# A CMOS Floating-Point Vector-Arithmetic Unit

D. Timmermann, B. Rix, H. Hahn and B. J. Hosticka

***Abstract*—This work describes a floating-point arithmetic unit based on the CORDIC algorithm. The unit computes a full set of high level arithmetic and elementary functions: multiplication, division, (co)sine, hyperbolic (co)sine, square root, natural logarithm, inverse (hyperbolic) tangent, vector norm, and phase. The chip has been integrated in 1.6 $\mu$m double-metal $n$-well CMOS technology and achieves a normalized peak performance of 220 MFLOPS.**

## I. Introduction

In this paper we present a floating-point arithmetic unit that is based on the COordinate Rotation DIgital Computer (CORDIC) algorithm [1], [2] and that exhibits an exceptional functionality. While the CORDIC algorithm enjoys an increasing popularity when realizing trigonometric or hyperbolic functions, its implementations exist either only as firmware [3] or do not realize the entire algorithm [4], [5], [20]. This approach is often adopted when the full set of arithmetic and elementary functions offered by CORDIC is not acquired, e.g., for dedicated special-purpose applications [6]–[8]. This contribution presents a CORDIC chip which implements all CORDIC functions available but in contrast with a previously reported work, it realizes an IEEE-754 floating point pipeline instead of a fixed-point recursive approach [9], thus achieving significantly higher functional throughput.

## II. CORDIC Algorithm

Iterative vector rotations form the mathematical basis for the computation of the CORDIC algorithm [1], [2]. Since the resulting iteration sequences are highly regular and their execution can be easily pipelined, they are amenable to monolithic integration. In addition, the CORDIC algorithm achieves unprecedented functionality because the coordinate systems for the iterations can be readily changed.

The arithmetic unit presented in this work realizes CORDIC iterations given by:

$$\begin{aligned} x_{i+1} &= x_i - m\sigma_i 2^{-S(m,i)} y_i \\ y_{i+1} &= y_i + \sigma_i 2^{-S(m,i)} x_i \\ z_{i+1} &= z_i - \sigma_i \alpha_{m,i}, \end{aligned} \tag{1}$$

Manuscript received November 9, 1992; revised January 31, 1994. This work was supported in part by the German Research Council (Deutsche Forschungsgemeinschaft), Bonn.

D. Timmermann was with Fraunhofer Institute of Microelectronic Circuits and Systems, 4100 Duisburg 1, Germany. He is now with Department Meschede, University of Paderborn, D-59872 Meschede, Germany.

H. Hahn and B. J. Hosticka are with Fraunhofer Institute of Microelectronic Circuits and Systems, 4100 Duisburg 1, Germany.

B. Rix is with the Department of Electrical Engineering, University of Duisburg, 4100 Duisburg 1, Germany.

IEEE Log Number 9401621.

TABLE I
CORDIC–Functions ($k_{-1,0,1} \equiv$ scaling factors)

| | $z_n \rightarrow 0$ (Rotation) | $y_n$ (Vectoring) |
|---|---|---|
| $m = -1$ hyperbolic | $x_n = k_{-1}(x_0 \cosh(z_0) + y_0 \sinh(z_0))$<br>$y_n = k_{-1}(x_0 \cosh(z_0) + y_0 \sinh(z_0))$ | $x_n = k_{-1}\sqrt{x_0^2 - y_0^2}$<br>$z_n = z_0 + \tanh^{-1}(y_0/x_0)$ |
| $m = 0$ linear | $x_n = x_0$<br>$y_n = x_0 z_0 + y_0$ | $x_n = x_0$<br>$z_n = z_0 + y_0/x_0$ |
| $m = 1$ circular | $x_n = k_1(x_0 \cos(z_0) - y_0 \sin(z_0))$<br>$y_n = k_1(y_0 \cos(z_0) + x_0 \sin(z_0))$ | $x_n = k_1\sqrt{x_0^2 + y_0^2}$<br>$z_n = z_0 + \tan^{-1}(y_0/x_0)$ |

where,

| | |
|---|---|
| $m$ | coordinate system, |
| $\sigma_i$ | rotation direction, |
| $S(m,i)$ | shift sequence |
| $\alpha_{m,i}$ | incremental rotation angle, |
| $i$ | $i$th iteration. |

The parameter $m$ can be chosen as 1, 0, or $-1$ and the corresponding vector movements can be interpreted as rotations on a circle, a straight line, or a hyperbolic, respectively.

