Dynamically Reconfigurable CORDIC Coprocessor for Trigonometric Computing

Francisco Fons[1], Mariano Fons[1], Enrique Cantó[1], Mariano López[2]

[1] Departament d'Enginyeria Electrònica, Elèctrica i Automàtica
Universitat Rovira i Virgili (URV)
ETSE, 43007 Tarragona, Spain
[2] Departament d'Enginyeria Electrònica
Universitat Politècnica de Catalunya (UPC)
EPSEVG, 08800 Vilanova i la Geltrú, Spain

Abstract: Engineering applications often demand high-performance processors to carry out specific compute-intensive tasks. This work describes the hardware-software co-design of the CORDIC (Coordinate Rotation Digital Computer) algorithm, all embedded in a system-on-chip device. This platform, based on an 8-bit RISC microcontroller and a dynamically reconfigurable FPGA, makes feasible the efficient implementation of the shift-add algorithm through a 32-bit fixed-point trigonometric computer that evolves on-the-fly to process functions as sin(z), cos(z), atan(y/x) and sqrt(x^2+y^2). Its balanced architecture – a low-cost processor extended by a dedicated slave coprocessor to accelerate the calculus – reaches significant improvements in throughput over conventional software-oriented solutions usually inspired on powerful 32-bit stand-alone microprocessors.

1 Introduction

There exist a broad field of scientific applications where the extensive trigonometric computing is present. As example, navigation systems responsible for calculating trajectories in real-time (satellites, radars, etc) make use of Trigonometrics. A well-known technique for trigonometric calculus is the CORDIC algorithm thanks to its implementation simplicity, efficiency and elegance. Originally credited to Volder [Vo59] and later generalized by Walther [Wa71], the CORDIC concept consists in rotating a 2-D vector a desired angle θ along a circular, linear or hyperbolic coordinate system decomposing it into a sum of predefined elementary angles θ_i such that, iteratively, in each step i, the rotated angle θ_i can be expressed as a value that depends on the i-th power of 2, what finally is computed by simple binary shifts and additions, and where the result is more and more accurate as the number of iterations increases since the vector orientation is successively closer to its target or convergence point.

The use of FPGAs emerged as a viable means of offsetting microprocessor performance limitations in applications that require high-speed processing of large data, as CORDIC computing. Standard DSP processors can be ill-suited to perform at the

required rates due to the serial nature of their architecture or to the lack of certain instructions set extension. FPGAs have been successfully used to mitigate these problems by performing them in hardware, bypassing the sequential stored-program techniques in favour of parallel and dedicated logic functions. Moreover, the multi-purpose CORDIC algorithm exhibits additional characteristics useful to implement by evolvable hardware because of the high similarity among its different operation modes.

After this introduction, section 2 describes the theoretical CORDIC aspects. Section 3 covers the technical criteria followed to implement the trigonometric coprocessor on the Atmel AT94K40 device. In section 4, the experimental results are discussed. Finally, the conclusions are presented in section 5.

2 CORDIC Algorithm

Given a vector in a 2-D coordinate system, the CORDIC method permits to compute trigonometric functions such as sine-cosine of the angle θ described by the vector in the coordinate system as well as its magnitude-phase components. The rotation θ that moves the vector from (x_0,y_0) to (x_n,y_n) is defined by the equations matrix [VT99]: (1)

$$\begin{bmatrix} x_n \\ y_n \end{bmatrix} = \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} = \frac{1}{\sqrt{1+tan^2\theta}} \begin{bmatrix} 1 & -tan\theta \\ tan\theta & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \end{bmatrix}, \quad cos\theta = \frac{1}{\sqrt{1+tan^2\theta}}.$$

The CORDIC algorithm is inspired on performing this effective rotation θ as an iterative process based on successive rotations through which the initial vector (x_0,y_0) is rotated by predetermined step angles θ_i. This mechanism can operate in two different modes:

- In rotation mode, the initial components (x_0,y_0) of the vector and the effective rotation angle $z_0=\theta$ are given in order to compute the new coordinate components of the resultant rotated vector. For this, in each rotation step i, fixed angles are subtracted or added from/to the angle accumulator z so that this remainder angle approaches to zero.
- In vectoring mode, the coordinate components of the vector (x_0,y_0) are known and the magnitude and phase of this original vector are computed by rotating the input vector to the X axis at the same time as storing the accumulated angle of this trajectory.

