}$ (coming from the auxiliary register ARY), where d is the word level shift (see section 3). The adder takes into account the incoming carry cx_j (a flip-flop was incorporated). At the end of the cycle, the value at the output of the adder $x(i+1)_j$ is stored in FRX at the same address where $x(i)_j$ was read (bus 3), the outgoing carry is stored in the flip-flop and the auxiliary register ARY is loaded with the value $y(i)_{j+d+1}$ to be used during the next cycle.

6.2. The z data path

This section of the architecture includes dedicated hardware to perform the interval sine and cosine functions efficiently (shadowing in Figure 6). First, we explain the CORDIC general functionality, and after this we deal with the sine and cosine functions.

As in the x and y data paths, the classical register has been substituted by the file register FRZ. During every cycle, the equation in z of (4) is performed. The value $[\tan^{-1}(2^{-i})]_j$ is obtained from either the table of angles or the shift register of Figure 6, as we explain in the following paragraphs.

At first, the size of the table is $n_{max} \cdot n_{max}$, where n_{max} is the maximum precision supported by the architecture. This table size may be too large (i.e. about 128KBytes for $n_{max} = 1024$). Therefore, it is important to reduce it. The table size may be significantly reduced if we take into account the following considerations:

1. The Taylor series expansion of function $\tan^{-1} 2^{-i}$ is given by:

$$tan^{-1}(2^{-i}) = 2^{-i} - \frac{2^{-3i}}{3} + \frac{2^{-5i}}{5} - \dots \qquad (6)$$

   This expression can be approximated by the first term from iteration $i > \frac{n_{max}}{3}$ because the remaining terms result in an amount that is less than the maximum precision of the system ($2^{-n_{max}}$)[29][33]. Really, with this approximation we are rounding the angle to the nearest value for n bit precision. Therefore, we have $\tan^{-1}(2^{-i}) \simeq 2^{-i}$ and we use a shift register to obtain the values 2^{-i} instead of storing these values in the table, as we can see in Figure 6. This reduces the size of the table to a third of its original size.

2. Analyzing the sequence of zeros and ones of every elementary angle we can observe that at iteration i, we can find a sequence of i consisting of fractional zeros followed by a sequence of $2i$ ones. This special pattern can be explained analyzing the first two terms of the Taylor series expansion given in the equation (6). Hence, it is possible to avoid the storage of those words containing *all zero* and *all one*. The first time this situation happens is when $i = m$ since the non zero MSB of $\tan^{-1}(2^{-i})$ is located in the second most significant word of $z(i)$, and therefore, the m–MSBs of $z(i)$ are all zeros and the next $2m$ bits are all ones. Consequently, every m CORDIC iterations the number of words of the next angles to be stored is reduced by three words which implies that elementary angle $\tan^{-1}(2^{-i})$ requires $L_{max} - 3\left\lfloor \frac{i}{m} \right\rfloor$ words stored in memory.

   This leads to a reduction almost by half with respect to the previous memory improvement, with total memory savings of five–sixths (22.2KBytes for $n_{max} = 1024$). To support this, the shift register of Figure 6 is set to *all ones* just in the cycles i with $i \bmod m = 0$, and one

zero is input to the left in every iteration for the next $m-1$ iterations.

6.3. Additional hardware for efficient interval sine and cosine evaluation

Now, we consider the interval sine and cosine functions. In the method proposed in section 4 is necessary to calculate the difference of angles $\Delta\beta = \beta_2 - \beta_1$. This subtraction can be performed in parallel with the initial load of the rotation angle β_1 in the FRZ register (through MUX–4 and MUX–5), in such a way that the different words of $\Delta\beta$ are calculated and stored in file register $FR\Delta$ sequentially.

Only the first $n/2$ words of $\Delta\beta$ are stored since $\Delta\beta$ must fulfill $\Delta\beta < 2^{-n/2}$ (see section 4), and hence the size of file register $FR\Delta$ is half the size of FRZ. That means that after iteration $i = n/2$ all the differences between the corresponding words of β_2 and β_1 must be zero. This can be detected by the adder/subtractor to abort the further operation with $\Delta\beta$ (in this case a second full rotation is required, see section 4).

To perform the rotation corresponding to $\Delta\beta$, we must calculate the exact iteration j which starts the first microrotation. This can be accomplished by detecting the first leading one of $\Delta\beta$. This operation is performed by the module *j–Detector* of Figure 6 in parallel with the microrotations corresponding to the first rotation (β_1). Basically the module *j–Detector* is composed by a shift register and a counter.