The convergence properties of the algorithm depend on the predetermined sequence of $S(m,i)$, which defines the angle:

$$\begin{aligned} \alpha_{m,i} &= \frac{1}{\sqrt{m}} \tan^{-1}(\sqrt{m}\, 2^{-S(m,i)}) \\ &= \begin{cases} \tan^{-1}(2^{-S(m,i)}) & \text{for } m = 1 \\ 2^{-S(m,i)} & \text{for } m = 0 \\ \tanh^{-1}(2^{-S(m,i)}) & \text{for } m = -1. \end{cases} \end{aligned} \tag{2}$$

During the iterations, either $z$ or $y$ are forced to zero by choosing

$$\sigma_i = \text{sign}(z_i) \text{ or } \sigma_i = -\text{sign}(x_i)\text{sign}(y_i), \tag{3}$$

respectively. By specifying the iteration goal and appropriate coordinate system, we can program the unit to obtain the desired functions (see Table I). $x_0$, $y_0$, and $z_0$ are the initial values and $k_m$ represents the scaling factor. As generally $k_m \neq 1$, this spurious factor must be compensated for.

There are several possible solutions to this problem. One common method is to increase the iteration count by repeating some of the iterations in such a manner so that the deviation from $k_m = 1$ can be adjusted by a binary shift [10]. Another approach relies on using double shifts, i.e., an additional shift is used in the CORDIC iteration besides the original one (see CORDIC equations above). The increase in iteration count is then much lower when compared with the first method [5]. We adopted a hybrid solution that relies on repeating as few iterations as possible and decomposing $1/k_m$ in factors $(1 \pm 2^{-j})$ [19].

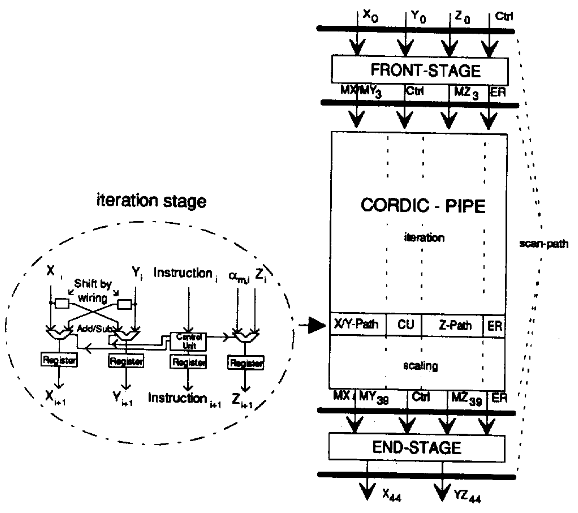

Fig. 1. Block diagram of the CORDIC pipeline.

## III. Functional Overview

The block diagram of the CORDIC-based arithmetic unit implemented in this work is shown in Fig. 1. The input data stream is fed into three input ports $x, y$, and $z$, and after a fixed pipeline latency of 44 cycles the results appear at the output ports $x$ and $yz$ (note: $M$ denotes mantissa and $E$ exponent in Fig. 1). It directly computes the following elementary functions: multiplication, division, (co)sine, hyperbolic (co)sine, square root, natural logarithm, inverse (hyperbolic) tangent, vector norm, and phase. As an example, a full coordinate transformation is performed using one instruction only (e.g., instruction ROT, cf. Table II). Further functions can be obtained when presetting some input values according to Table III.

The unit employs an FXP pipeline that implements hardwired add-and-shift sequences and yields a much higher throughput than recursive implementations [9]. The FXP pipeline carries out the CORDIC iterations and subsequent scaling operation to compensate for the scaling factor $k_m$. Each iteration pipeline stage executes one iteration according to (1). The FXP-pipeline (29 stages) is preceded by a front-stage (see Fig. 2) which accepts the input data in IEEE-754 single-precision FLP format and carries out the conversion into an internal FXP format (Fig. 3). This format was developed to suit the modified floating-point CORDIC algorithm which is employed in the pipeline [11]. Thus any further mantissa alignments and exponent computations inside the inner pipeline can be omitted. The conversion is based on the same method as found in FLP adders: 1) determining a common exponent for both arguments, called a reference exponent, 2) mantissa shifting depending on the difference between the arguments' exponent and the reference exponent, 3) processing of the shifted mantissas, 4) normalization of the mantissas and exponent alignment. However, in our case it is more complicated due to the three arguments of the CORDIC algorithm.

