A CORDIC-based Low-power Statistical Feature Computation Engine for WSN Applications

Dwaipayan Biswas, Koushik Maharatna

Abstract: In this paper we present a carry-save arithmetic (CSA) based Coordinate Rotation Digital Computer (CORDIC) engine for computing eight fundamental time domain statistical features. These features are used commonly in association with major classifiers in remote health monitoring systems with an aim of executing them on a node of Wireless Sensor Network (WSN). The engine computes all the eight features sequentially in *3n* clock cycles where *n* is the number of data samples. We further present a comparative analysis of the hardware complexity of our proposed architecture with an alternate architecture which does not use CORDIC (instead uses standalone array multiplier, divider, square rooter and logarithm converter). The hardware complexity of the two architectures presented in terms of full adder count reflects the effectiveness of using CORDIC for the given application. The engine was synthesized using the STMicroelectronics 130 nm technology library and occupied 205K NAND2 equivalent cell area and consumed 1nW dynamic power @ 50 Hz as estimated using Prime time. Therefore, the design can be applicable for low-power real-time operations within a WSN node.

Keywords: Wireless sensor network, CORDIC, statistical feature extraction, classification, low-power.

D. Biswas
Faculty of Physical Sciences and Engineering, University of Southampton, United Kingdom, SO17 1BJ
Email: db9g10@ecs.soton.ac.uk

K. Maharatna
Faculty of Physical Sciences and Engineering, University of Southampton, United Kingdom, SO17 1BJ
Email: km3@ecs.soton.ac.uk, Telephone: +44(0)2380599322

1 Introduction

Sensor Network (WSN) has revolutionized the modern world, facilitating novel applications like remote health monitoring [1, 2] using pervasive battery-powered sensors for real-time multimodal data acquisition and analysis. The fundamental requirement for the sensor nodes in such a system is low power operation to prolong the battery life of the sensors owing to its resource constrained nature. The major components of these systems include computationally intensive steps like feature extraction from the data acquired by the sensors and its classification and therefore traditionally, these are carried out off-line on mainframe computational facilities. However, for continuous monitoring scenario like motion detection, cardiac monitoring etc., these two steps need to be performed within the sensor node itself for compensating the significant energy required at the radio front-end of the sensors for continuous data transmission [2].

Typical features required for the major classes of classification problems that are relevant to remote monitoring scenario are: mean (μ), root mean square (*rms*), standard deviation *(σ),* index of dispersion (D), kurtosis, skewness, absolute difference and information entropy [3]. These features are extracted from long data series using computationally intensive statistical techniques. Since the energy consumption is proportional to the arithmetic complexity, implementation of such processes in a WSN node consumes a significant amount of energy, affecting the battery life. Therefore, it is a major roadblock when performing these operations in a WSN framework [2]. Therefore, it is of paramount importance to develop a low-power strategy for feature extraction and classification in the resource constrained environment of WSN.

Most of these features require the basic arithmetic operations like addition, subtraction, multiplication, division, square root and logarithm for successful computation. Amongst these arithmetic operations, division, square root and logarithm require special attention for low power implementation. With the recent advances in VLSI, several effective low-power design techniques have been proposed which include the non-restoring algorithm for division [4] and square root calculations [5] and the piecewise-polynomial approximation for logarithm calculation [6]. These algorithms provide a good trade-off between accuracy and hardware complexity and hence have been widely employed in digital signal processing (DSP) applications.

In terms of functional forms most of the features mentioned above have similarities to the different transcendental functions realizable using CoOrdinate Rotation Digital Computer (CORDIC) algorithm. CORDIC is a well-researched area and several specialized architectural implementations [7]-[12] of it have been proposed over the years which can be utilized for processing algorithms in low-power WSN nodes. Therefore, in this paper we propose the design and implementation of a CORDIC-based low-power engine for computing eight common statistical features that are not only used in the classification problems but also for statistical analysis of large data set representing physical phenomena. The primary motivation for using the CORDIC algorithm is to explore its different transcendental functions and compute the complex arithmetic operations reusing the same architecture which can be implemented at low-cost with basic shift-add operations of the form $a \pm b.2^{-i}$ [7].

The engine proposed here could be used for computing a stand-alone or a sequence of features depending on the application. The fundamental mathematical processes of the above mentioned features have been formulated in terms of CORDIC and its optimized implementation strategy has been adapted by analyzing their shared computational stages. The engine is implemented in STMicroelectronics 130 nm CMOS technology with a supply voltage of 1.08V and occupies 4 mm^2 Silicon area after layout (2-input NAND equivalent – 205K). The application area we consider is that of human activity recognition where a sampling frequency of up to 50 Hz is deemed sufficient for capturing the kinematic information of the subjects [3] [13] . The designed engine has a dynamic power consumption of 1nW@50 Hz making it amenable for low-power WSN.

The rest of the paper is structured as follows: a brief review of CORDIC fundamentals is described in Section 2 and the theoretical formulation of the features in terms of CORDIC rotation is described in Section 3. Section 4 describes the architectural design of the proposed feature extraction engine. The implementation and performance evaluation of the designed engine has been discussed in Section 5. Finally, a discussion is presented in Section 6.

2 CORDIC Fundamentals

CORDIC is an iterative algorithm for computing different transcendental functions using 2D vector rotation by employing the following iterative equation:

$$\begin{aligned} x_{j+1} &= x_j - \mu\sigma_j.2^{-j}.y_j \\ y_{j+1} &= y_j + \sigma_j.2^{-j}.x_j \\ z_{j+1} &= z_j - \sigma_j.\alpha_j \end{aligned} \tag{1}$$

where, $[x_j, y_j]^T$, z_j and $\sigma_j \in \{1, -1\}$ are the intermediate result vector, the residual angle and the direction of vector rotation at the *j-th* iteration stage respectively; α_j is the pre-defined angle of rotation at each *j*-th iteration stage $\{= \tan^{-1}(2^{-j})\}$ which add up to make the final target angle of rotation θ; $\mu \in \{1, 0, -1\}$ being the coordinate of rotation – circular, linear and hyperbolic respectively. Given an input vector $[x_0 \quad y_0]^T$, in different coordinate system, CORDIC operates in two modes *viz.* rotation ($z_0 \rightarrow 0$) and vectoring ($y_0 \rightarrow 0$), for computing a series of transcendental functions as shown in Table 1. For a detailed survey of CORDIC please refer to [7]. These forms of the transcendental functions, more specifically, those generated by the vectoring operation of CORDIC in different coordinate systems could be adapted for computing the target features as described in Section 3. For convenience, we use the operators Vec_c, Vec_l and Vec_h to represent vectoring operation of CORDIC in circular, linear and hyperbolic coordinate system respectively.