Taking in mind that any angle θ can be decomposed into a set of n step angles θ_i in a certain accuracy (2), the successive rotations of the algorithm are quantified such that $tan\theta_i$ represents a series of powers of 2, that is, angle steps θ_i of value $atan2^{-i}$ that arithmetically amount to successive binary shifts and additions left in a good place to be efficiently implemented by hardware:

$$\theta = \sum_{i=0}^{n-1} s_i\theta_i + \varepsilon, \qquad tan\theta_i = s_i 2^{-i}, \qquad s_i \in \{-1,1\}, \qquad i = 0,\ 1,\ 2,\ 3,\ \ldots,\ n-1 \tag{2}$$

where s_i represents the sign or direction of each rotation i, and the error ε converges to zero when n is big enough. Therefore, the rotation from i to $i+1$ results in: (3)

$$\begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = \frac{1}{\sqrt{1+tan^2\theta_i}} \begin{bmatrix} 1 & -tan\theta_i \\ tan\theta_i & 1 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} = k_i \begin{bmatrix} 1 & -s_i 2^{-i} \\ s_i 2^{-i} & 1 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix}, \quad k_i = 1/\sqrt{1+2^{-2i}}.$$

If the factors k_i are removed from Eqn. (3) and an auxiliary variable z_i is introduced to

compute the accumulated angle, finally the called CORDIC micro-rotations are obtained:

$$x_{i+1} = x_i - s_i y_i 2^{-i}, \quad y_{i+1} = y_i + s_i x_i 2^{-i}, \quad z_{i+1} = z_i - s_i atan2^{-i}. \tag{4}$$

These three difference equations, restricted to angles comprised within the range $-90^o \leq \theta \leq +90^o$ due to convergence reasons, define the CORDIC algorithm for trigonometric computing. The fact of not considering the removed k_i factors in the final equations makes a CORDIC micro-rotation be not a pure rotation but a rotation with an intrinsic increase of the magnitude of the resultant vector that is quantified by the term A_n. Thus, in a CORDIC rotation, after n iterations: (5)

$$\begin{bmatrix} x_n \\ y_n \end{bmatrix} = A_n \begin{bmatrix} cos\left(\sum_{i=0}^{n-1} s_i\theta_i\right) & -sin\left(\sum_{i=0}^{n-1} s_i\theta_i\right) \\ sin\left(\sum_{i=0}^{n-1} s_i\theta_i\right) & cos\left(\sum_{i=0}^{n-1} s_i\theta_i\right) \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \end{bmatrix}, \quad A_n = \prod_{i=0}^{n-1} \sqrt{1 + tan^2\theta_i}, \quad K_n = \prod_{i=0}^{n-1} k_i = 1/A_n.$$

Although the value of the distortion or scale factor A_n depends on the number of iterations n (micro-rotations), this term approaches to the constant 1.6467 as the number of iterations goes to infinity. On the other hand, in a real rotation its value would be 1.

In rotation mode, the angle accumulator z is initiated with the target angle θ and the decision concerning the direction of rotation taken at each iteration is made to reduce the magnitude of the residual angle present in that angle accumulator. This criterion is therefore based on the sign of the resultant angle after each step. The sign rules along with the resultant outputs are as follows:

$$s_i = \begin{cases} -1, & z_i < 0 \\ +1, & z_i \geq 0 \end{cases} \qquad i = 0,\ 1,\ \ldots,\ n-1 \tag{6}$$

$$x_n = A_n\left(x_0 cosz_0 - y_0 sinz_0\right), \quad y_n = A_n\left(y_0 cosz_0 + x_0 sinz_0\right), \quad z_n \to 0 \qquad when\ \ n \to \infty.$$

The elementary functions sine and cosine can be computed from Eqn. (6): if the initial vector is of unit magnitude and is aligned with the abscissa, that is, x_0=1, y_0=0 and the angle accumulator is initialized to $z_0=\theta$, then the results obtained in x_n and y_n equals to $A_n cos\theta$ and $A_n sin\theta$. An additional division is needed to compensate the scaling factor A_n originated in the CORDIC rotations and thus get $sin\theta$ (y path) and $cos\theta$ (x path).