6.4. Timing of one CORDIC iteration

Every CORDIC iteration (that is, equations (1)) is composed of L cycles in which equations (4) are carried out, plus an extra cycle to deal with the guard bits. Furthermore, we need a previous cycle to load the auxiliary registers ARX and ARY of Figure 6 with $x_{i \bmod m}$ and $y_{i \bmod m}$ respectively, just at the beginning of each CORDIC iteration i. Hence, the total number of cycles per iteration becomes $L+2$. This number may be reduced if we take into account some circumstances, as we will show in section 7. Figure 5 shows the cycle distribution of one CORDIC iteration.

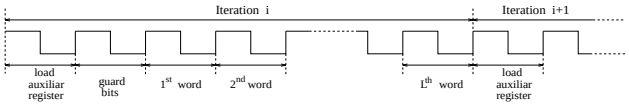

Figure 5. Timing.

7. Number of cycles

A substantial reduction in the number of cycles can be obtained if bit–serial techniques are applied to our system . In this section we give some indications about the application of these techniques in our word–serial variable–precision CORDIC and we calculate the total number of cycles for a full rotation and for a interval sine and cosine evaluation.

7.1. Number of cycles for a whole rotation

For a CORDIC rotation, $n+1$ iterations are required, hence the total number of cycles is $(n+1)\cdot(L+2)$, according to the number of cycles per iteration shown in section 6.4. Nevertheless, this number can be significantly reduced if the addition/subtraction of the leading *all zero* words involved in iterations with $i > m$ is prevented.

We assume that the initial coordinates are both fractional (see section 3) and first we perform the scaling iterations to compensate the scale factor. Under these conditions, the bound of the values added to $x(i), y(i)$ and $z(i)$ to obtain $x(i+1), y(i+1)$ and $z(i+1)$, respectively [30]

$$\begin{aligned} |2^{-i}y(i)| &< 2^{-i} \\ |2^{-i}x(i)| &< 2^{-i} \\ |\tan^{-1}(2^{-i})| &< 2^{-i} \end{aligned} \tag{7}$$

This means that these values have i leading zeros (ones if negative) and then, the computation of the iteration can be stopped when no carry is produced after the word which holds the bit of weight 2^{-i}. This condition must be fulfilled in the three coordinates at the same time and if random values of coordinates are assumed, the probability of a carry propagating is very low.

Nevertheless, in the *rotation mode*, coordinate z tends to zero as the iterations progress, in such a way that the final value becomes zero for the precision required. In this case, the change in the sign of this coordinate occurs quite often and thus, a carry propagating to the MSWs is produced continuously, strongly reducing the possibility of stopping the computation. For the convergence conditions of the CORDIC algorithm, we have

$$|z(i)| < \tan^{-1}(2^{-i}) < 2^{-i} \tag{8}$$

From this expression we conclude that the $z(i)$ coordinate also has i leading zeros (or ones if negative). Therefore, if only the sign bit is stored instead of the sign extension words, the carry propagation in $z(i)$ coordinate is avoided . This means that the "no carry" condition must be fulfilled only on the $x(i)$ and $y(i)$ coordinates which have a random distribution of "0" and "1".

In *vectoring mode*, coordinate y tends to zero as the iterations progress instead of z, and the same considerations and solution may be used since the roles of the z and y coordinates are interchanged.

We conclude that, if the base word length of the architecture is m and no carry is produced, every m iterations we can eliminate the computation of one word, in such a way that for CORDIC iteration i the number of words to compute is $L+2-\lfloor\frac{i}{m}\rfloor$ instead of $L+2$. If a carry is produced, this number increases depending on the carry propagating through the next words.

We have studied the probability of the occurrence of a carry. We have assumed that the values of the coordinates are random and the bits of this number can be viewed as independent and with probability 0.5 for 0 as well as 1[1]. Taking into account that the probability of a carry for the coordinates x, y (x, z for vectoring mode) is the same, we have found that the mean of cycles per iteration which are needed to add when a carry occurs is less than $2/m$ [11]. Hence, the mean number of cycles per iteration is $L+2-\lfloor\frac{i}{m}\rfloor+\frac{2}{m}$.

Therefore, the average number of cycles for a full rotation is

$$N_r=\sum_{i=0}^{n} L+2-\left\lfloor\frac{i}{m}\right\rfloor+\frac{2}{m}=\frac{n}{2}\left(L+5+\frac{4}{m}\right)+2+\frac{2}{m} \tag{9}$$

From this expression and for practical values of m and L, the number of cycles required to perform a whole rotation is approximately halved.

7.2. Number of cycles for interval sine and cosine evaluation

For a interval sine and cosine function evaluation, we have seen that in general, a whole rotation plus a few iterations are needed instead of two whole rotations. Since the number of cycles per iteration goes down with i and the incremental rotation starts at with high values of i, the total number of cycles for the second rotation is very low.