Table 1 Generalized CORDIC algorithm in three co-ordinate systems

μ	**ROTATION MODE ($Z_0 \rightarrow 0$)**	**VECTORING ($Y_0 \rightarrow 0$)**
	$x_n = K(x_0 \cos z_0 - y_0 \sin z_0)$	$x_n = K\sqrt{x_0^2 + y_0^2}$
1	$y_n = K(y_0 \cos z_0 + y_0 \sin z_0)$	$y_n = 0$
	$z_n = 0$	$z_n = z_0 + \tan^{-1}(y_0/x_0)$
	$x_n = x_0$	$x_n = x_0$
0	$y_n = y_0 + x_0 z_0$	$y_n = 0$
	$z_n = 0$	$z_n = z_0 + (y_0/x_0)$
	$x_n = K_h(x_0 \cosh z_0 - y_0 \sinh z_0)$	$x_n = K_h\sqrt{x_0^2 - y_0^2}$
-1	$y_n = K_h(y_0 \cosh z_0 + x_0 \sinh z_0)$	$y_n = 0$
	$z_n = 0$	$z_n = z_0 + \tanh^{-1}(y_0/x_0)$

3 CORDIC Formulation of Features

In our formulation, the input dataset is represented by d_{si}, where $i \epsilon \{0, 1, 2 \ldots n\text{-}1\}$ and d_i is the output of vectoring CORDIC operation on $d_{s(i\text{-}1)}$ data sample. With this convention the formulation of the eight target statistical features in terms of CORDIC operation is described below.

3.1 Mean (μ)

The mean represents the average of a number of data samples calculated by accumulating n samples and dividing the resultant by n. If $n = 2^m$, where m is an integer, the final division can be achieved by m bit right shift of the result.

3.2 Root mean square (rms)

The *rms* is a measure of the signal energy normalized by the number of samples and is given by:

$$rms = \sqrt{\frac{1}{n}(\sum_{i=0}^{n-1} d_{si}^2)} \tag{2}$$

In terms of the operator Vec_c, *rms* computation could be represented as:

$$rms = \frac{1}{\sqrt{n}}\left(\prod_{i=0}^{n-1} Vec_c[d_i \quad d_{si}]^T\right)_x \tag{3}$$

Physically, (3) means that d_{si} are fed in the y input of the CORDIC while the x-component of the output is fed back to the x-component of its input. Therefore at every clock cycle as the new data sample d_{si} arrives the computed x-component of the CORDIC is given by:

$$d_i = K\sqrt{{d_{s0}}^2 + {d_{s1}}^2 + \ldots.. + {d_{s(i-1)}}^2} \tag{4}$$

After every complete CORDIC operation the x-component of the output is scaled with the scale factor K. If uncompensated, feeding back this result into the x-component of the CORDIC input will result in accumulation of this scale factor corresponding to each d_{si} and therefore (4) will not hold true. To avoid this problem, after every complete CORDIC operation (comprising of N stages) with a set of input data, the scale factor compensation step needs to be invoked before feeding this output to the x-input of the CORDIC for the next iteration. With this scale factor compensation step in place, after n number of operations the final result at the x output of the CORDIC needs to multiplied with $1/\sqrt{n}$ to obtain the true result of *rms*. However since n is a fixed number the value of $1/\sqrt{n}$ could be pre-computed and finally multiplied with the CORDIC output using a reduced complexity fixed-number multiplier or multiplier-less shift-and-add technique.

3.3 Standard Deviation (σ)

σ represents the variation of the data samples from the mean and is expressed as:

$$\sigma = \sqrt{\frac{1}{n}(\sum_{i=0}^{n-1} (d_{si} - \mu)^2} \tag{5}$$

As can be seen from the functional similarity of (2) and (5) the formulation shown in (3) can be reformulated for computing σ in terms of CORDIC operation as:

$$\sigma = \frac{1}{\sqrt{n}}\left(\prod_{i=0}^{n-1} Vec_c[d_i \quad (d_{si} - \mu)]^T\right)_x \tag{6}$$

Similar to the *rms* computation here the *x* component of the CORDIC output needs to be multiplied with the pre-computed value of $1/\sqrt{n}$ to obtain the true value of σ which again can be achieved by a reduced complexity fixed-number multiplier or shift-and-add technique. Like *rms*, here also the scale factor compensation step needs to run after each complete CORDIC operation.

3.4 Index of Dispersion (D)

It is a normalized measure of the dispersion of a data distribution, expressed as:

$$D = \frac{\sigma^2}{\mu} \tag{7}$$

In terms of CORDIC operation *D* may be formulated as:

$$D = \left(Vec_l[\mu \quad \sigma]^T\right)_z \times \sigma \tag{8}$$

Referring to Table 1, setting μ and σ as the x_0 and y_0 inputs to the CORDIC operating in vectoring mode in linear coordinate, the output will result in (σ/μ). This output is then multiplied with σ to obtain the desired value of *D* using a multiplier.

