Pipelined Implementation of High Radix Adaptive CORDIC as a coprocessor

Saharsh Samir Oza*, Ankit Parag Shah*, Tarun Thokala* and Sumam David*
*Department of Electronics and Communication Engineering
National Institute of Technology Karnataka, Surathkal, India
Email: ankit.p.1992@ieee.org

***Abstract*—The Coordinate Rotational Digital Computer (CORDIC) algorithm allows computation of trigonometric, hyperbolic, natural log and square root functions. This iterative algorithm uses only shift and add operations to converge. Multiple fixed radix variants of the algorithm have been implemented on hardware. These have demonstrated faster convergence at the expense of reduced accuracy. High radix adaptive variants of CORDIC also exist in literature. These allow for faster convergence at the expense of hardware multipliers in the datapath without compromising on the accuracy of the results. This paper proposes a 12 stage deep pipeline architecture to implement a high radix adaptive CORDIC algorithm. It employs floating point multipliers in place of the conventional shift and add architecture of fixed radix CORDIC. This design has been synthesised on a FPGA board to act as a coprocessor. The paper also studies the power, latency and accuracy of this implementation.**

***Index Terms*—CORDIC, iterative, adaptive, pipelined, coprocessor**

I. Introduction

CORDIC (Coordinate Rotation Digital Computer) [1] is an iterative algorithm that computes a plethora of mathematical operations such as natural log, division, square root, trigonometric and hyperbolic transforms. The algorithm has been widely adopted for a diverse set of applications ranging from signal and image processing [2], [3], communication hardware [4], [5], robotics [6] and 3D computer graphics [7] to name a few. All these applications leverage the algorithms simple implementation of a shift and add architecture to obtain accurate results. A variety of dedicated hardware datapath designs have been developed for this implementation as can be seen in [8], [9] and [10]. The parallel pipelined architecture has demonstrated the highest throughput. The simplicity of its design is attributed to a constant scaling factor which fixes the number of iterations in the hardware implementation. This however introduces the key drawback of the CORDIC algorithm; n bit precision in the output requires n iterations for it to converge [9].

Numerous modifications have been proposed in the form of fixed high radix algorithms to reduce the number of iterations. Higher radix CORDIC aims at increasing the granularity of the micro-rotations, hence reducing the number of non convergent micro-rotations. Radix 4 variants have been developed and implemented in hardware as in [11], [12] and [13]. These have proven to give n bit precision in $n/2$ iterations, a reduction of half in the number of micro-rotations. However, this faster convergence comes at the expense of reduced accuracy.

Angle recoding [14], [15], [16] is another approach that tackles the problem of latency in CORDIC. It skips all non-convergent iterations hence reducing the total number of iterations for convergence. A greedy algorithm is used to select only those micro-rotations that will result in fast convergence. The computation involved in the selection of the micro rotations is expensive and hence the algorithm works best with inputs that are known *apriori*. This solves a useful problem in a wide range of DSP applications where the inputs of common transforms like Fourier and DCT can be predicted. However, in applications where the input cannot be predicted, the selection of micro-rotations has to be performed dynamically which can nullify gains made by fewer iterations in terms of number of cycles. To overcome this input constraint, a parallel angle recoding (PAR) [17] technique has been introduced. This proposes a variant of the angle recoding technique where the selection of the micro rotations is done *apriori*. Every input is compared against a set of predetermined boundary conditions to predict the micro rotations that would result in the fastest convergence.

A hybrid approach has also been studied that splits the CORDIC iterations into coarse and fine micro rotations. The finer rotations approximate $tan^{-1}(2^{-i})$ to 2^{-i}. This reduces latency of the lookup and the size of the lookup table.

All the algorithms discussed above retain the shift and add architecture of CORDIC. This restricts them to using a fixed scaling factor which fixes the set and range of micro rotations that the algorithm can make. While these micro rotations can be skipped using angle recoding or made more granular by using a higher radix, the main limitation is not being able to adaptively decide the micro rotation that would eliminate all non convergent iterations and yet retain the accuracy of CORDIC.

978-1-4673-7309-8/15/$31.00 ©2015 IEEE

High radix adaptive CORDIC algorithm (HCORDIC) proposed in [18] by El Guibaly et. al. aims at exploring the departure from shift and add architecture to the use of hardware multipliers for increased parallelism and reduced iterations. Its inherent dependency on IEEE 754 floating point format results in better accuracy. The hardware multipliers allow for a non constant scaling factor and hence enable an adaptive selection of micro rotations. While this algorithm has proven to have faster convergence through fewer micro rotations theoretically, the challenge that remains is to identify if fewer iterations can translate to fewer clock cycles after the implementation of hardware multipliers in the execute datapath. Thus, we propose the following hardware design aiming to address this challenge.

The paper is organised as follows. The second section aims at revisiting the CORDIC algorithm and then presents the adaptive radix variant of the same. The section ends with results obtained through software validation of a convergence test of the two algorithms. The next section presents a quick recap of existing CORDIC datapaths followed by the hardware design of HCORDIC. This will be followed by an illustrative example demonstrating HCORDIC functionality on hardware. Power, timing and accuracy results obtained will be presented in the subsequent section.