In vectoring mode, the input vector (x_0,y_0) is rotated until the resultant vector gets aligned to the X axis. For this, the term y of the vector is step-by-step minimized until converging to zero. If the angle accumulator is started with zero (z_0=0), at the end of the rotation loops it will contain the effective rotated angle $\theta=atan(y_0/x_0)$. Apart from the angle, another simultaneous result obtained is the magnitude of the original vector scaled by the gain A_n, which is stored into the component x and can be corrected by multiplying it by K_n. Thus, the vectoring mode simultaneously provides the magnitude and the phase of the original vector involved in the cartesian to polar coordinates conversion. In this iterative mechanism, the sign of the residual component y determines the direction of the following rotation step in accordance with the rule:

$$s_i = \begin{cases} +1, & y_i < 0 \\ -1, & y_i \geq 0 \end{cases} \qquad i = 0,\ 1,\ \ldots,\ n-1 \tag{7}$$

$$x_n = A_n\sqrt{x_0^2 + y_0^2}, \quad z_n = z_0 + atan\left(y_0/x_0\right), \quad y_n \to 0 \qquad when\ \ n \to \infty.$$

3 Hardware/Software Co-design. Dynamic Partial Reconfiguration

Despite the habitual trend of implementing digital signal processing algorithms by means of DSPs or general-purpose processors, the CORDIC algorithm gets optimized through dedicated hardware owing to its potential customizable parallelism as well as other reasons enumerated next [Ti91], [An98]:
- The algorithm presents an iterative mechanism easily synthesizable on a FPGA. Many of the involved elementary functions such as trigonometric, exponential and logarithmic operations, on the contrary, cannot be efficiently evaluated with multiply-accumulate (MAC) units present on DSP processors (consequently, whenever algorithms incorporate these functions, it is not unusual to observe significant performance degradation).
- The accuracy of the result depends basically on the number of iterations done by the algorithm and the data size of the operands involved in the operations. This characteristic demands to increase the word width to the necessary length in accordance with the desirable precision, and this fact, unlike the microprocessor architecture, can be customized into a FPGA: in an arithmetic process that handles products and divisions, for instance, it is usual that intermediate results have larger size than the final result. In a MCU, the length of the ALU buses delimits the maximum data range admissible in the computation. In ASIC or FPGA-based designs, this area restriction can be optimized by splitting the computing in phases and adapting each parallel bus size to the range of data needed in each of them.

Additionally, after inspecting the previous section, an outstanding characteristic is noticed that gives rise to a novel implementation strategy: the only difference between calculating the sine/cosine or the arctangent/magnitude functions is found in the sign criterion, given that the three CORDIC equations (4) remain invariable in both rotation and vectoring modes. This fact let us inspire the implementation on flexible hardware; our approach divides the computing unit in a static circuitry or hardware skeleton that remains unchanged and a dynamic circuitry or reconfigurable block that can evolve at run-time depending on the mathematic function to compute. In this way, our design deals with the implementation of a CORDIC coprocessor under a system-on-chip (SoC) device with dynamic partial reconfiguration performances. The chosen platform is the Atmel AT94K40, also known as FPSLIC (Field Programmable System Level Integration Circuit), especially thanks to its fine-grained architecture from a configuration point of view: the system is all integrated on a single-chip device composed of on an 8-bit AVR MCU and an AT40K40 FPGA where hardware coprocessors can be instantiated. Moreover, FPSLIC supports full and partial dynamic reconfiguration: the entire device or select portions can be reconfigured at run-time by the MCU or the FPGA itself through the internal configuration controller, while the remaining logic keeps active and operates normally without any disruption [At01]. In this way, MCU and FPGA work seamlessly in computing elementary functions such as $sin(z_0)$, $cos(z_0)$, $atan(y_0/x_0)$ and $magn(y_0,x_0)$.

```
long sin(char);
long cos(char);
long atan(char, char);
long magn(char, char);
```