3.5 Kurtosis

Kurtosis is a normalized measure of the dispersion of a data distribution, as expressed in (9) and re-framed in (10):

$$kurtosis = \frac{1}{n}\left(\sum_{i=0}^{n-1} (d_{si} - \mu)^4 \Big/ \sigma^4\right) \tag{9}$$

$$kurtosis = \frac{1}{n}\left[\left\{(d_{s0} - \mu)/\sigma\right\}^4 + \ldots + \left\{(d_{si} - \mu)/\sigma\right\}^4\right] \tag{10}$$

For each sample $(d_{si} - \mu)$, the operator Vec_l produces the output $[(d_{si}-\mu)/\sigma]$ when $(d_{si} - \mu)$ and σ are set as the x_0 and y_0 inputs to the CORDIC. Two squaring circuits and an accumulator module are then used followed by multiplying it with the pre-computed value of $1/n$ to achieve the desired value of kurtosis as shown in Fig. 1:

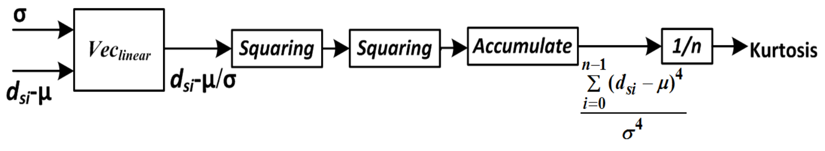


Fig. 1 Architecture for computation of kurtosis

3.6 Skewness

Skewness is a measure of the alignment of the probability distribution of a real valued random variable to one side of the mean and mathematically defined as:

$$skewness = \frac{1}{n}\left(\sum_{i=0}^{n-1}(d_{si}-\mu)^3 \Big/ \sigma^3\right) \tag{11}$$

Exploiting the functional similarity of (9) and (11), the overall architecture for computing *skewness* is shown in Fig. 2.

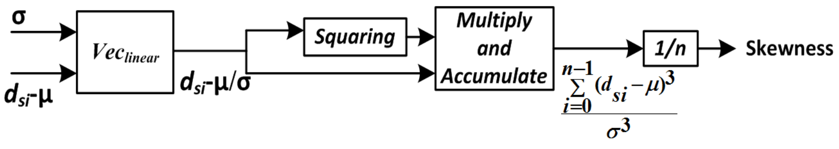


Fig. 2 Architecture for computation of skewness

3.7 Absolute Difference (abs. diff)

It is the absolute difference between the maximum value and the minimum value of a signal and is given by:

$$abs.diff = abs(\max(d_{si}) - \min(d_{si})) \tag{12}$$

This computation does not need a CORDIC operation and can be achieved by using a minimum and maximum detection circuit as shown in Fig. 3 and then taking the absolute difference of the values corresponding to them.

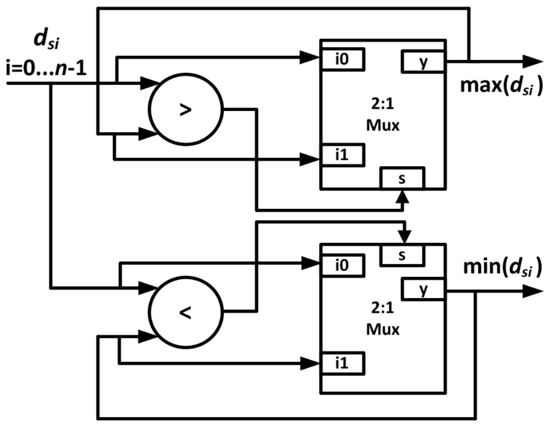


Fig. 3 Architecture for computing maximum and minimum values of a signal

3.8 Information Entropy

Information entropy is a measure of the randomness present in a signal represented by [14]:

$$Info.Entropy = -\sum_{i=1}^{n} p(d_{si}) \log_2 p(d_{si}) \tag{13}$$

To compute it, the histogram plot representing the distribution of data sample is first divided into a number of bins and then the probability distribution is computed by counting the number of samples (frequency of observations) in each bin.

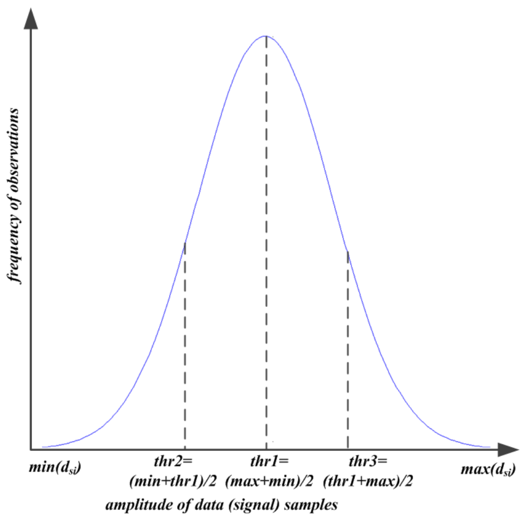


Fig. 4 The selected bins from normal distribution of data samples

For ease of implementation we applied an approximation where the signal d_{si}, is divided into four bins (bin_k, where $k \in \{0, 1, 2, 3\}$) between the *minimum* and *maximum* values of the signal as shown in Fig. 4 considering a Gaussian distribution as an example. In hardware, the bin thresholds could be computed by simple add-shift mechanism. The architecture for computing the probability of the signal in each of the four bins is illustrated in Fig. 5. A comparator logic is used to find out the appropriate bin in which each sample d_{si} belongs and accordingly for each of the bins a sample counter is used to compute the total number of samples lying in it. The sample count in each bin is multiplied with the pre-computed value of $1/n$ to calculate the probability of data samples $p(bin_k)$.

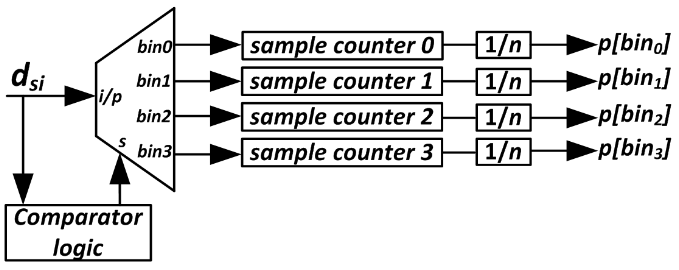


Fig. 5 Architecture for computing the probability of each bin of the signal

The logarithm of the respective probability for each bin can be calculated using the CORDIC operator Vec_h represented as:

$$\ln p(bin_k) = \left(Vec_h \left[(1 + p(bin_k)) \quad (1 - p(bin_k)) \right]^T \right)_z \tag{14}$$

It should be noted that CORDIC computes the natural logarithm (base e), which is further multiplied with a constant scale factor to obtain $\log_2$ (base 2). Therefore altogether four CORDIC operations are needed for computing logarithm of the probability corresponding to the four bins. Accordingly (13) could be realized using the block diagram shown in Fig. 6:

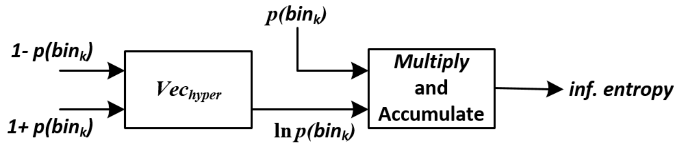


Fig. 6 Architecture for computation of information entropy

4 Architecture and Evaluation

From the foregoing section it is clear that the actual CORDIC operation is needed only for six features out of the eight considered here (except for the μ and *abs. diff*). Typically CORDIC is implemented in two ways: iterative and pipelined. The iterative CORDIC architecture utilises a single implementation of (1) and computes the final result in b iterations where b is governed by the required accuracy and the word-length. Therefore, b clock cycles are required for completion of one CORDIC operation. The pipelined architecture overcomes this problem by exploiting the identical nature of the CORDIC iterations (shift/add operations) and mapping them onto a pipelined architecture. The first output of a N-stage pipelined CORDIC is obtained after N clock cycles and thereafter the outputs will be generated at each clock, helping in achieving a high throughput and is therefore the most popular approach for CORDIC implementation.

The equations (3) and (5) suggest a tight computing recursion, which indicates a computing loop in the corresponding signal flow graph (c.f. Fig. 7). Any attempt to pipeline the computing node in the loop would lead to inaccurate result and therefore, in this particular application, the mathematical formulations described in section 3 cannot be realized using the pipelined CORDIC approach. A slow down process, which inserts N-1 (where N is the pipelined stages) null inputs to every two valid inputs, to obtain the correct result. Without loss of generality, this can be explained by the following example. For pipelined design, this means a poor hardware utilization ratio (only $1/N$). We consider a 4-stage pipelined CORDIC for computing *rms* with a dataset of 8 samples, $\{d_{s0},\ldots,d_{s7}\}$. A cycle-by-cycle snapshot of the process, is presented in Fig. 7, where $x_{i/p}$, $y_{i/p}$ are the x and y component of the CORDIC input and $x_{o/p}$ is the x-component of the output, d_{si} is the input data sample, d_i is the CORDIC output of the $d_{s(i-1)}$ data sample. In this operation, the final target operation is given by:

$$rms = \sqrt{\frac{1}{8}\left[d_{s0}^{\ 2} + d_{s1}^{\ 2} + \ldots.. + d_{s7}^{\ 2} \right]} \tag{15}$$

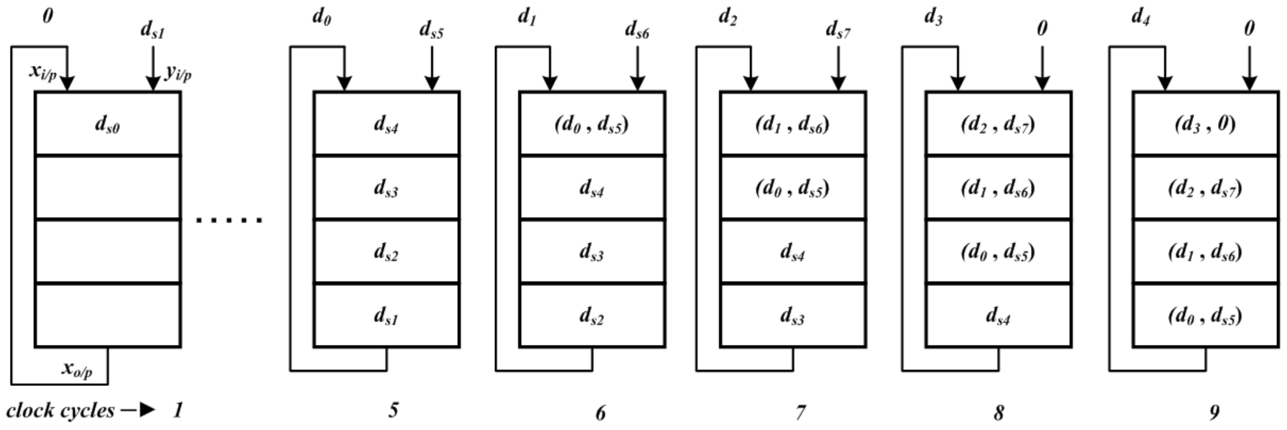


Fig. 7 Pipelined architecture for a 4-stage CORDIC for computing rms on a data stream having eight samples

It is clearly evident from Fig. 7 that the expression in (15) cannot be computed using this pipelined architecture. Moreover, if extra registers are used to store the intermediate result of each pipeline stage, it would nullify the advantage of the pipelined architecture and result in increased complexity in the control mechanism and the associated hardware. On the other hand, although the mathematical form in (15) can be realized using iterative CORDIC, the overall time required will be significantly high and hence the throughput suffers. Therefore, to overcome this problem a unit latency design, which coalesce all iterations into one computing stage (clock cycle), is adopted in this paper. We use carry-save arithmetic (CSA) technique enabling a complete CORDIC operation in one clock cycle [15]. Since, the delay of one CSA adder is equivalent to one carry propagation, for an *N*-stage CSA-CORDIC, the overall propagation delay for computing one complete operation is equivalent to *N* number of carry propagation delay (equivalent to an *N*-bit ripple carry adder) and therefore, is achievable in one clock cycle. Accordingly we have employed a CSA-based CORDIC in our proposed design.

4.1 Architecture

The overall architecture of the statistical feature computation engine is shown in Fig. 8. In principle, it consists of a CSA-based CORDIC module, a subtractor, an accumulator, a probability estimator (as shown in Fig. 5) and a multiply-squaring-accumulate unit. In order to maintain an acceptable level of accuracy (in our case 16-bit) we implement the CORDIC with 24-bit datapath following the principles described in [9]. A 2-bit *mode* signal (*01: circular, 00: linear and 10: hyperbolic*) is used to enable CORDIC operation in different coordinate systems. A control counter is used to input the appropriate data (d_{si} - for *rms*, (d_{si} -μ) - for σ and (d_{si} -μ), μ and σ - for *D*, *kurtosis*, *skewness*) to the CORDIC module at appropriate clock cycle.