TABLE II
Partial List of Available Functions

| Mnemonic | Hex | Description |
|---|---|---|
| DIVADD | 04 | $x = x_{\text{in}}, yz = z_{\text{in}} + y_{\text{in}}/x_{\text{in}}$ |
| DIVSUB | 06 | $x = x_{\text{in}}, yz = z_{\text{in}} - y_{\text{in}}/x_{\text{in}}$ |
| MULADD | 0C | $x = x_{\text{in}}, yz = y_{\text{in}} + z_{\text{in}} * x_{\text{in}}$ |
| MULSUB | 0E | $x = x_{\text{in}}, yz = y_{\text{in}} - z_{\text{in}} * x_{\text{in}}$ |
| HVECT | 05 | $x = ((x_{\text{in}})^2 - (y_{\text{in}})^2)^{0.5}$, $yz = z_{\text{in}} + \tanh^{-1}(y_{\text{in}}/x_{\text{in}})$ |
| HVECTM | 08 | $x = ((x_{\text{in}})^2 - (y_{\text{in}})^2)^{0.5}$, $yz = z_{\text{in}} - \tanh^{-1}(y_{\text{in}}/x_{\text{in}})$ |
| HROT | 0D | $x = x_{\text{in}} * \cosh(z_{\text{in}}) + y_{\text{in}} * \sinh(z_{\text{in}})$, $yz = y_{\text{in}} * \cosh(z_{\text{in}}) + x_{\text{in}} * \sinh(z_{\text{in}})$ |
| HROTM | 0F | $x = x_{\text{in}} * \cosh(z_{\text{in}}) - y_{\text{in}} * \sinh(z_{\text{in}})$, $yz = y_{\text{in}} * \cosh(z_{\text{in}}) - x_{\text{in}} * \sinh(z_{\text{in}})$ |
| VECT | 15 | $x = ((x_{\text{in}})^2 + (y_{\text{in}})^2)^{0.5}$, $yz = z_{\text{in}} + \tan^{-1}(y_{\text{in}}/x_{\text{in}})$ |
| VECTM | 18 | $x = ((x_{\text{in}})^2 + (y_{\text{in}})^2)^{0.5}$, $yz = z_{\text{in}} - \tan^{-1}(y_{\text{in}}/x_{\text{in}})$ |
| ROT | 1D | $x = x_{\text{in}} * \cos(z_{\text{in}}) - y_{\text{in}} * \sin(z_{\text{in}})$, $yz = y_{\text{in}} * \cos(z_{\text{in}}) + x_{\text{in}} * \sin(z_{\text{in}})$ |
| ROTM | 1F | $x = x_{\text{in}} * \cos(z_{\text{in}}) + y_{\text{in}} * \sin(z_{\text{in}})$, $yz = y_{\text{in}} * \cos(z_{\text{in}}) - x_{\text{in}} * \sin(z_{\text{in}})$ |
| IHROT | 09 | $x = x_{\text{in}} * \cosh(z_{\text{in}}) + y_{\text{in}} * \sinh(z_{\text{in}})$, $yz = y_{\text{in}} * \cosh(z_{\text{in}}) + x_{\text{in}} * \sinh(z_{\text{in}})\|^1$ |
| IHROTM | 0B | $x = x_{\text{in}} * \cosh(z_{\text{in}}) - y_{\text{in}} * \sinh(z_{\text{in}})$, $yz = y_{\text{in}} * \cosh(z_{\text{in}}) - x_{\text{in}} * \sinh(z_{\text{in}})\|^1$ |
| IROT | 19 | $x = x_{\text{in}} * \cos(z_{\text{in}}) - y_{\text{in}} * \sin(z_{\text{in}})$, $yz = y_{\text{in}} * \cos(z_{\text{in}}) + x_{\text{in}} * \sin(z_{\text{in}})\|^1$ |
| IROTM | 1B | $x = x_{\text{in}} * \cos(z_{\text{in}}) + y_{\text{in}} * \sin(z_{\text{in}})$, $yz = y_{\text{in}} * \cos(z_{\text{in}}) - x_{\text{in}} * \sin(z_{\text{in}})\|^1$ |