II. HCORDIC Algorithm

A. CORDIC Algorithm

CORDIC computes its mathematical functions by moving vectors x and y over a geometric trajectory fixed by a mode, $m \in \{1, 0, -1\}$, where $m = 1, 0, -1$ corresponds to circular, linear and hyperbolic coordinate system respectively. The variable z represents the accumulated angle. The elegance of this algorithm lies in the approximation of the tangent by 2^{-i} which is implemented by a simple binary shift where i represents the iteration number. The tangent defines the angular increment the input vector makes along the trajectory. This approximation is multiplied by a weighting factor $\sigma_i \in \{-1, 1\}$ depending on clockwise or anticlockwise rotation of the vector. This is implemented by a simple sign flip to the hardware adder. The following are the CORDIC equations.

$$x_{i+1} = x_i + m\sigma_i 2^{-i} y_i \quad (1)$$

$$y_{i+1} = y_i - \sigma_i 2^{-i} x_i \quad (2)$$

$$z_{i+1} = \begin{cases} z_i + \sigma_i \tan^{-1}(2^{-i}) & if\, m = 1 \\ z_i + \sigma_i (2^{-i}) & if\, m = 0 \\ z_i + \sigma_i \tanh^{-1}(2^{-i}) & if\, m = -1 \end{cases} \quad (3)$$

Fixed radix variations of the algorithm have replaced this 2^{-i} with a higher radix multiplied by an appropriate range of weighting factors [11]. The rationale behind doing so is to have more control over the angular increment to enable faster convergence. All these variants only give a fixed range of values that the angular increments can take for a given iteration number.

B. HCORDIC Equations

This paper focuses on the implementation of the highly adaptive radix CORDIC presented by El Guibaly et. al. [18]. The algorithm presented in the paper replaces the simple 2^{-i} with an approximation that is not fixed by the iteration i but instead depends on the value of the variables at the i^{th} iteration. This adaptive approximation is then multiplied by an adaptively determined weighting factor. The algorithm promises faster convergence with the above modifications but scales up the hardware complexity by adding the necessity to compute the weighting factor, δ_i.

The proposed algorithm inherently assumes the use of IEEE 754 floating point format. The notation of x_e, y_e and z_e represent the exponent part of the number while x_s, y_s and z_s represent the mantissa of these variables approximated to the first s bits followed by padding 0's for x_s and z_s whereas a 1's padding for y_s. The value of s has been fixed to 4 for this implementation and n denotes the number of bits used to represent the mantissa of the floating point number. In order to realise a non-constant scaling factor, K_i, its value has to be updated in each iteration by a factor κ_i.

$$x_{i+1} = x_i + m y_i \delta_i \quad (4)$$

$$y_{i+1} = y_i - x_i \delta_i \quad (5)$$

$$z_{i+1} = z_i + \theta_i \quad (6)$$

$$K_{i+1} = K_i * \kappa_i \quad (7)$$

$$\kappa_i = \begin{cases} 1/cos\theta_i & m = 1 \\ 1 & m = 0 \\ 1/cosh(\theta_i) & m = -1 \end{cases} \quad (8)$$

Two of the three HCORDIC operations discussed in [18] are identified for implementation. The y-vectoring operation was implemented since x-vectoring computation of δ_i can be viewed as a subset of y-vectoring computation of δ_i. This was done with a view to achieve a reduction of hardware. The operations resemble the vectoring and rotation operation of CORDIC. Unlike CORDIC, the computation of the shift and it's subsequent angular increment differs between the two operations.

1) Vectoring Operation:

$$\delta_i = \begin{cases} \text{sign}[(y_i)_s/(x_i)_s] & |y_i| \geq |x_i|, m \neq 0 \\ [(y_i)_s/(x_i)_s] * 2^{y_e - x_e} & |y_i| < |x_i|, m \neq 0 \\ [(y_i)_s/(x_i)_s] * 2^{y_e - x_e} & m = 0 \end{cases} \quad (9)$$

$$\theta_i = \begin{cases} \tan^{-1}\delta_i & m = 1 \\ \delta_i & m = 0 \\ \tanh^{-1}\delta_i & m = -1 \end{cases} \quad (10)$$

2) Rotation Operation:

$$\theta_i = \begin{cases} -1 & z_e > 0 \\ -z_s 2^{z_e} & -n/2 \le z_e \le 0 \\ -z_i & z_e < -n/2 \end{cases} \quad (11)$$

$$\delta_i = \begin{cases} \tan\theta_i & m = 1 \\ \tanh\theta_i & m = -1 \end{cases} \quad (12)$$

C. Geometric Visualization

The key difference between the algorithms can be explained geometrically using the vectoring operation in linear mode. Fundamentally, both algorithms dictate that the vector must traverse a linear path till it iterates to the X axis. This vector traversal is done in the form of angular increments. Fixed angular increments in CORDIC force the vector to oscillate about the X axis prior to convergence, resulting in more iterations. Angular increments in HCORDIC adapt based on the variables in a given iteration. Hence the vector doesn't overshoot the X axis prior to convergence, resulting in fewer iterations. To demonstrate this, both algorithms were simulated on Matlab and the following graphs, in Figure 1 and 2, were plotted for a common input vector in vectoring mode or y-vectoring as defined in [18].

An iteration by iteration illustration of the HCORDIC algorithm has been presented in [18] and clearly showcases its superiority over conventional CORDIC. It is instructive to tabulate the expressions that CORDIC converges to for a combination of mode and operation as shown in Table I and Table II. With a little pre processing, the HCORDIC algorithm can be used to implement higher functionality such as exponent, natural log and square root, apart from the functions presented in Tables I and II.