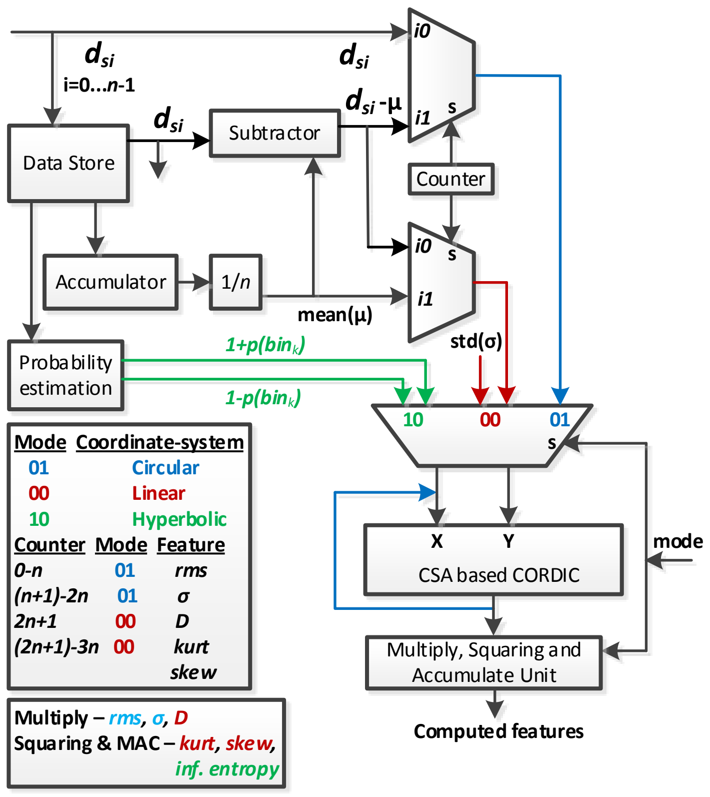
For computing μ, d_{si} are initially stored in the register bank '*Data Store*' from where they are sequentially passed to the accumulator and finally after *n* number of clock cycles, the result is multiplied by the pre-computed value of 1/*n*.

For *rms* computation, in the first cycle the raw samples d_{s0} and d_{s1} are fed into the *x* and *y* inputs of the CORDIC (*Vec*$_c$ *mode*–01*, shown in blue* in Fig. 8). The subsequent *x*-component of the output of the CORDIC is fed back into its *x* input and the next sample d_{s2} into the *y* input of the CORDIC and this process is repeated for *n* number of clock cycles while the end result is multiplied by the pre-computed value of (1/√*n*) to achieve the desired *rms* value.

For computing σ; μ is subtracted from d_{si} using a subtractor and is fed into the CORDIC module to compute the resulting expression $\sqrt{(\sum_{i=0}^{n-1}(d_{si}-\mu)^2}$ which is multiplied by (1/√*n*) to obtain the true value of σ at the end of 2*n* clock cycles.

For computing Dispersion (D) the values μ, σ are used as inputs to the CORDIC (Vec_l *mode*–00, *shown in red* in Fig. 8) to compute (σ/μ) in one cycle which is multiplied by σ to obtain D.

Each sample (d_{si} -μ) and σ are used as inputs to the CORDIC (Vec_l) to generate the expression [(d_{si}-μ)/σ] which is used for calculating *Kurtosis* and *Skewness* in n cycles as shown previously in Fig. 1 and 2. Hence, 3n clock cycles are required for computing the *Kurtosis* and *Skewness*.



Mode	Coordinate-system
01	Circular
00	Linear
10	Hyperbolic

Counter	Mode	Feature
0-n	01	rms
(n+1)-2n	01	σ
2n+1	00	D
(2n+1)-3n	00	kurt skew

Multiply – *rms*, *σ*, *D*
Squaring & MAC – *kurt*, *skew*, *inf. entropy*

Fig. 8 Architectural overview of the CORDIC operation

For computing the *information entropy*, first the values *p(bin$_k$)* is computed for each bin using the probability estimation block and then a pair of adder and subtractor are used for computing [*1- p(bin$_k$)*] and [*1+ p(bin$_k$)*] which are used as inputs to the CORDIC (Vec_h mode–10, shown in green in Fig. 8) and thereby computing ln[*p(bin$_k$)*] at every clock cycle. Finally, the result is multiplied with a constant scale factor (for computing $\log_2$) and *p(bin$_k$)* and accumulated using a MAC unit as shown in Fig. 6.

In our implementation we considered a signal having 256 data samples. Following the procedure described above the μ and *rms* are computed in 256 (n) clock cycles while the σ computation takes 512 (2n) clock cycles altogether. The value of σ is used to compute D in the 513-th (2n+1) cycle and *Kurtosis* and *Skewness* are generated in the 768-th (3n) cycle. *Information entropy* is independent of the other features and requires 4 clock cycles as the logarithm of the probability of each bin is computed in one CORDIC operation.

4.2 Hardware complexity analysis of proposed architecture

We present a hardware complexity analysis considering a generalized word-length b of the N-stage CORDIC module for a single iteration. The hardware resource for one iteration of CORDIC can be reused for multiple iterations (for example, *rms* computation), applicable for all three modes of operation. We present the complexity in terms of the total number of full adders (FA) used. It is important to note here that we have only considered those features which employ the CORDIC module and have not included ($1/\sqrt{n}$) or ($1/n$) in our estimation since it can pre-computed. The mathematical operations with respect to the computation of the features is summarised in Table 2.