$|^1$ rotation angle is obtained from preceding operation

TABLE III
Additional Functions Available by Presetting Specific Input Values

| Instruction | Input | | | Output | |
|---|---|---|---|---|---|
| | $x$ | $y$ | $z$ | $x$ | $yz$ |
| ROT | $x$ | 0 | $z$ | $x\cos(z)$ | $x \sin(z)$ |
| HROT | $x$ | 0 | $z$ | $x\cosh(z)$ | $x \sinh(z)$ |
| HVECT | $x+1$ | $x-1$ | 0 | $2\sqrt{x}$ | $\frac{1}{2}\ln(x)$ |
| HROT | $x$ | $x$ | $z$ | $xe^z$ | $xe^z$ |
| HVECT | $x$ | 1 | 0 | $\sqrt{x^2-1}$ | $\coth^{-1}(x)$ |
| HVECT | $x+\frac{1}{4}$ | $x-\frac{1}{4}$ | 0 | $\sqrt{x}$ | $\ln(\frac{1}{4}/x)$ |
| HVECTM | 1 | $y$ | $\pi/2$ | $\sqrt{y^2+1}$ | $\cot^{-1}(y)$ |
| HVECT | $x+y$ | $y-x$ | 0 | $2\sqrt{x \cdot y}$ | $\frac{1}{2}\ln(y/x)$ |

Two's complement of all three mantissas is computed (TCOMPL) and a bias of 127 is subtracted (BIAS) to yield a reference exponent. This is generated according to Table IV and passed through the FXP pipeline at the right hand side. Several interim results determine the amount of mantissa shifting. The FXP pipeline also incorporates at its end eight scaling stages that adjust the spurious factor $k_m$ using successive multiplications by factors of $(1 \pm 2^{-j})$, as mentioned above. The end-stage (Fig. 4) follows the FXP pipeline and performs mantissa normalization and rounding and also exponent computation so that the output data conform to the IEEE floating-point format. For this reason, the internal data have to be converted into sign/magnitude format (TCOMPL). The

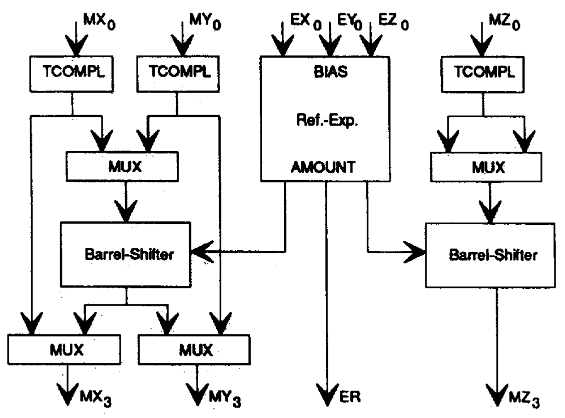

Fig. 2. Front-stage.

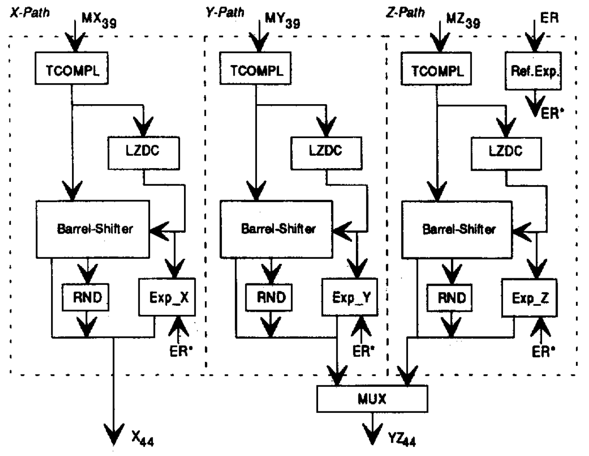

Fig. 4. End-stage.

TABLE IV
DETERMINATION OF THE REFERENCE EXPONENT

| | Vectoring | Rotation |
|---|---|---|
| Circular | ER = max ($EY_0$,$EX_0$) | ER = max ($EY_0$,$EX_0$) |
| Linear | ER = max ($EZ_0$,$EY_0$-$EX_0$) | ER = max ($EY_0$,$EX_0$ + $EZ_0$) |
| Hyperbolic | ER = max ($EY_0$,$EX_0$) | ER = max ($EY_0$,$EX_0$ |

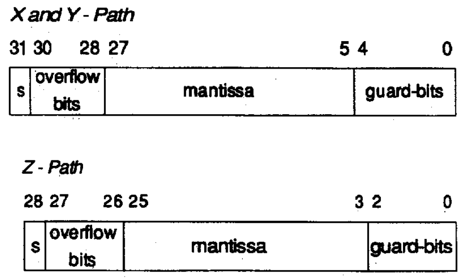

Fig. 3. Internal data format.

block labeled LZDC inspects the leading zeros of the mantissas and controls the barrel-shifter used for normalization. Using these results and the reference exponent calculated in the front-stage, the three output exponents are computed. Sticky bit rounding is employed for the mantissa rounding which causes no ripple and limits the rounding error to max. 1 LSB.