TABLE I
CORDIC CONVERGENCE IN THE VECTORING OPERATION

$Mode$	x	z
$m = 1$	$K_1\sqrt{x_0^2 + y_0^2}$	$z_0 + \tan^{-1}(y_0/x_0)$
$m = 0$	x_0	$z_0 + y_0/x_0$
$m = -1$	$K_{-1}\sqrt{x_0^2 - y_0^2}$	$z_0 + \tanh^{-1}(y_0/x_0)$

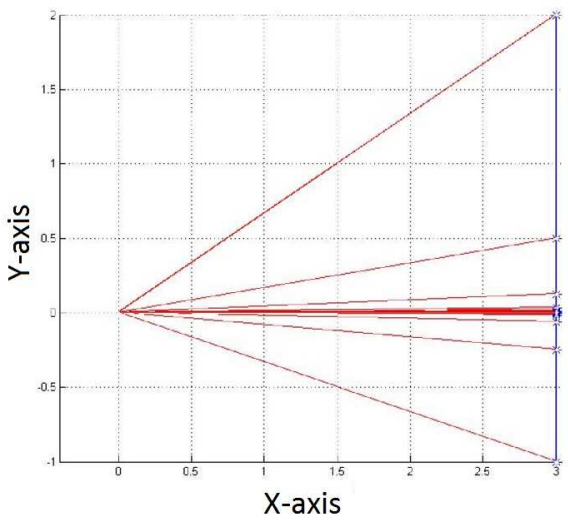

Fig. 1. (a) CORDIC takes 21 iterations for linear vectoring from (3,2) to (3,0)

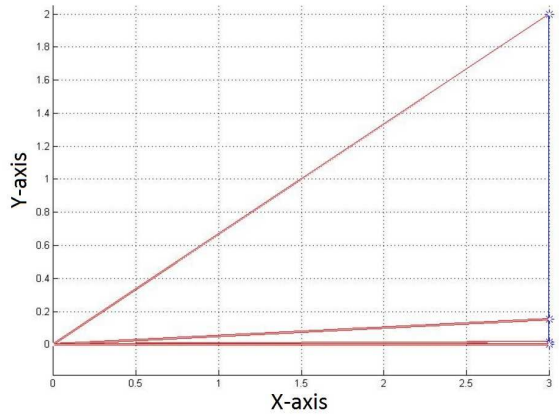

Fig. 2. (b) HORDIC takes 6 iterations from (3,2) to (3,0)

TABLE II
CORDIC CONVERGENCE IN ROTATION OPERATION

$Mode$	x	y
$m = 1$	$K_1(x_0\cos z_0 - y_0\sin z_0)$	$K_1(y_0\cos z_0 + x_0\sin z_0)$
$m = 0$	x_0	$y_0 + x_0 z_0$
$m = -1$	$K_{-1}(x_0\cosh z_0 + y_0\sinh z_0)$	$K_{-1}(y_0\cosh z_0 + x_0\sinh z_0)$

From Table I, it is seen that the square root of a number a can be computed by setting $x = (a+1)/2$ and $y = (a-1)/2$ in the hyperbolic vectoring operation.

$$\tanh^{-1}a = \frac{1}{2}[ln(1+a) - ln(1-a))] \quad (13)$$

Similarly, from the expression of hyperbolic vectoring as shown in Table II and from (13), it is evident that setting $x = (a+1)$ and $y = (a-1)$, the output will converge to scaled natural log of a.

III. Hardware Implementation

Traditionally, the implementation of CORDIC in DSP applications had been limited to software implementation on general purpose processors. The shift and add logic could be implemented as part of the general hardware datapath. It was identified that providing a dedicated hardware path to CORDIC computations would give much better throughput. FPGA based implementation of CORDIC hardware architectures, namely bit serial, bit parallel and loop unrolled architecture is discussed in detail in [8]. Modifications to these conventional architectures are presented in [11], [13] and [19] for higher radix variants of CORDIC algorithm. A majority of these implement a design that assumes a constant scaling factor and a state machine for a fixed number of iterations. A ROM is used to store the value of arctangent.

The hardware design presented in this paper is significantly different to that discussed above. This is attributed to the following differences in the algorithms:

- Non-constant scaling factor: As presented in (7) and (8), the scaling factor is updated in each iteration. This requires a multiplier in the datapath. The hardware cost of the multiplier is compensated by faster convergence in terms of number of iterations. The value of κ_i as shown in (8) has to be stored as ROM in the design.
- Approximation of the tangent by (9) and (11): The update of 2^{-i} is replaced by a more complex computation and the shift and add architecture is not used. As the radix of the approximation is no longer guaranteed to be a multiple of 2, a multiplier is needed to perform (4), (5), (6) and (7). In addition, computing the expression given in (9) and (11) in each iteration is computationally expensive. Hence a look up table (LUT) containing values of all possible permutations of inputs to (9) and (11) is included. This provides the correct value of δ_i, θ_i and κ_i for the ALU to perform its iterative update as shown in (4), (5), (6) and (7).
- Use of IEEE 754 floating point format: The use of the IEEE 754 floating point format is key to the adaptive radix algorithm as this enables a more accurate approximation. However, this introduces the overhead of floating point multipliers and adders which add significantly more number of clock cycles to the design.

The design has a 12 stage deep pipelined architecture with a feedback path. Since it acts as a coprocessor, the implementation has an opcode corresponding to the individual functions performed by the algorithm. An instruction tag is associated with each input to synchronise their out-of-order execution. Among these blocks, fetch contributes to 2 stages of the pipeline for all instructions except square root and natural log. This will be discussed in detail in this section. Decode adds 2 stages while the execute block requires 8 stages owing to its floating point ALUs. This results in a 12 stage pipeline. The coprocessor can be conceptually viewed with 4 main stages, ie Fetch, Decode, Execute and Scaling block. This can be seen in Figure 3 that shows the top level schematic of the design. This design has been simulated using the Xilinx IDE Suite and implemented on Xilinx Virtex 6 XC6VLX240T.