Code 1. Prototypes of the trigonometric functions

These functions are instantiated by the MCU and processed by hardware in the FPGA to speed up the computing. The MCU reconfigures the FPGA to synthesize the required IP core depending on the function called by the software code. The FPGA execution is performed when demanded by a user program: when the applications programmer invokes a function that is supported in hardware, this results in starting an automatic mechanism that handles the MCU-FPGA data transfer and FPGA partial reconfiguration to download the specific hardware computer into the FPGA and perform the trigonometric operation. For this, the software is organized in a model of two layers:
- A low-level or hardware abstraction layer offers the library of drivers that the high-level software developer can use without taking care about the hardware platform in use. These platform-dependant routines make both data transfer and partial reconfiguration tasks transparent to the high-level programmer, who does not bother of the functions implementation but only needs to know the API (application program interface) to call them, as shown in Code 1.
- A high-level or application layer. This part of the application code constitutes the high-level design. It can be exported to other hardware platforms keeping a full compatibility whereas the low-level layer takes charge of carrying out the platform customization.

3.1 Coprocessor Architecture

Many research efforts have been directed to develop CORDIC-based architectures for computing applications [Hu92], [VM02]. Two types of architectures are considered depending on the speed-area tradeoff intended by the application:
- Iterative CORDIC implementation. An iterative architecture is simply obtained by synthesizing in hardware each of the three difference equations (4). The processing is composed of binary shifts and arithmetic additions/subtractions where the partial results of each loop are stored in accumulator registers.
- Unrolled or on-line CORDIC topology. The previous iterative architecture can be unrolled n times, giving rise to an on-line implementation. Unrolling the processor results in several significant simplifications: the need for registers is eliminated making the processor strictly combinatorial and the n-shifters and look-up values for the angle accumulator can be wired and implemented distributed as hardwired constants.
Both types of architectures can be synthesized by bit-serial or bit-parallel data paths.

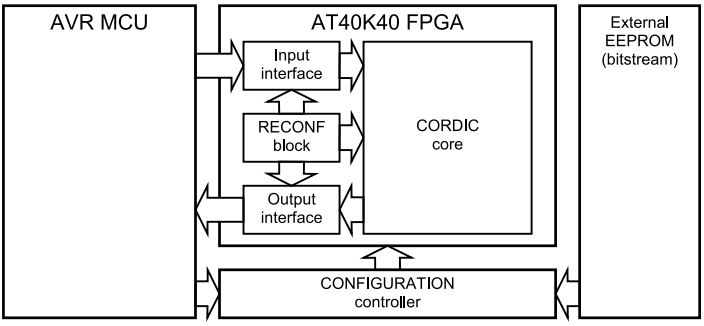

Fig. 1. Block diagram of the AT94K40-based trigonometric CORDIC coprocessor

Our trigonometric coprocessor is based on an iterative bit-parallel CORDIC architecture. It is divided into several blocks:

- Microprocessor. The 8-bit AVR is responsible for executing the C program that contains the different calls to the trigonometric functions directly computed on the FPGA. With regard to the hardware/software partitioning, the hardware coprocessor monopolizes the arithmetic functions whereas the low-cost MCU assumes the data management under a master-slave topology. This solution is an alternative to the use of a stand-alone 32-bit processor (e.g. ARM) to implement all the computing by software. The fact of having an FPGA makes possible, in turn, to require a CPU of lower power.
- Input interface. The coprocessor exchanges data between MCU-FPGA. An input interface allows the MCU to transfer the function arguments to the FPGA registers.
- Output interface. The FPGA performs the computing and the result is send back to the MCU through the output interface. Afterwards, the FPGA remains inactive until a new hardware-supported function is called from the C code of the user application.
- CORDIC core. The trigonometric calculus is implemented by iterative and parallel hardware architecture. Together with the hardware skeleton there is a finite state machine that handles the *n* iterations required until obtaining the final result.
- RECONF block. Our coprocessor concept divides the design into a static part and a dynamic part. Initially, after a system reset, the bitstream is automatically downloaded from an external memory to configure the whole device; the basic CORDIC skeleton is loaded in the FPGA and, from now on, only partial reconfigurations are required to customize the coprocessor to the concrete trigonometric function on demand or in progress. The RECONF block is directly managed by the MCU and makes possible to modify some parts of the algorithm that affect to the input, CORDIC and output blocks.

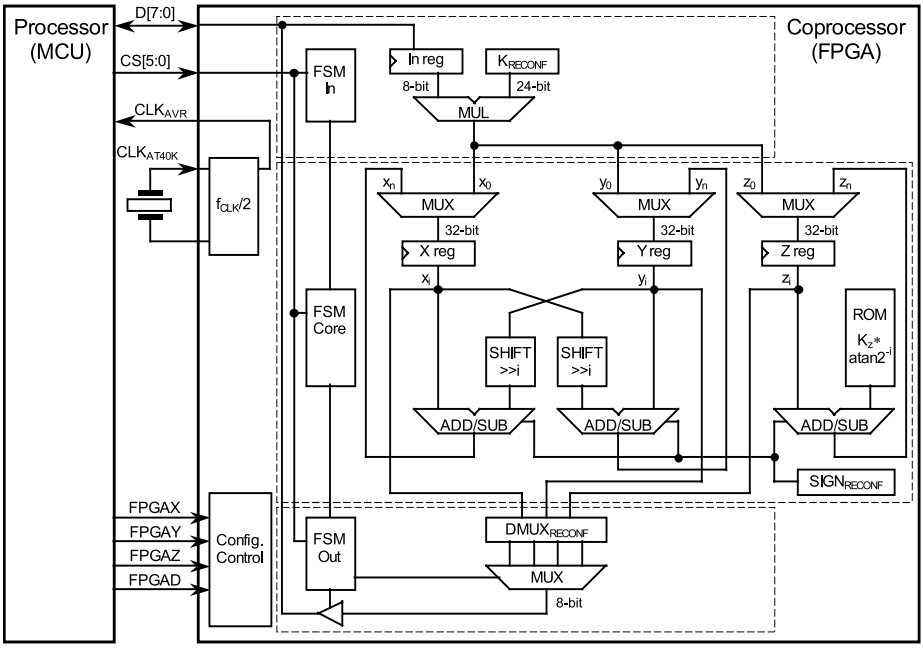

Fig. 2. Internal structure of the CORDIC coprocessor

4 Experimental Results

The design, described in C and VHDL languages, has been divided in two stages:
- A first stage consisting in developing the entire algorithm exclusively by software in a PC platform. This stage studies the accuracy required by our coprocessor according to the precision of the data and the number of iterations considered.
- The second stage goes deeply into the Hw/Sw co-design of the algorithm and its implementation on the FPSLIC. The results obtained in the first stage are useful here: a profiling of all the tasks involved in the software algorithm lets us recognize the bottlenecks of the application to achieve a reasonable hardware/software partitioning.

The coprocessor operates with 8-bit integer data as inputs and gives 32-bit data in fixed-point representation as result. Concerning the input data range analysis, the input values for sine and cosine functions goes from –90º to 90º, and for magnitude and arctangent any 8-bit integers that give rise to an resultant angle fitted within the first or fourth quadrant. The precision of the results is determined by both angular and truncation errors [Li99]. The angular error depends on the number of iterations performed whereas the truncation error is function of the data width established for the operands involved in the calculus, due to limited storage capability or area restrictions. Hence, our CORDIC implementation operates with 32-bit data and 32 iterations are carried out. The hardware design comprises 1423 logic cells, what means the 61.8% of the FPGA resources.