Six bit instruction controls the data flow, the iteration goal (rotation or vectoring), and the coordinate system selection. Besides its floating-point capability, the chip also accepts input data in a 24-bit fixed-point format. In addition, it offers a constant angle coding option for those algorithms that require subsequent vector rotations by the identical vector phase. This feature speeds up the computation in such cases since it dispenses with repeated calculations of $z$.

## IV. DESIGN

The implementation of the CORDIC equations (see (1)) requires three data paths. Due to the characteristics of the functions computed and the add/shift property of the algorithm we have to provide additional overflow (integer) and guard bits in the internal data representation. As we are employing the IEEE-754 single precision format two internal 32b data paths (1 sign, 3 overflow, 23 fraction, and 5 guard-bits) are used for $x$ and $y$ computation, and a 29b data path (1 sign, 2 overflow, 23 fraction, and 3 guard-bits) is required for realizing of the $z$ calculation. The fraction pipeline consists in total of 29 iteration and 8 scaling factor compensation stages, each also containing additional registers for the 10b reference exponent. Hence, the internal data path measures 103b × 37b in total which dictates mandatory use of chip area saving methods.

In order to execute the CORDIC equations, a repeated addition/subtraction of the $x$-mantissa and the $y$-mantissa shifted by $S(m,i)$ and vice versa is required (see (1) and Fig. 1). Direct mapping of these operations into silicon would require interconnect criss-crossing of both data paths in each iteration stage. These crossings would require a minimum routing length of 32b (for $S(m,i) = 1$) and would yield in general a highly irregular layout. To eliminate such crossings, the $x$ and $y$ paths are bitwise interleaved in layout thus reducing the minimum routing length required for shifting to $2b$ (1b for $x$ and 1b for $y$).

Spacing and routing of the pipeline depend on the actual CORDIC sequence $S(m,i)$ as given by Table V. This has been found using computer simulations which optimized the number of iterations and the resulting error distribution [19]. It is used in an area optimizing CORDIC FXP pipeline module generator. Fig. 5 depicts the layout and routing scheme used in the generator, together with the bitwise interleaved paths and shift routing (MCA stands for Manchester carry chain adder). Each bit cell contains two identical outputs on the left and the right edge of the cell. Together with a) an asymmetric arrangement of the shifted inputs (see dashed incision of Fig. 5) and b) routing the (longer) shifted $x$ input in second metal layer (metal2) and the shifted $y$ input in 1st metal layer (metal1) these measures save one routing track. Therefore,

TABLE V
SHIFT AND SCALING PARAMETERS [19]

| $i$ | $S(m,i)$ | $i$ | $S(m,i)$ | $i$ | $S(m,i)$ | $i$ | $S(m,i)$ | $i$ | $S(m,i)$ | $i$ | $S(m,i)$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 6 | 3 | 11 | 7 | 16 | 12 | 21 | 16 | 26 | 21 |
| 2 | 2 | 7 | 4 | 12 | 8 | 17 | 13 | 22 | 17 | 27 | 22 |
| 3 | 2 | 8 | 5 | 13 | 9 | 18 | 13 | 23 | 18 | 28 | 23 |
| 4 | 2 | 9 | 6 | 14 | 10 | 19 | 14 | 24 | 19 | 29 | 24 |
| 5 | 2 | 10 | 6 | 15 | 11 | 20 | 15 | 25 | 20 | – | – |

| $m$ | $a(m,1)$ | $a(m,2)$ | $a(m,3)$ | $a(m,4)$ | $a(m,5)$ | $a(m,6)$ | $a(m,7)$ | $a(m,8)$ |
|---|---|---|---|---|---|---|---|---|
| 1 | -2 | 4 | -5 | 6 | 0 | 17 | -20 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| -1 | 2 | 4 | 0 | 6 | -6 | 0 | -20 | -21 |

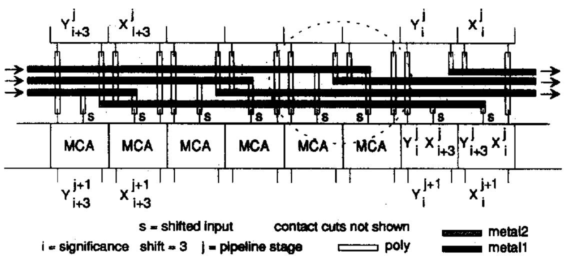

Fig. 5. Bitwise interleaved $x/y$ paths and shift routing.