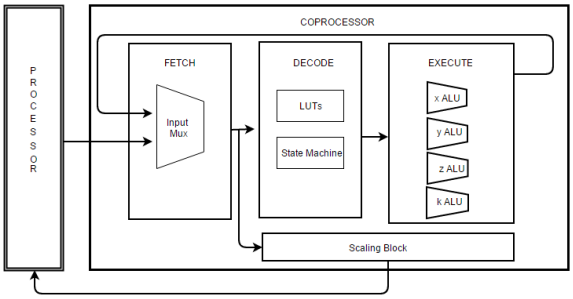

Fig. 3. Schematic design illustrating the functionality of each stage of the implementation as a coprocessor

The schematic in Figure 4 shows the input and output lines to the coprocessor design module. Inputs contain three 32 bit variables representing x, y and z in IEEE 754 floating point format. A 4 bit opcode defining the instruction to be performed by the design and an 8 bit instruction tag to synchronise out-of-order execution is also present. The output contains three 32 bit wide variables x, y and z along with an instruction acknowledge which signals to the processor whether the instruction has been accepted by the module. It will go low when the pipeline is full from the feedback path. The output ready signal is set high to signal to the processor when the module has valid outputs.

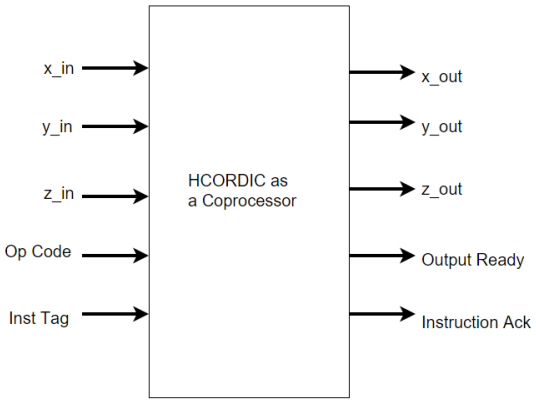

Fig. 4. Inputs and the Outputs to the HCORDIC module

TABLE III
OPCODE, MODE AND OPERATION FOR EACH INSTRUCTION

Instruction	Opcode	Mode	Operation
sin/cos	0000	Circular	1
sinh/cosh	0001	Hyperbolic	1
arctan	0010	Circular	0
arctanh	0011	Hyperbolic	0
Exponential	0100	Hyperbolic	1
Square root	0101	Hyperbolic	0
Natural Log	1001	Hyperbolic	0

Table III shows the opcode, mode and operation corresponding to different instructions. An operation bit 1 denotes rotation while 0 denotes vectoring operation. Mode bits 00, 01 and 11 represent linear, circular and hyperbolic modes respectively

A. Fetch Stage

The fetch block performs three functions as displayed in Figure 5. First, it performs pre processing for square root and natural log. This contains a floating point adder which computes a value for x and y for a given a as explained in the Section II through (13). The extra computation for inputs with opcodes corresponding to these instructions results in out-of-order execution. An instruction tag appended to each of these enable the outputs to be mapped back to its corresponding input by the processor. This will be further illustrated through as example in Section IV.

The next stage is pre decode. For a given opcode, the corresponding mode and operation values are determined as shown in Tables I and II. For example, an opcode of 0000 corresponding to the computation of sin and cos will be pre decoded as circular mode in rotation operation and will be assigned mode and operation bits corresponding to the same. This has been tabulated in Table III.

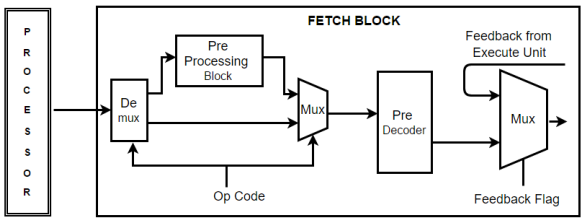

Fig. 5. Block Diagram of Fetch Unit

The final stage is that of arbitration. Multiplexer logic is used to select between the feedback path and the new input received by the processor. The former is given precedence over the latter in case both exist. In addition, the feedback mux plays the crucial role of checking for convergence. In rotation operation, the final limit in (11) ensures that z_{i+1} zeros out by setting θ to $-z_i$ in the last iteration. Hence the mux signals convergence for rotation only when the feedback z_i value is zero. In vectoring, the mux checks if the exponent $y_e - x_e$ has exceeded the range supported by the LUT in the decode block. This limit on the range will be discussed in detail in Part B of this section. When this condition is met, the mux signals convergence in the vectoring operation. Upon convergence, the path is routed to the scaling block.

B. Decode Block

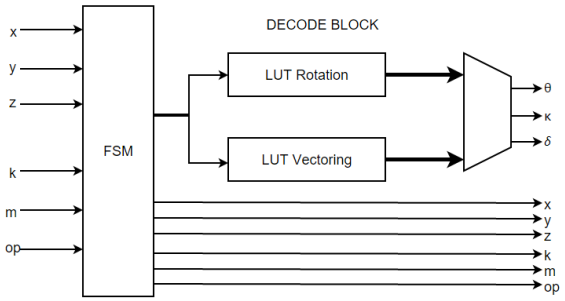

Fig. 6. Block Diagram of Decode Unit

The decode block contains the crux of the control logic in the FSM along with the LUTs that contain the pre-computed values of the weighting factor δ_i, its corresponding angle θ_i and the scaling factor update variable of κ_i as seen in Figure 6. These LUTs store different values for each mode and operation. In rotation, the value of θ_i is identified based on the input and κ_i and δ_i are dependent only on the mode. Whereas vectoring in non linear mode, the value of δ_i is decided based on the inputs alone and θ_i and κ_i are decided based on the mode. The LUTs are designed so that a common address can be used to access the appropriate values of κ_i, δ_i and θ_i.
The control logic in the state machine implements the limits set in (9), (10), (11) and (12). A state is defined corresponding to each limit. Based on the values of mode and operation, the appropriate LUT is enabled. The address to be accessed in the LUT depends upon the values of the inputs. A conceptual view of the LUTs is presented in Figure 7.