Table 2 Arithmetic operations required in the proposed Cordic based architecture

Features	CORDIC	Multiplication	Addition/Subtraction	Accumulator	Squaring
rms	1				
σ	1		1		
D	1	1			
kurtosis	1		1	1	2
skewness	1	1	1	1	1
inf. entropy	1	1	5	1	

- *rms* – CORDIC,
- σ - 1 subtractor for computing (d_{si} -μ) and CORDIC,
- *D* – CORDIC and 1 multiplier for multiplying (σ/μ) with σ,
- *kurtosis* – 1 subtractor for computing (d_{si} -μ), CORDIC, 2 squaring units and 1 accumulator block,
- *skewness* - 1 subtractor for computing (d_{si} -μ), CORDIC, 1 squaring unit, 1 multiplier and 1 accumulator block
- *inf. entropy* - 3 adder/subtractor for calculating the bin thresholds for computing the respective probabilities of each bin, 2 adder/subtractor for calculating the inputs [1- $p(bin_k)$] and [1+ $p(bin_k)$] for computing $log_2p(bin_k)$, hence 5 add/sub operations in total. We can consider 2 adder/subtractors in total which can be used for computing both the inputs to the CORDIC and also the bin thresholds. Moreover we need CORDIC, 1 multiplier and accumulator for multiplying $log_2p(bin_k)$ and $p(bin_k)$ for computing the information entropy as shown in Fig. 6.

As mentioned in the architectural implementation, the features *rms*, σ, *D* and *kurtosis*/*skewness* are computed sequentially, due to their functional dependencies. However, *kurtosis* and *skewness* can be computed in parallel and the computation of *inf. entropy* is independent of any of the above features. In view of this operational sequence, one can reuse majority of the arithmetic components thereby saving hardware. Considering such resource sharing, the optimal list of hardware components required for computing all the six mentioned features are:

- the CORDIC module can be reused for computing all the features based on their formulation and architecture as mentioned in section 3,
- 1 subtractor is required for computing (d_{si} -μ) for the features: σ, *kurtosis*, *skewness*. We can reuse this subtractor in the computation of *inf. entropy* (requiring 2 add/sub operations), thereby requiring 1 additional adder/subtractor.
- 1 multiplier can be reused for *D*, *skewness* and *inf. entropy*,
- 2 accumulator blocks are needed since the computation of *kurtosis* and *skewness* takes place in parallel and can be reused for *inf. entropy*. ,
- 3 squaring units are needed for *kurtosis* and *skewness*.

For the sake of convenience, we consider 2 squaring units as 1 multiplier, therefore we require 2.5 multipliers in total (1 + 1.5 squaring unit). A conventional array multiplier (CAM) requires $b(b - 2)$ FA, b half adders (HA) and b^2 AND gates [16]. Considering, 2 HA as 1 FA and 4 AND gates as 1 FA [16] (due to transistor count and area), we can reduce the total gate count of a CAM to ($1.25b^2 – 1.5b$) FA. Hence, for 2.5 multipliers we need $A_{mult} = 2.5(1.25b^2 – 1.5b)$ FA, where (A_*) represents the total number of FA's in each circuit.

We consider an accumulator block to comprise of a FA (we do not consider any registers associated with the accumulator, since we are concerned with the mathematical operations only). A b-bit Ripple carry adder/subtracter (RCA) requires b full adders (FA). Therefore in total we require $A_{add/sub} = 4b$ FA (2 adder/subtractors + 2 accumulators).

A N stage b-bit CORDIC implemented using Carry-Save Arithmetic (CSA) requires $6Nb$ FA. For our case, N=16, hence the CORDIC module requires $A_{CORDIC} = 96b$ FA. Therefore the total gate count for the implementation of these six features (let us name it as *archt1*) in terms of FA count is $A_{\text{archt1}} = (3.125b^2 + 96.25b)$ FA. Hence, for a 24-bit datapath, $A_{\text{archt1}} = 4110$ FA.

4.3 Hardware complexity analysis of non-CORDIC architecture

Now let us consider an alternative architecture (*archt2*) for computing the same six features (*rms*, *σ*, *D*, *kurtosis*, *skewness* and *inf. entropy*), also summarized in Table 3. In this architecture we consider a Ripple carry adder (RCA), conventional array multiplier (CAM), non-restoring iterative cellular square rooter (SQRT), non-restoring array divider (NAD) and multiplicative normalization based logarithm [17] as the arithmetic components for implementing the fundamental mathematical operations. As in the proposed CORDIC based design (*archt1*), we do not consider ($1/\sqrt{n}$) or ($1/n$) in our estimation.

Table 3 Arithmetic Operations Required in Non-Cordic Based Architecture

Features	Division	Multiplication	Addition/Subtraction	Square root	Accumulator	Squaring
rms				1	1	1
σ			1	1	1	1
D	1	1				
kurtosis	1		1		1	2
skewness	1	1	1		1	1
inf. entropy		1	9		1	

- *rms* – 1 squaring unit, 1 SQRT and 1 accumulator,
- *σ* - 1 subtractor for computing (d_{si} -μ), 1 squaring unit, 1 SQRT and 1 accumulator,
- *D* – 1 NAD and 1 CAM,
- *kurtosis* – 1 subtractor for computing (d_{si} -μ), 2 squaring units, 1 NAD and 1 accumulator block,
- *skewness* - 1 subtractor for computing (d_{si} -μ), 1 squaring unit, 1 NAD, 1 CAM and 1 accumulator block,
- *inf. entropy* - 3 adder/subtractor for calculating the bin thresholds for computing the respective probabilities of each bin. For computing the logarithm of each bin, $log_2p(bin_k)$ we need 2 variable shifters, a 4:2 adder, a 3:2 adder, a carry propagate adder (CPA), selection module, 2 multiplexers, a look-up table (LUT) and 4 registers [17]. A 4:2 adder corresponds to $3b$ FA, a 3:2 adder can be realized by $2b$ FA and a b-bit CPA requires b FA, hence requiring $6b$ FA in total. In addition, we require 1 CAM for multiplying the probability of each bin, $p(bin_k)$ with its logarithm $log_2p(bin_k)$. It is important to mention here that we are considering one iteration of the logarithm computation. In the end, we use an accumulator block for adding the product [$p(bin_k)* log_2p(bin_k)$] for each bin.

We can reuse most of the components keeping in view the sequential nature of the feature computation with an exception for *kurtosis* and *skewness* which are computed in parallel after computing *σ* and the computation of *inf. entropy* is independent of the other features. Therefore the optimal list of hardware components required for the computing the six features are:

- 1 SQRT can be reused for *rms* and *σ*,
- 1 subtractor is required for computing (d_{si} -μ) for the features: *σ, kurtosis*, *skewness* and 1 adder for computing the bin thresholds for *inf. Entropy*.