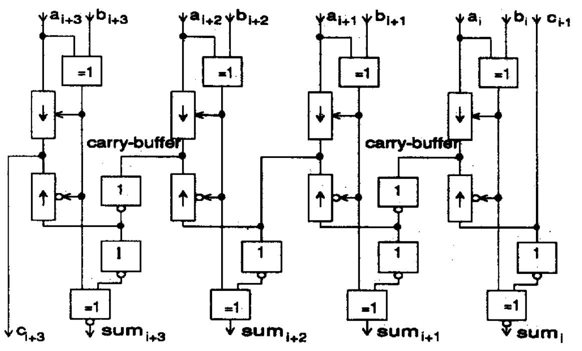

Fig. 6. Four bit-slice Manchester carry chain.

we need at most 16 routing tracks for the worst case and the entire shift wiring occupies only 18% of the whole FXP pipeline area.

The generator places and routes the whole internal FXP pipeline in 25 CPU seconds (VAX 11-8550) and is fully flexible in terms of design rules and parameters of the algorithm. It computes the properly rounded binary representation of the rotation angles $\alpha_{m,i}$ and programs one input of the $z$ data path adder by connecting the inputs of a 3-to-1 multiplexer to $V_{\mathrm{dd}}$ or $V_{\mathrm{ss}}$. The pipeline, consisting mainly of adders and registers, occupies the main part of the chip area. Investigations have shown that a ripple carry adder, consisting of Manchester carry chain adders, results in the best area/speed trade-off for our purposes. To get a fast carry path we implemented 4bit-slices (Fig. 6) with carry buffers after 2bit. The sign-extension of the shifted MSB's severely increases its capacitive loading so we have provided an optimized MSB-buffer which is hidden beneath the power supply wiring. The generator has been developed in such a way that fast redundant addition schemes, such as redundant binary [6] or carry save, can also be implemented if desired. According to our estimates, employing these adders would more then double our current chip area. The remaining parts of the chip were laid out manually.

The chip also accepts data in FXP format. The input FXP format is then one sign and 23 fraction bits. As we multiplex the $y$ and $z$ path onto one output port the output FXP format depends on whether the last pipeline stage has executed a rotation or vectoring instruction (refer to Table I). In rotation mode the format for both the $x$ and $yz$ output data is one sign, 3 integer, and 20 fraction bits. The same is valid for the $x$ output in vectoring mode. The $yz$ output format for vectoring is one sign, two integer, and 21 fraction bits. In all cases the input and output FXP numbers are represented in Sign-Magnitude format.

As mentioned in Section III, the chip also provides the option of constant angle coding. In general, the implementation of the CORDIC equations requires the knowledge of $\sigma_i$ (see (1)). This is accomplished in a control unit, located between the $xy$ and the $z$ data path, which senses the sign of the three data paths and computes $\sigma_i$ depending on the current instruction (3). In our case, we provided a special mode which can significantly speed up some matrix algorithms for linear equation solving, i.e., QR decomposition and Givens rotation. In these applications we start with computing the phase of a vector and then rotate subsequent vectors (i.e., matrix elements) by this angle. The special INI(tialize)-instructions which are marked with $|^1$ in Table II use the angle of a preceding phase calculation, thus avoiding a whole pipeline latency of 44 cycles. Assume we have a continuous data stream of $(x, y)$ vector coordinates $(a_1, b_1)$, $(a_2, b_2)$, $(a_3, b_3)$,$\ldots$ and we want to rotate $(a_2, b_2)$, $(a_3, b_3)$,$\ldots$ by the phase of $(a_1, b_1)$, e.g., $\tan^{-1}(b_1/a_1)$. Then the input instruction sequence would look: VECT, IROT, IROT, .... This is easily implemented in the control unit by retaining the value of $\sigma_i$ of the preceding VECT instruction, a method which has been first reported in [5].