The range of values stored in the LUT for rotation operation can be explained as follows. From (11) we can see that the LUT needs to be accessed only when the value of z_e is either less than 0 or greater than $-n/2$, which in the case of IEEE 754 32 bit floating point format corresponds to -12. This value of z_e denotes the number obtained after removing the implicit bias of 127. A range of -1 to -12 requires 4 bits for representation. Further, the value of s has been fixed to 4 as explained in Section II. Hence z_s denotes z_i with all mantissa bits

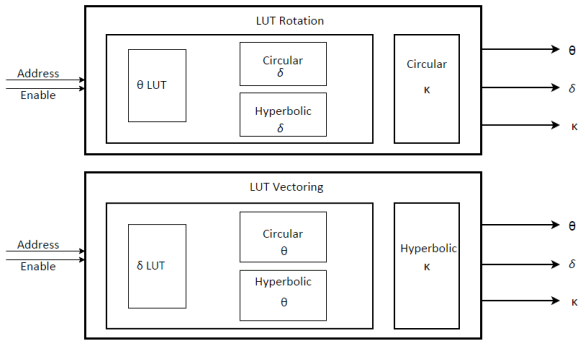

Fig. 7. Conceptual View of LUT

after the most significant four approximated to 0. This gives rise to sixteen possible permutations of z_s. Each combination of $z_s * 2^{z_e}$ can be uniquely represented by four bits of z_s and four bits of z_e with a range of 12. This means the eight bit address line can access a LUT of range 12*16 = 192. Hence the range of the rotation LUTs are 192. The range of values stored in vectoring LUT can be understood along similar lines. From (9), we can see that in non linear modes, the LUT needs to accessed only when the value of y_i is less than x_i. The exponent denoted by $y_e - x_e$ is represented by four bits. However, the approximation of x_s and y_s is slightly different. The value of s for this approximation is set to two to maintain an 8 bit address line. x_s is obtained by approximating all mantissa bits of x_i after the first 2 most significant ones to zero. y_s is obtained by approximating the same bits of y_i to one. A more detailed illustration is presented in [18]. Each 2 bit combination of x_s and y_s gives rise to 4 permutations. This is combined with the 16 permutations of the 4 bit exponent resulting in a LUT range of 4*4*16 = 256 values.

C. Execute Block

The execute block contains 4 parallel datapaths, one for each variable to be updated. Observing (4) and (5), we see that two computations are performed sequentially to update x_i and y_i where first a multiplication is followed by an addition. (6) and (7) show that only one computation is needed for z_i and K_i update, ie addition for z_i and a multiplication for K_i. All the multipliers and adders have been designed for IEEE 754 32 bit floating point arithmetic. Hence, the adder has a latency of 5 clock cycles while the multiplier has a latency of 4 clock cycles. The steps involved in floating point addition and multiplication have been added in Figure 9 for reference. The ALU in the datapath of x_i and y_i requires a multiplier and an adder. This introduces a latency of 8 clock cycles. In order to minimise on latency, longer combinational paths have been introduced to the

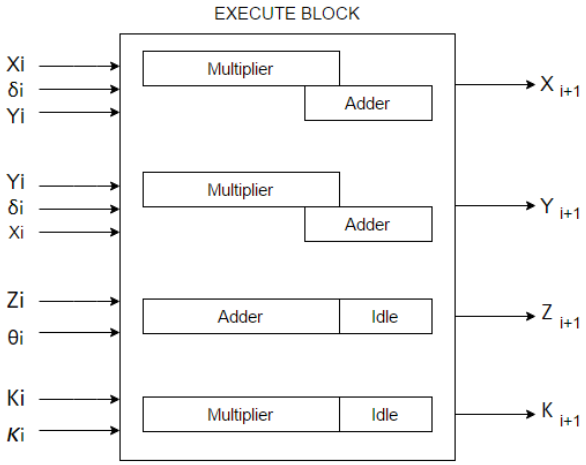

Fig. 8. Block Diagram of Execute Unit

normalization stage of computation using an elaborate mux logic.

Floating point numbers are normalized when the leading mantissa bit is set to 1. For normalization, the number has to left shifted and the exponent subtracted by 1 for every shift. The design employs a combinational mux logic to perform the required number of left shifts in a single clock cycle. This minimises latency at the expense of extra hardware.

A stage has been introduced to identify special cases such as a 0 input for addition or a 1 input for multiplication. If such a case is detected, the remaining datapath is disabled to reduce power dissipation.

Adders require the exponents of both the numbers to be equal to perform fixed point addition of the mantissa. Decrementing the exponent is accompanied by a left shift of the mantissa. This step of addition is performed in the align stage. To reduce this to a single cycle operation, the difference between the exponents of the two numbers is computed in a prior stage and the corresponding shift of the mantissa is performed in the next cycle. Both the add and multiply states seen above are performed as fixed point arithmetic on the mantissa component of the two numbers.

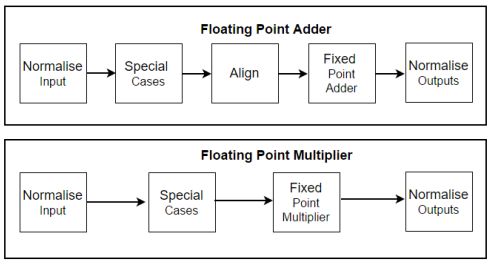

Fig. 9. Stages of Floating point Adder and Multiplier

D. Scaling block

Upon convergence of a set of inputs, the scaling block is enabled. Here the values of x and y are scaled back to the correct output by multiplication with K. This uses floating point multipliers in parallel. Its design is very similar to the same presented in Figure 9.