- For computing logarithm, considering, *m* iterations for computing logarithm, we require *m*6*b* FA. For the sake of convenience, we consider *m*=16 (similar to the stages in CORDIC), thereby we require 96*b* FA.
- 2 accumulator blocks are needed since the computation of *kurtosis* and *skewness* takes place in parallel, they can be reused for *rms*, σ and *inf. entropy*,
- 1 CAM can be reused for *D*, *skewness* and *inf. entropy*,
- 3 squaring units required for *kurtosis* and *skewness* which are computed in parallel, one of the squaring units can be reused for *rms* and σ.
- 2 NAD for *kurtosis* and *skewness*, one of which can be reused for computing *D*.

Therefore, we require 2.5 multipliers in total (1 + 3/2 squaring unit), hence $A_{mult} = 2.5(1.25b^2 - 1.5b)$ FA. The total count for adder/subtractor, $A_{add/sub} = 100b$ FA (2 adder/subtractor + 2 accumulators + 96*b* FA*)*. A $b \times b$ NAD requires $0.5 \times b(3b - 1)$ FA and $0.5 \times b(3b - 1)$ XOR gates. One *b*-bit SQRT requires $0.125 \times b(b + 6)$ FA and XOR gates [16]. Therefore, the total FA count: $A_{NAD} = (4.5b^2 - 1.5b)$ FA, $A_{SQRT} = (0.1875b^2 - 1.125b)$ FA. The total gate count for computing these six features using an alternate architecture (*archt2*) in terms of FA count is $A_{archt2} = (7.8125b^2 + 93.625b)$ FA. Hence, for a 24-bit datapath, $A_{archt2} = 6747$ FA.

It is important to note here that for the complexity analysis, we did not consider the selection module, 2 multiplexers, a look-up table (LUT) and 4 registers used in logarithm computation. We also leave out the variable shifters since it cannot be ascertained. This is similar to our assumptions for the proposed CORDIC based design where we did not consider the comparator and counter logic for computing the probability of each bin $p(bin_k)$.

We present a comparative analysis in Fig. 9, for both the architectures (*archt1* – CORIC based, *archt2* – without CORDIC, considering *m* = 8, 12 and 16 in logarithm computation), varying the word-length, to show the number of FA being used for computing the six features. This clearly shows the effectiveness of our proposed CORDIC based feature computation engine in terms of hardware complexity which is further prominent with an increase in word-length.

To the best of our knowledge there is no unified architecture or design for computing the statistical features considered in our study. Therefore, for the alternate architecture (*archt2* – without CORDIC) we consider a Ripple carry adder (RCA), conventional array multiplier (CAM), non-restoring iterative cellular square rooter (SQRT), non-restoring array divider (NAD) and multiplicative normalization based logarithm to compute the required features and provided a hardware complexity analysis in terms of basic arithmetic operations i.e. full adder (FA) counts. This complexity analysis presents a more objective reflection of the qualitative difference between the two architectures. Hence in the next section we present the synthesis and verification results for only our proposed CORDIC based design.

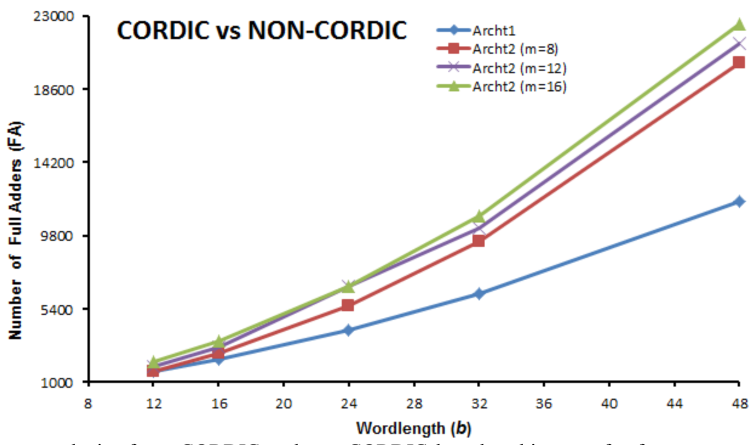


Fig. 9 Comparison of hardware complexity for a CORDIC and non-CORDIC based architecture for feature extraction, showing variation in the number of full adders (FA) required with change in word-length

5 Synthesis and Verification

The architecture was coded using Verilog as the HDL. To verify its functional correctness we randomly generated five datasets of 256 samples in the range of -20 to +20, and computed the target features using Matlab. The Verilog output on the same dataset are then compared with the Matlab results and the average errors are calculated as shown in Table 4. It is evident that the average error may become significant for the features particularly involving higher-order terms even when the accuracy of the CORDIC itself is set high. To achieve higher accuracy, therefore, adjusting the datapath width for the MAC unit may be necessary depending on the error tolerance of an application.

Table 4 Average error between Matlab and RTL simulation

Features	Average Error
Mean	0
Absolute Diff	0
RMS	$O(2^{-8})$
Standard Deviation	$O(2^{-10})$
Index of Dispersion	$O(2^{-11})$
Kurtosis	$O(2^{-8})$
Skewness	$O(2^{-12})$
Entropy	$O(2^{-12})$

The design was synthesized using STMicroelectronics 130-nm technology library with a supply voltage of 1.08V and a clock frequency of 50 Hz. The 2-input NAND gate equivalent cell area of the synthesized design was 205K. The dynamic power was calculated with the synthesized design using Prime time and is 1nW@50 Hz. Although the design was synthesized at 50 Hz, the functionally verified upper range of frequency it is capable of running is 75 MHz. Since the application area we consider is that of human activity recognition where these time domain features are generally extracted from kinematic data sampled at very low frequencies (20 ~ 50 Hz) from body-worn inertial sensors, we synthesize our design at 50 Hz. However its functionality was verified at higher clock frequencies (up to 75 MHz) for high speed applications. The core layout consumes 4 mm^2 silicon area and is shown in Fig. 10.