4.1 FPGA-Based Implementation. Reconfigurable Components

Concerning the dynamic partial reconfiguration aspects of the design, our interest is focused on the synthesizable 4-in/1-out or 3-in/2-out look-up table (LUT) present in each logic cell of the FPGA. In general, our design consists of a static circuitry that constitutes the CORDIC skeleton and three flexible blocks that their behavioral descriptions are conditioned to the particular function to implement since they differ from one to other:
- Input RECONF block. As discussed in section 2, the results of the CORDIC equations are affected by the gain factor A_n defined in Eqn. (5). Its inverse, K_n, in our design is a pre-calculated constant applied to the initial values x_0 and y_0 to compensate this amplifier effect. A multiplier synthesized at the input block does this operation, as shown in Fig 2. Our application takes n=32, what gives rise to K_{32}=0.607252935. The coprocessor works with data in fixed-point numbering format, to be exact, in 6 decimal digits. Thus, the scaling factor applied to the terms x and y would be 607252.935. In the same way, the angle is expressed in degrees but to 6 decimal places, what means a factor of 10^6 for the term z. After doing a data range analysis, finally these constants are shifted 3 bits to the left. The resultant multiplication factors are shown next.

Table 1. Numerical representation of the CORDIC corrective constants K_{32}

Variable	K_{CORDIC} (dec)	K_{CORDIC} (hex)	K_{CORDIC} (bin)
X	4858023	4A20A7	0100 1010 0010 0000 1010 0111
Y	4858023	4A20A7	0100 1010 0010 0000 1010 0111
Z	8000000	7A1200	0111 1010 0001 0010 0000 0000

These constants are assigned depending on the variable to transfer: instead of multiplexing them by a *mux2x24*, this can be performed by reconfiguring an only logic cell of the FPGA. In fact, taking advantage of the fine-grain FPGA characteristics and the easy access to the configurable resources by the configuration controller, it is not necessary to synthesize a generic *mux* with its select lines controlled by a dedicated finite state machine (FSM); this can be handled by the MCU through reconfiguring the LUT of a logic cell in order to negate or not its input and in this way generate two outputs that are applied to the bits that differ from a constant to another, as depicted in Fig. 3. The input is permanently tied to '0' and the output switching takes place by reconfiguring the logic function. Following this strategy, the routing of the design is fixed and only some logic resources change. Hence, each time a value has to be loaded from the MCU to the computing registers, the MCU previously reconfigures on-the-fly the corrective factor *K* accordingly: two partial reconfigurations are performed in each trigonometric calculus, one for both variables X and Y, and another for Z.

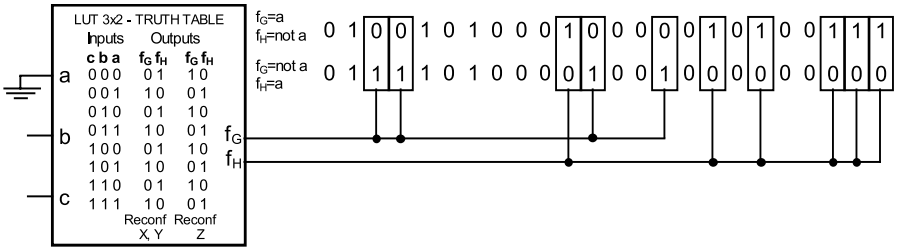

Fig. 3. Multiplexing of K_{CORDIC} by dynamic partial reconfiguration

- Sign RECONF block. Rotation and vectoring modes only differ in the sign criterion applied. In rotation mode, z_i is considered for taking the addition/subtraction decision as shown in Eqn. (6) whereas in vectoring mode the variable y_i becomes the selection key in accordance with Eqn. (7). Like this, the sign criterion can be implemented through an only logic cell combining a LUT of 2 inputs, the signs of y_i and z_i. As above, by solely reconfiguring the 8-bit truth table of this logic cell, one of both sign criteria is applied to the three adder/subtract modules present in the coprocessor shown in Fig. 2.