E. Summary

- The design acts as a 12 stage deep pipelined coprocessor with out-of-order execution.
- The fetch stage receives input from the feedback path and the processor and arbitrates between the two. Thus, conceptually it acts as a mux.
- The decode stage identifies the right value of δ_i, κ_i and θ_i to update the variables. Since these expressions are too computationally expensive to evaluate in every iteration, they are pre-coded and stored in Look Up Tables (LUTs), with specific address for various inputs. This provides for faster hardware access from the LUTs of the values requires.
- The execute stage contains the IEEE 754 32 bit floating point adder and multiplier used to update the x_i, y_i, z_i and K_i variables. Since x_i and y_i require a multiplier followed by an adder sequentially, it defines the latency of the entire datapath.
- Once an input satisfies the convergence test of the feedback mux in Fetch stage, it is routed to the Scaling block where x_i and y_i are scaled back to their correct values by multiplication with K_i.

IV. Illustration

Figure 11 shows the flow of instructions in the pipeline of the HCORDIC block. The horizontal axis represents all the hardware stages of each of the 4 blocks discussed above. The vertical axis represents clock cycles. The values filled in the figure denote the 8 bit wide tags assigned to each of the instructions issued by the processor to the HCORDIC block.

The fetch stage is denoted by $F1_{1-5}$, F2 and F3. $F1_{1-5}$ represents the 5 stages of the pre processing block. F2 and F3 stand for the pre decode and multiplexer logic stages. D1 and D2 perform the 2 decode stages of FSM and LUT access. The next 8 stages correspond to the execute block. The final 5 stages represent the scaling block which is enabled only after the convergence of an instruction.

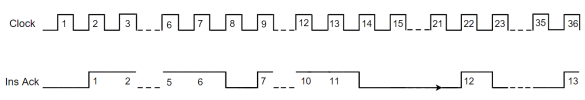

Fig. 10. Waveform illustrating working of Instruction Acknowledge signal

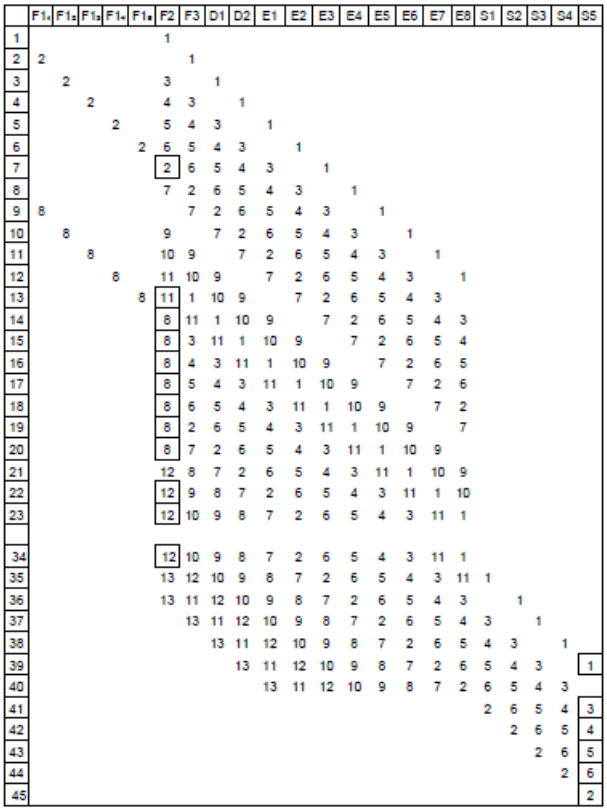

Fig. 11. Example illustrating the working of pipelined implementation

The following assumptions have been made:

1) The processor sends one instruction every cycle to the co-processor.
2) Inputs for each of the instructions converge in three iterations.
3) Functions corresponding to a subset of instruction tags have been tabulated in Table IV. Instruction 2 and 8 correspond to square root and natural log respectively and hence require pre processing.

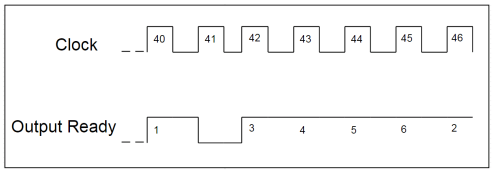

Fig. 12. Waveform illustrating the working of the Output Ready Signal

Instructions 2 and 8 enter the pipeline from the F1 stage while all other instructions enter the pipeline from F2. Once the pipeline is filled, the instruction in the final stage of execute is fed back to F3 which represents the feedback mux. For instruction tag 1, cycle 12 marks the end of the first iteration, cycle 23 ends the second iteration and cycle 34 marks the end of the third iteration.

TABLE IV
PARAMETERS CORRESPONDING TO EACH INSTRUCTION TAG.

Tag	Instruction	Opcode	Mode	Operation	$F1_{1-5}$
1	sin and cos	0000	Circular	Rotation	No
2	Square root	0101	Hyperbolic	Vectoring	Yes
4	sinh and cosh	0001	Hyperbolic	Rotation	No
6	arctan	0010	Circular	Vectoring	No
8	Natural Log	1001	Hyperbolic	Vectoring	Yes
13	arctanh	0011	Hyperbolic	Vectoring	No

Four observations of interest are as follows:

- When a prior instruction that requires pre processing completes F1 and a new instruction that does not require pre processing is issued by processor, two instructions try to access F2 in the same cycle. The prior instruction is given precedence and the new instruction is stalled. The above described scenario is observed in clock cycle 7.
- Once the pipeline gets filled, the feedback path will have a valid instruction. If a new instruction is issued in the same cycle, it will be stalled as priority is given to the feedback path. F1 and F2 will remain stalled as long as the feedback path has a valid instruction. This is seen in clock cycles 13 to 20 and then 22 to 23.
- When an instruction converges, it is routed to the scaling block by the F3 stage. At the same time, the new instruction issued by the processor is also absorbed into the pipeline in that cycle itself. Hence it doesn't result in additional latency. This is seen in the transition from cycle 34 to 35. Instruction 1 converges at the end of cycle 34. Subsequently, F3 routes it to the scaling block and instruction 13 is absorbed into the pipeline.
- The instructions enter the pipeline in order but exit the scaling block. This is clear in the order of the outputs in cycles 39 to 45. This displays execution of the HCORDIC block.