At the end of the FXP pipeline the output data of the iteration part of the FXP pipeline are appropriately scaled to compensate for the scale factor $k_m$ given by Table I. This is accomplished by successive multiplication of the $x$ and $y$ data path by $(1 + \mathrm{sign}(a(m,i))2^{-|a(m,i)|})$, as defined by Table V. This results in the desired vector contraction $(m = 1, 1/k_1 = 0.784039965)$ or extension $(m = -1, 1/k_{-1} = 1.327798882)$. This requires again add-and-shift operation which compared with the original CORDIC equations demonstrates that we only have to change the wiring procedure and the control unit to achieve the desired effect, and do not have to alter the data paths. The different shifts (depending on $m$) for each scaling stage are implemented by adding a 3-to-1 multiplexer, controlled by $m$, and programming its input.

The front- and end-stage are connected to the pipeline via scan-path registers. Because of the very high regularity of the pipeline stages there is no need for a full scale internal scan-path and the increase in chip area would be too costly if used.

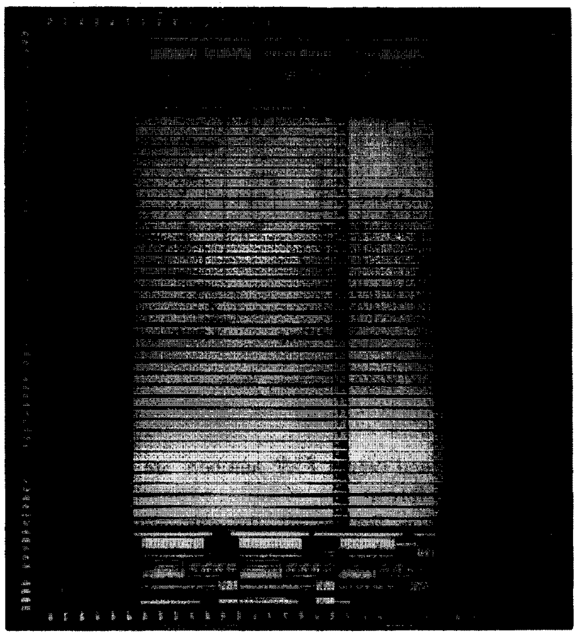
Fig. 7. Chip photomicrograph.

TABLE VI
TECHNICAL DATA

| | |
|---|---|
| Process | Double-metal CMOS |
| Design Rule | 1.6 $\mu$m |
| Die Size | 13.3 × 14.2 mm |
| Active Area | 8.2 × 13.4 mm |
| Transistors | 210,000 |
| Package | 280 pin PGA |
| # pipe stages | (3+37+4)=44 |
| Max. clock | 10 MHz |
| Data format: | |
| external | 24b mant., 8b exp. |
| | (IEEE–754 single prec.) |
| | or |
| | 24b fixed point |
| internal | 32b mant., 10b exp. |
| Instruction | 6 bit |

## V. Chip Data

The CORDIC-based arithmetic unit has been integrated in an 1.6-$\mu$m double-metal $n$-well CMOS process. The fabricated device contains 210,000 transistors on an active die area of 110 mm$^2$ (Table III). The chip operates at 5 V power supply voltage and 10 MHz clock and reaches a normalized peak performance of 220 MFLOP's. The chip is packaged in a 280 pin PGA and already first silicon was fully operational (Fig. 7).

## VI. Summary

In this contribution we have presented a CORDIC-based arithmetic unit. Unlike other pipeline implementations, the presented device realizes all available CORDIC functions. It offers an extremely high computational flexibility (see Table I). It adheres to IEEE-754 single precision FLP data format and realizes the full set of available CORDIC functions. This yields a unique computational flexibility at high functional throughput rates. To achieve the desired functionality in the underlying technology, the inner pipeline and the shift wiring have been carefully area optimized. Further chip area reductions are possible, when employing the techniques recently reported in [18]. The chip can be used for numerous applications, ranging from matrix processing [10] to computer graphics [12] and digital signal processing [13], [14]. In particular, hardware implementations of fixed and adaptive lattice filters [15], quadrature amplitude modulation [15], and image transforms [16], [17] can be greatly simplified when using the CORDIC algorithm on a chip.

## Acknowledgment

The authors wish to thank the staff of Technology Department of the Fraunhofer Institute of Microelectronic Circuits and Systems for wafer fabrication and technology support. Further thanks go to Michael Leineweber for measurement support.