Let $2^{-j-1}\leq\Delta\beta<2^{-j}$, then the number of cycles is:

$$N_{int}=N_r+\sum_{i=j+1}^{n} L+2-\left\lfloor\frac{i}{m}\right\rfloor+\frac{2}{m} \tag{10}$$

Figure 6 shows the total number of cycles as a function of the size of the interval argument $\Delta\beta$ for the computation of the interval sine and cosine functions for variable–precision. The number of cycles corresponding to one full rotation has been also displayed in the figure, and we refer to it as FRC. We can see that for wide interval arguments ($\Delta\beta>2^{-n/2}$) the number of cycles corresponds to that of two full rotations ($2tFRC$) since we can not take advantage of the incremental rotation (see section 4). For $\Delta\beta=2^{-n/2}$ the number of cycles falls to approximately $1.25FRC$ times the cycles of one full rotation. From this point on, the curve is almost exponential (for example, for $\Delta\beta=2^{-n/4}$ the number of cycles is $1.07FRC$), in such a way that it finishes with $1FRC$ when the interval argument becomes a point argument ($\Delta\beta=2^{-n}$).

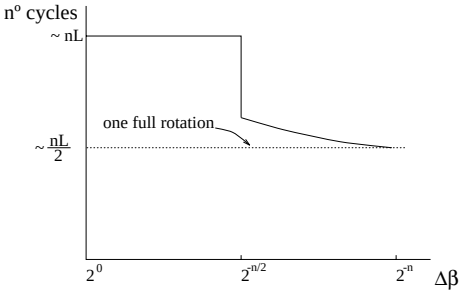

Figure 6. Number of cycles for interval evaluation.

These results can be improved if digits skipping technique are employed [13]. This technique is based on skipping consecutive zeros and Booth recoding of the consecutive ones in the z coordinate after iteration $n/2$. In this way, we only perform microrotations for bits equal to one in $z(n/2)$, which allow us to reduce the average number of cycles by 62% in the incremental rotation. In a whole rotation, this reduction is about 31% in the number of iterations but only about 15% in the number of cycles.

8. Conclusion

This paper presented a word–serial architecture to support the CORDIC algorithm for variable–precision. The proposed system represents direct hardware support to avoid the long computation time of software solutions for the variable–precision elementary functions managed by the CORDIC algorithm. Based on this architecure, a new algorithm has been proposed to compute the variable–precision interval sine and cosine functions efficiently. This system allows us to specify the precision to perform the sine and cosine functions and control the accuracy of the result, in such a way that recomputation of inaccurate results can be carried out with higher precision.

Each variable–precision CORDIC iteration is composed of several cycles with a cycle time similar to the iteration time of a m bit standard CORDIC. The number of cycles has been reduced by half applying bit–serial techniques. One of the most important drawbacks was the large size of the angle table, but this has been reduced to one–sixth of the original size. The number of scaling factors have been unified

[1] For the sine and cosine functions the initial coordinates are not random ($x(0)=1/K, y(0)=0$), but the distribution of "1" and "0" can be considered random after a few iterations

for any precision, needing only $n/3$ iterations to compensate the scale factor.

To implement the interval sine and cosine fuctions efficiently some minor modifications have been performed on the basic variable–precision CORDIC architecture. This allows us to greatly reduce the number of iterations. Furthermore, the number of cycles per iteration of the incremental rotation is very low, because the bit serial techniques of the variable–precision system permits the elimination of a large number of cycles when i is high. Consequenctly, for most cases the total computation time for interval function is close to that of a point function.

References

[1] K. Braune. Standard functions for real and complex point and interval arguments with dynamic accuracy. *Computing, Suppl.*, (6):159–184, 1988.

[2] R. B. Brent. A FORTRAN multiprecision arithmetic package. *ACM Trans. Mathematical Software*, 4:57–70, 1978.

[3] T. M. Carter. Cascade: Hardware for high/variable precision arithmetic. In M. D. Ercegovac and E. Swartzlander, editors, *Proc. 9th Symposium on Computer Arithmetic*, pages 184–191, 1989.

[4] D. M. Chiarulli, W. G. Rudd, and D. A. Buell. DRAFT: A dynamically reconfigurable processor for integer arithmetic. In *Proc. 7th Symposium on Computer Arithmetic*, pages 309–318, 1985.

[5] M. S. Cohen, T. E. Hull, and V. C. Hamarcher. CADAC: A controlled-precision decimal arithmetic unit. *IEEE Trans. on Computers*, C-32:370–377, 1983.

[6] J. S. Ely. The VPI software package for variable precision interval arithmetic. *Interval Computation*, 2:135–153, 1993.

[7] J. S. Ely and G. R. Baker. High-precision calculations of vortex sheet motion. *Journal of Computatinal Physics*, 111(2):275–281, 1994.