Two control signals are needed to implement the above explained communication with the processor. First, the instruction acknowledge signal is set high when the HCORDIC block has accepted the previous instruction. When the instruction cannot be accepted due to reasons explained in the first two observations, the acknowledge signal is set to zero. This stalls the pipeline prior to F3 which indicates to the processor to stop sending further inputs. Second control signal is the output ready signal that goes high when a valid signal is seen at the output of the scaling block. The waveforms for these two signals along with transition relative to clock have been illustrated in Figure 10 and 12.

V. RESULTS

This section discusses results obtained from the co-processor design presented in Section III. The design has been tested extensively through python generated testbenches for various operations in numerous edge cases. Its performance with these testbenches were simulated using the Xilinx ISE Suite. Further, it was synthesized for the Xilinx Virtex 6 XC6VLX240T board. The results obtained from these tests will be presented as follows. First, the design will be analysed for each of its four modules explained in Section III. Next, an operation based analysis will be presented on three fronts: Power, Timing and Accuracy. Finally, a device utilization summary has been presented to display synthesis results.

A. Modular Analysis

Individual modules were evaluated for their critical path, latency and power consumption. The results have been tabulated in Table V. All three parameters are maximum for Execute Block. The reason for the high latency is the clock cycles needed for the floating point ALU in the x and y datapath. As explained in Figure 9, the multiply stage of the floating point multiplier is a single cycle combinational path. This contributes the highest critical path of the design. The maximum power is due to the combinational path containing an elaborate multiplexer logic in the normalization stage of the floating point adder. This contributes to the highest portion of the power of the Execute unit.

TABLE V
MODULAR ANALYSIS OF HCORDIC AS A COPROCESSOR

Block	Critical Path(ns)	Power(mW)	Latency
Fetch	3.424	8.254	2
Decode	3.42	15.496	2
Execute	4.544	74.148	8
Scaling	4.073	2.102	5

Table V shows the stage wise split with parameters critical path, power dissipation and latency. Figures in Table V for power dissipation is a percentage distribution amongst the stages.

B. Timing Analysis

Timing analysis of the design will be presented on two fronts, namely, the maximum frequency of operation and the latency per instruction. The maximum frequency is determined by the longest critical path in the design, which in this implementation is the execute block. This as shown in Table V is 4.544 ns corresponding to a maximum frequency of 234.1 MHz. Latency varies between 41 and 46 cycles depending on the instruction. This result is expected since a majority of the input vectors require three iterations to converge. Each iteration corresponds to 12 cycles followed by the latency of the scaling block. This latency value is true only for the first instruction. The pipelined implementation ensures single cycle execution for all subsequent instructions. Table VI displays the latency of individual instructions implemented in the design.

TABLE VI
TIMING AND POWER ANALYSIS FOR VARIOUS INSTRUCTION

Instruction	Power Dissipation	Latency
sin and cos	1.9507 mW	41
sinh and cosh	1.9545 mW	41
arctan	1.9712 mW	43
arctanh	1.9566 mW	43
Square root	1.9448 mW	46
Natural Log	1.9485 mW	46

C. Power Analysis

Xpower provided by Xilinx ISE Design Suite is used to obtain the readings of power consumption.To understand the power consumption of the design, the analysis focused on power dissipation per instruction and percentage dissipation on the FPGA. For the former, the testbench was evaluated over a 100 instructions at the maximum frequency. The results obtained varied between 1.94 mW to 1.97 mW per instruction. For a better understanding of power dissipation the latter approach was used. These findings are tabulated in Table VII. Signal power is a measure of power dissipation due to signal transitions. These signal transitions are decided by the testbench used for power analysis. Logic power represents the power dissipation at the gate level instantiation of the HDL logic. The most significant contribution to dynamic power dissipation is through clock toggling at maximum frequency.

TABLE VII
HARDWARE POWER DISSIPATION

Dynamic Power	HCORDIC
Clock	41.6 %
Logic	23.25 %
Signal	24.64 %
IO	9.4 %
DSP Block	1 %

D. Synthesis

The design presented in Section III has been synthesized on Xilinx Virtex 6 XC6VLX240T board. The device utilization summary has been tabulated in Table VIII indicating percentage logic utilization for the HCORDIC design.

TABLE VIII
DEVICE UTILIZATION SUMMARY

Logic Utilization	Used	Available	Utilization
Number of Slice Registers	3294	301440	1%
Number of Slice LUT's	9124	150720	6%
Number of fully used LUT_FF pairs	2648	9770	27%
Number of bonded IOB's	216	600	36%
Number of BUFG/BUFGCTRLs	1	32	3%
Number of DSP48E1s	10	768	1%

E. Accuracy Analysis

The results obtained from the hardware implementation of HCORDIC have been tabulated in Table IX. It presents the values computed by the coprocessor design for specific instruction and input combinations. The difference seen between this value and the expected result represents the error of the operations and assesses the accuracy of the proposed architecture. The IEEE 754 floating point format gives an accuracy of upto the 8th digit after the decimal point in the case of square root. This is attributed to error cancellation seen in hyperbolic vectoring which is the HCORDIC operation used to compute square root.