## References

[1] J. E. Volder, "The CORDIC trigonometric computing technique," *IRE Trans. Electron. Comput.*, vol. 8, no. 3, pp. 330–334, Sept. 1959.
[2] J. S. Walter, "A unified algorithm for elementary functions, " *Proc. of Spring Joint Comput. Conf.*, (SJCC) 1971, *AFIPS Proc.*, vol. 38, pp. 379–385.
[3] R. Nave, "Implementation of transcendental functions on a numerics processor," *Microprocessing and Microprogramming*, vol. 11, pp. 221–225, 1983.
[4] G. L. Haviland and A. A. Tuszynski, "A CORDIC arithmetic processor chip," *IEEE Trans. Comput.*, vol. 29, no. 2, pp. 68–79, February 1980.
[5] E. F. Deprettere, P. Dewilde, and F. Udo, "Pipelined CORDIC architectures for fast VLSI filtering and array processing," *ICASSP'84*, pp. 41 A6.1–6.4, San Diego, March 1984.
[6] H. Yoshimura, T. Nakanishi, and H. Tamauchi, "A 50 MHz CMOS geometrical mapping processor," Dig. of Tech. Papers, *IEEE Int. Solid-State Circuits Conf. (ISSCC)*, Feb. 1988, San Francisco, CA, pp. 162–163.
[7] Advance Information TMC 1330 CMOS Coordinate Transformer, TRW LSI Products, July 1989.
[8] Preliminary Information PDSP16340 Polar To Cartesian Converter, Plessey Semiconductors, April 1990.
[9] D. Timmermann, H. Hahn, B. J. Hosticka and G. Schmidt, "A programmable CORDIC chip for digital signal processing applications," *IEEE J. Solid-State Circuits*, vol. 26, no. 9, pp. 1317–1321, Sept. 1991.
[10] H. M. Ahmed, "Signal processing algorithms and architectures," Ph.D. thesis, Department of Electrical Eng., Stanford University, Dec. 1981.
[11] B. Yang, "Investigations of a 32 bit floating-point CORDIC processor" (in German), diploma thesis, Dept. of Electrical Eng., Ruhr-University Bochum, July 1986.
[12] D. Timmermann, J. F. Böhme, H. Hahn, B. J. Hosticka, G. Schmidt and G. Zimmer, "CORDIC-Coprozessor für schnelle 3-D Grafik," NTG-Fachberichte 96, Mikroelektronik für die Informationstechnik-Vermittlung, Übertragung und Verarbeitung, pp. 81–86, Sept. 1986 (in German).
[13] H. Hahn, J. Büddefeld, B. J. Hosticka and U. Kleine, "CORDIC realization of power-wave digital filters," *Proc. ECCTD-85*, pp. 507–510, Prague, Sept. 1985.
[14] R. Lerch, J. F. Böhme, H. Hahn, B. J. Hosticka, G. Schmidt, D. Timmermann and G. Zimmer, "CORDIC realization of a DFT processor," *Proc. EUSIPCO-86*, pp. 1319–1322, The Hague, Sept. 1986.
[15] H. Hahn, B. J. Hosticka and D. Timmermann, "An alternative signal processor arithmetic for a modified implementation of an normalized adaptive channel equalizer," *IEE Proc.-F, Radar and Signal Processing*, vol. 139, no. 1, pp. 36–42, Feb. 1992.
[16] D. Timmermann, H. Hahn and B. J. Hosticka, "Hough transform using CORDIC method," *Electron. Lett.*, vol. 25, no. 3, pp. 205–206, Feb. 1989.
[17] A. Teuner and B. J. Hosticka, "An adaptive filter for two-dimensional Gabor transformation and its implementation," *IEE Proc. on Image Processing*, accepted for publication (Feb. 1993).

[18] D. Timmermann and I. Sundsbφ, "Area and latency efficient CORDIC architectures," *IEEE Int. Symp. on Circuits and Systems (ISCAS92)*, pp. 1093–1096, San Diego, May 1992.
[19] D. König and J. F. Böhme, "Optimizing the CORDIC algorithm for processors with pipeline architecture," *Proc. EUSIPCO-90*, pp. 1391–1394, Barcelona, Sept. 1990.
[20] E. F. Deprettere, A. A. J. de Lange and P. Dewilde, "The synthesis and implementation of signal processing applications specific VLSI CORDIC arrays," *Proc. ISCAS-90*, pp. 974–977, New Orleans, LA, May 1990.