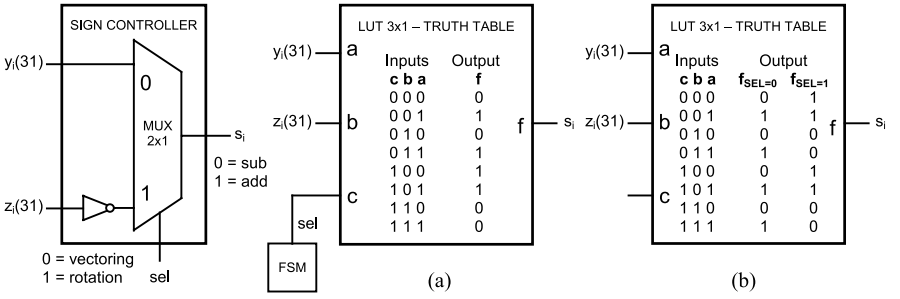

Fig. 4. Sign controller. Static version (a) versus dynamic version (b)

- Output RECONF block. The result of the CORDIC algorithm is located in one of the three registers x_i, y_i and z_i, depending on the function selected. The three 32-bit wide registers are sequentially transferred to the MCU through a bidirectional 8-bit data bus.

According to this restriction, a smaller static *mux4x8* is implemented to select each of the four bytes that compound the 32-bit data, and the selection among the three 32-bit registers is done by other 32 dynamic *mux3x1*, each of them implemented in a logic cell using a reconfigurable LUT. In FPSLIC, taking into account the internal logic cell structure based on a 4x1 or 3x2 LUT, a static *mux3x1* is implemented with two logic cells since five inputs are required. On the other hand, this same *mux3x1* can be synthesized dynamically in only one logic cell, what represents a 50% of area saving (even more if the FSM necessary to control the select lines of the multiplexer are also considered, whereas in a dynamic multiplexer it is not needed since the MCU program does it) and also a considerable time reduction (inertial time and mainly transport time thanks to the inherent routing simplification) of the data path. Hence, reconfiguring this dynamic multiplexer of 32 *mux3x1* involves rewriting the 32 affected LUTs.

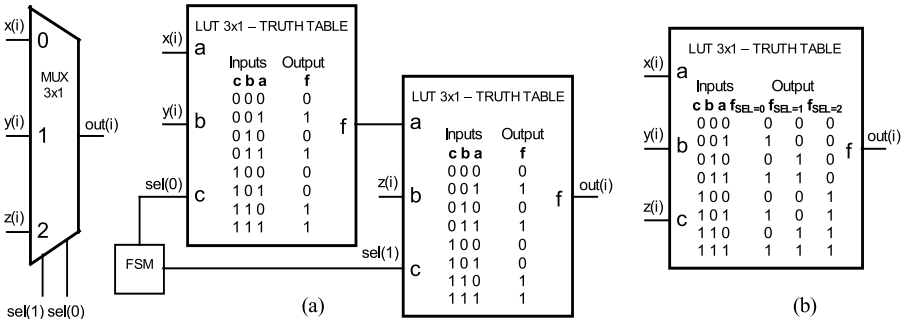

Fig. 5. Static 3x1-multiplexer (a) versus dynamic 3x1-multiplexer (b)

4.2 Performance Evaluation

In order to realize the benefits of the Hw/Sw co-design of the CORDIC algorithm, the efficiency of this coprocessor has been compared against a shift-add software-based implementation running on several PC platforms. Table 2 shows the performances obtained in computing an identical trigonometric calculus under different systems. Taking in mind the lower working frequency of our prototype, our system is able to process the trigonometric calculus in fewer clock cycles than whichever software-oriented approach. Our multiprocessor platform reduces the execution time through the scheduling of concurrent MCU-FPGA tasks. The FPGA only needs 32 cycles to perform the calculus. On the other hand, the number of clock cycles is deeply extended in a software-oriented implementation, getting the critical part of the CORDIC execution.