[8] R. K. et al. *C-XSC:A C++ Class Library for Extended Scientific Computing*. Springer-Verlag, 1993.

[9] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23:5–48, 1991.

[10] G. Haviland and A. Tuszynski. A CORDIC arithmetic processor chip. *IEEE Trans. on Computers*, C-29(2):68–79, Feb 1980.

[11] J. Hormigo, J. Villalba, and E. L. Zapata. CORDIC and variable-precision. *Internal Report UMA-DAC-97/18. University of Malaga, Spain*, 1997.

[12] J. Hormigo, J. Villalba, and E. L. Zapata. A hardware approximation to interval arithmetic for sine and cosine functions. *Volume of Extended Abstracts SCAN-98, Budapest (Hungary)*, pages 58–59, September 1998.

[13] J. Hormigo, J. Villalba, and E. L. Zapata. CORDIC algorithm with digit skipping. *Proceeding of 32th. Asilomar Conference on Signals, Systems, and Computers*, November 1998 (to appear).

[14] Y. Hu. CORDIC–based VLSI architectures for Digital Signal Processing. *IEEE Signal Processing Magazine*, (7):16–35, July 1992.

[15] Y. Hu. The quantization effects of the CORDIC algorithm. *IEEE Trans. on Signal Processing*, 40(4):834–844, 1992.

[16] V. Kantabutra. On hardware for computing exponential and trigonometric fuctions. *IEEE Transactions on Computers*, 45(3):328–339, March 1996.

[17] A. Knofel. Hardware kernel for scientific/engineering computations. *Scientific Computing with Automatic Result Verification, E. Adams and U. Kulisch, eds.*, pages 549–570, 1993.

[18] O. Knuppel. PROFIL/BIAS - a fast interval library. *Computing*, 53:277–288, 1994.

[19] K. Kota and J. R. Cavallaro. Numerical accuracy and hardware tradeoffs for CORDIC arithmetic for special purpose processors. *IEEE Trans. Computers*, 42(7):769–779, July 1993.

[20] R. E. Moore. *Interval Analysis*. Prentice Hall, 1966.

[21] M. Müller, C. Rüb, and W. Rülling. Exact accumulation of floating-point numbers. In P. Kornerup and D. W. Matula, editors, *Proc. 10th Symposium on Computer Arithmetic*, pages 64–69, 1991.

[22] D. M. Priest. Fast table-driven algorithms for interval elementary functions. *Proc. 13th Symposium on Computer Arithmetic*, pages 168–174, 1997.

[23] M. J. Schulte. *A Variable-Precision, Interval Arithmetic Processor*. PhD thesis, The University of Texas at Austin, May 1996.

[24] M. J. Schulte and E. E. Swartzlander, Jr. Hardware design and arithmetic algorithms for a variable-precision, interval arithmetic coprocessor. In *Proc. 12th Symposium on Computer Arithmetic*, pages 222–229, 1995.

[25] N. Takagi, T. Asada, and S. Yajima. Redundant CORDIC methods with a constant scale factor for sine and cosine computation. *IEEE Transations on Computers*, 40(9):989–995, Sept 1991.

[26] P. T. P. Tang. Table-lookup algorithms for elementary functions and their error analysis. In *Proc. 10th Symposium on Computer Arithmetic*, pages 232–236, 1991.

[27] D. Timmermann, H. Hahn, B. Hosticka, and B. Rix. A new addition scheme and fast scaling factor compensation methods for CORDIC algorithms. *The VLSI journal of integration*, (11):85–100, 1991.

[28] D. Timmermann, B. Rix, H. Hahn, and B. Hosticka. A CMOS Floating-Point Vector-Arithmetic Unit. *IEEE Journal of Solid-State Circuits*, 29(5):634–639, May 1994.

[29] J. Villalba. Diseño de Arquitecturas CORDIC multidimensionales. *Ph.D Universidad de Malaga*, November 1995.

[30] J. Villalba, J. Hidalgo, E. Antelo, J. Bruguera, and E. Zapata. CORDIC Architecture with Parallel Compensation of the Scale Factor. *Proc. Int. Conf. on Application Specific Array Processors (ASAP'95)*, pages 258–269, July 1995.

[31] J. Volder. The CORDIC Trigonometric Computing Technique. *IRE Trans. Elect. Comput.*, EC(8):330–334, 1959.

[32] J. Walther. A unified algorithm for elementary functions. *Proc. Spring. Joint Comput. Conf.*, pages 379–385, 1971.

[33] S. Wang, V. Piuri, and E. E. Swartzlander, Jr. Hybrid cordic algorithms. *IEEE Trans. Computers*, 46(11):1202–1207, November 1997.