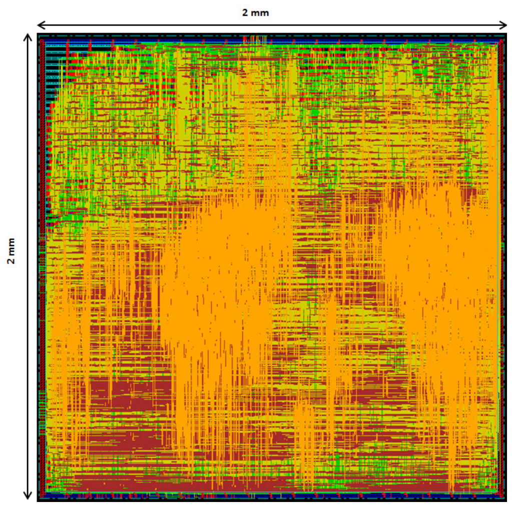


Fig. 10 Core chip layout

6 Discussion

In this paper we have designed and implemented a CORDIC based engine for computing eight time-domain statistical features fundamental in remote monitoring applications involving classification. The engine computes all the eight features sequentially in $3n$ clock cycles and could also be utilized for stand-alone feature computation.

Our analysis shows that the engine is power efficient and therefore may be implemented within a sensor node for on-board feature extraction. It also shows that specifically for the features involving higher-order terms the accumulation of arithmetic error may be significant and therefore an appropriate trade-off between the accuracy and word-length is required depending on a particular application.

Acknowledgments

This work was supported by the Information and Communication Technologies Theme of the European Union Seventh Framework Programme, under the project name "StrokeBack: Telemedicine system empowering stroke patients to fight back" (grant number 288692).

References

[1] H. Alemdar, C. Ersoy, "Wireless sensor networks for healthcare: A survey," *Elsevier Computer Networks*, vol. 54, no. 15, Oct. 2010.

[2] K. Maharatna, E. B. Mazomenos, J. Morgan, and S. Bonfiglio, "Towards the development of next-generation remote healthcare system: Some practical considerations," *in Proc. IEEE Int. Symp. Circuits and Systems (ISCAS)*, Seoul, pp. 1–4, May. 2012.

[3] M. Ermes, J. Parkka, J. Mantyjarvi, and I. Korhonen, "Detection of daily activities and sports with wearable sensors in controlled and uncontrolled conditions," *IEEE Trans. Inf. Technol. Biomed.*, vol. 12, no.1, pp. 20-26, Jan. 2008.

[4] P. Nair, D. Kudithipudi, E. John, "Design and Implementation of a CMOS Non-Restroing Divider," *Region 5 Conference, IEEE*, pp. 211-217, 7-9 Apr. 2006.

[5] K.N. Vijeyakumar, V. Sumathy, P. Vasakipriya, and A.D. Babu, "FPGA Implementation of Low Power High Speed Square Root Circuits," *Computational Intelligence & Computing Research (ICCIC), IEEE International Conference on*, pp. 1-5, 18-20 Dec. 2012.

[6] D.D. Caro, M. Genovese, E. Napoli, N. Petra, and A.G.M. Strollo, "Accurate Fixed Point Logarithm Converter," *IEEE Trans. Circuits and Systems II: Express Briefs*, no. 99, pp. 1-5, May. 2014.

[7] P.K. Meher, J. Valls, T.B. Juang, K. Sridharan, K. Maharatna, "50 Years of CORDIC: Algorithms, Architectures, and Applications," *IEEE Trans. Circuits and Systems I*, vol. 56, no. 9, pp. 1893-1907, Sept. 2009.

[8] C.H. Lin and A.Y. Wu, "Mixed-Scaling-Rotation CORDIC (MSR-CORDIC) Algorithm and Architecture for High-Performance Vector Rotational DSP Applications," *IEEE Trans. Circuits and Systems-I*, vol. 52, no. 11, pp. 2385-2396, Nov. 2005.

[9] L. Vachhani, K. Sridharan, and P.K. Meher, "Efficient CORDIC Algorithms and Architectures for Low Area and High Throughput Implementation," *IEEE Trans. Circuits and Systems-II*, vol. 56, no. 1, pp. 2385-2396, Jan. 2009.

[10] C.Y. Kang and E. Swartzlander, "Digit-Pipelined Direct Digital Frequency Synthesis Based on Differential CORDIC," *IEEE Trans. Circuits and Systems-I*, vol. 53, no. 5, pp. 1035-1044, May. 2006.

[11] E. Grass, B. Sarker, and K. Maharatna, "A Dual-Mode Synchronous/Asynchronous CORDIC Processor," *in Proc. Eighth Int. IEEE Symposium on Asynchronous Circuits and Systems*, pp. 76–83, Apr. 2002.

[12] K. Kota and J.R. Cavallaro, "Numerical Accuracy and Hardware Tradeoffs for CORDIC Arithmetic for Special-Purpose Processors", *IEEE Trans. Computers*, vol. 42, no. 7, pp. 769-779, Jul. 1993.

[13] S. Chernbumroong, S. Cang, A. Atkins, and H. Yu, "Elderly activities recognition and classification for applications in assisted living," *Expert Systems with Applications*, vol. 40, no. 5, pp. 1662-1674, Apr. 2013.

[14] Y.J. Hong, I.J. Kim, S.C. Ahn, and H.G. Kim, "Mobile health monitoring system based on activity recognition using accelerometer," *Elsevier Simulation Modelling Practice and Theory*, vol. 18, no. 4, pp. 446-455, Apr. 2010.

[15] R. Kunemund, H. Soldner, S. Wohlleben, and T. Noll, "Cordic Processor with Carry-Save Architecture," *in Proc. ESSCIRC 90*, Sept. 1990, pp 193-196.

[16] A. Acharyya, K. Maharatna, B.M. Al-Hashmi, and J. Reeve, "Coordinate Rotation Based Low Complexity N-D FastICA Algorithm and Architecture," *IEEE Trans. Signal Processing*, vol. 59, no. 8, pp. 3997-4011, Aug. 2011.

[17] M.D. Ercegovac and T. Lang, "Digital Arithmetic", Morgan Kaufmann, pp. 549 – 607, 2043.