Table 2. Comparison of different Hw/Sw implementations of the CORDIC algorithm

Platform (Operating System)	Time (ns)	Development Tools
Pentium 4 @ 2.66GHz (Windows XP)	5050	MS Visual C++ 6.0 (Win32)
AMD K6-2 @ 450MHz (MS-DOS)	13200	Borland C++ 3.1 (MS-DOS)
AMD K6-2 @ 450MHz (Windows 98)	4000	MS Visual C++ 6.0 (Win32)
FPSLIC coprocessor @ 12.5MHz	(*) 5840/17040	Atmel System Designer / IAR

(*) best/worst case depending on the number of hardware modules that have to be reconfigured to compute the trigonometric function.

Concerning the area saving, this reconfigurable strategy offers advantages due to the high similarity between the rotation and vectoring modes of the algorithm. Although the design is feasible without flexible hardware, in a DR-FPGA the interface for accessing all the routing and logic reconfigurable resources is already available and it is not necessary to design neither some control lines (*wr/rd*, *en*…) nor the FSM for controlling them. This simplifies the design, the routing and, in turn, minimizes the critical path.

Table 3. Experimental results of the reconfigurable coprocessor

Hw Resources	Data	Execution Time (ns)	Data	Compute Error	Data
Flip-Flops	125	Data-Control I/O	3920	Sine	$\leq 10^{-6}$
Gates	1177	Reconf. Kmul	960	Cosine	$\leq 10^{-6}$
IO cells	33	Reconf. Sign	640	Arctangent	$\leq 10^{-6}$
Total logic cells	1423	Reconf. Dmux	10560	Magnitude	$< 2 \cdot 10^{-6}$

5 Conclusions

This work describes the design of a dynamically reconfigurable trigonometric coprocessor based on the CORDIC algorithm and mapped on an FPSLIC device. While a low-cost MCU runs the program flow, a hardware coprocessor takes charge of the computing. The MCU transfers the operands to the FPGA and reconfigures specific flexible blocks of the coprocessor at run-time. In parallel, the FPGA computes the trigonometric function requested by the software application and finishes by transferring the result to the MCU. The concept reached is a single-chip cost-effective embedded system that only just running at a low frequency processes sine, cosine, arctangent or 2-D magnitude (square root) operations at rates comparable to powerful PC platforms.

Bibliography

[Vo59] Volder, J. E.: "The CORDIC Trigonometric Computing Technique", IRE Trans. on Electronic Computers, 1959; vol.EC-8, no.3, pp.330-334.

[Wa71] Walther, J. S.: "A Unified Algorithm for Elementary Functions", Proc. 38th Spring Joint Computer Conference, 1971; pp.379-385.

[VT99] Vladimirova, T.; Tiggeler, H.: "FPGA Implementation of Sine and Cosine Generators Using the CORDIC Algorithm", MAPLD'99, 1999.

[Ti91] Timmermann, D. et al.: "A Programmable CORDIC Chip for Digital Signal Processing Applications", IEEE Journal of Solid-State Circuits, 1991; vol.26, no.9, pp.1317-1321.

[An98] Andraka, R.: "A Survey of CORDIC Algorithms for FPGA Based Computers", Proc. 6th International Symposium on FPGAs, Monterey, USA, 1998; pp.191-200.

[At01] Atmel Corp.: "AT94K Series Cache Logic® (Mode 4) Configuration", 2001.

[Hu92] Hu, Y. H.: "CORDIC-Based VLSI Architectures for Digital Signal Processing", IEEE Signal Processing Magazine, 1992; pp.16-35.

[VM02] Vadlamani, S.; Mahmoud, W.: "Comparison of CORDIC Algorithm Implementations on FPGA families", Proc. 34th Southeastern Symposium on System Theory, 2002.

[Li99] Ligon, W. B. et al.: "Implementation and Analysis of Numerical Components for Reconfigurable Computing". Proc. IEEE Aerospace Conference, 1999; vol.2.