TABLE IX
OUTPUT ACCURACY OF HCORDIC FOR VARIOUS INSTRUCTIONS

Instruction	Input	Design Output	Error
sin and cos	z_0 = 60 degree	cos z_0= 0.50000011920 sin z_0= 0.866025507450	cos z_0= 1.192*10^{-6} sin z_0= 1.04*10^{-7}
sinh and cosh	z_0 = 60 degree	cosh z_0= 1.6002863645553 sinh z_0= 1.2493667602539	cosh z_0= -4.93*10^{-6} sinh z_0= -2.9*10^{-6}
Square Root	2 x_0 = 1.5; y_0 = 0.5	1.41421356237	2*10^{-8}

VI. CONCLUSION

Faster convergence of high radix adaptive CORDIC (HCORDIC) as compared with CORDIC has been validated through software simulations. A unique architecture has been proposed to realise a dedicated hardware datapath for HCORDIC algorithm in the form of a 12 stage deep pipelined co-processor. Hardware multipliers have been used to enable faster convergence by iterative computation of a non constant scaling factor. The decode unit has been designed for adaptive selection of micro rotations using a novel design of a lookup ROM table. A floating point ALU has been parallelized for higher throughput. The design supports out-of-order execution to accommodate square root and natural log operations. The use of IEEE 754 floating point format has provided high accuracy in computation. The resulting design has been implemented on a Xilinx Virtex 6 X6VLX240T board and analysed for accuracy, power and timing performance. The authors believe that an on-line hardware design can be obtained by loop unrolling the architecture. This may further enhance timing performance.

ACKNOWLEDGEMENT

We are grateful to Dr. El Guibaly, University of Victoria, Canada, for the crucial guidance he has provided. We would like to thank Mr. Pratik Gujjar for his contributions towards the project.

REFERENCES

[1] J. E. Volder, "The CORDIC trigonometric computing technique," *Electronic Computers, IRE Transactions on*, no. 3, pp. 330–334, 1959.

[2] A. S. Dhar and S. Banerjee, "An array architecture for fast computation of Discrete Hartley transform," *Circuits and Systems, IEEE Transactions on*, vol. 38, no. 9, pp. 1095–1098, 1991.

[3] P. Meher, J. Satapathy, and G. Panda, "Efficient systolic solution for a new prime factor Discrete Hartley transform algorithm," *IEE Proceedings G (Circuits, Devices and Systems)*, vol. 140, no. 2, pp. 135–139, 1993.

[4] J. Valls, T. Sansaloni, A. P. Pascual, V. Torres, and V. Almenar, "The use of CORDIC in software defined radios: a tutorial," *Communications Magazine, IEEE*, vol. 44, no. 9, pp. 46–50, 2006.

[5] L. Cordesses, "Direct digital synthesis: a tool for periodic wave generation (part 2)," *Signal Processing Magazine, IEEE*, vol. 21, no. 5, pp. 110–112, 2004.

[6] C. Krieger and B. Hosticka, "Inverse kinematics computations with modified CORDIC iterations," in *Computers and Digital Techniques, IEE Proceedings-*, vol. 143, no. 1. IET, 1996, pp. 87–92.

[7] T. Lang and E. Antelo, "High-throughput CORDIC-based geometry operations for 3D computer graphics," *Computers, IEEE Transactions on*, vol. 54, no. 3, pp. 347–361, 2005.

[8] R. Andraka, "A survey of CORDIC algorithms for FPGA based computers," in *Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*. ACM, 1998, pp. 191–200.

[9] K. Maharatna, J. Valls, T. Juang, K. Sridharan, and P. Meher, "50 years of CORDIC: Algorithms, architectures, and applications." *Circuits and Systems I: Regular Papers, IEEE Transactions on*, vol. 56, no. 9, pp. 1893–1907, 2009.

[10] K. Maharatna, S. Banerjee, E. Grass, M. Krstic, and A. Troya, "Modified virtually scaling-free adaptive CORDIC rotator algorithm and architecture," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 15, no. 11, pp. 1463–1474, 2005.

[11] E. Antelo, J. Villalba, J. D. Bruguera, and E. L. Zapata, "High performance rotation architectures based on the radix-4 CORDIC algorithm," *Computers, IEEE Transactions on*, vol. 46, no. 8, pp. 855–870, 1997.

[12] E. Antelo, T. Lang, and J. D. Bruguera, "Very-high radix circular CORDIC: Vectoring and unified rotation/vectoring," *Computers, IEEE Transactions on*, vol. 49, no. 7, pp. 727–739, 2000.

[13] B. Lakshmi and A. Dhar, "Low latency VLSI architecture for the radix-4 CORDIC algorithm," in *2008 IEEE Region 10 and the Third international Conference on Industrial and Information Systems*, Dec 2008, pp. 1–5.

[14] Y. H. Hu and S. Naganathan, "An angle recoding method for CORDIC algorithm implementation," *Computers, IEEE Transactions on*, vol. 42, no. 1, pp. 99–102, 1993.

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

[16] C.-S. Wu, A.-Y. Wu, and C.-H. Lin, "A high-performance/low-latency vector rotational CORDIC architecture based on extended elementary angle set and trellis-based searching schemes," *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, vol. 50, no. 9, pp. 589–601, 2003.

[17] T. K. Rodrigues and E. Swartzlander Jr, "Adaptive CORDIC: Using parallel angle recoding to accelerate rotations," *Computers, IEEE Transactions on*, vol. 59, no. 4, pp. 522–531, 2010.

[18] F. Elguibaly, N. Sui, and A. Rayhan, "HCORDIC: A high-radix adaptive CORDIC algorithm," *Canadian Journal of Electrical and Computer Engineering -Revue Canadienne de Genie Electrique et Informatique*, vol. 25, no. 4, pp. 149–154, 2000.

[19] H. Dawid and H. Meyr, "The differential CORDIC algorithm: Constant scale factor redundant implementation without correcting iterations," *Computers, IEEE Transactions on*, vol. 45, no. 3, pp. 307–318, 1996.