# Fast, Low Power Evaluation of Elementary Functions Using Radial Basis Function Networks

Parami Wijesinghe, Chamika M. Liyanagedera, *Student Member, IEEE* and Kaushik Roy, *Fellow, IEEE*
School of Electrical and Computer Engineering
Purdue University
West Lafayette, Indiana, 47907

***Abstract*—Fast and efficient implementation of elementary functions such as sin(), cos(), and log() are of ample importance in a large class of applications. The state of the art methods for function evaluation involves either expensive calculations such as multiplications, large number of iterations, or large Lookup-Tables (LUTs). Higher number of iterations leads to higher latency whereas large LUTs contribute to delay, higher area requirement and higher power consumption owing to data fetching and leakage. We propose a hardware architecture for evaluating mathematical functions, consisting a small LUT and a simple Radial Basis Function Network (RBFN), a type of an Artificial Neural Network (ANN). Our proposed method evaluates trigonometric, hyperbolic, exponential, logarithmic, and square root functions. This technique finds utility in applications where the highest priority is on performance and power consumption. In contrast to traditional ANNs, our approach does not involve multiplication when determining the post synaptic states of the network. Owing to the simplicity of the approach, we were able to attain more than 2.5× power benefits and more than 1.4× performance benefits when compared with traditional approaches, under the same accuracy conditions.**

## I. Introduction

The generation of elementary functions such as sin(), cos(), log() etc. finds widespread applications in Digital Signal Processing (DSP) [1], robotics [2], bio-medical systems [3], 3-D computer graphics [4] and other multimedia systems. Function evaluation can often be the performance and energy efficiency bottleneck of many compute-bound applications. Different software routines for elementary function evaluation are discussed in [5]. These routines are often not sufficiently fast for numerically intensive applications [6]. Thus the use of dedicated hardware to compute functions is of paramount interest.

Popular methods of elementary function evaluation include lookup table based interpolation [7][8], CORDIC [9], BKM [10] and Polynomial approximations [11]. All these algorithms use LUTs and their size is a deciding factor for the accuracy of the evaluated function. Some applications use direct lookup tables, where the output values of a function for all pertinent inputs are stored. Such a design can be attractive owing to its ease of design and fast execution time, given that no expensive computation is required [12]. However, the table size grows exponentially with the word length and can become impractically large [12]. Larger LUTs consume more on-chip area and power. Hence, there is a need for increased functionality with minimal on-chip memory requirements.

The proposed method of function evaluation consists of an RBFN and a minimal sized LUT. RBFN is a type of ANN with a set of radial basis functions (RBFs) as its activation functions. RBFs ($\phi$) are functions where the response monotonically increases or decreases with the distance from a point referred to as the bias value. Typical choices for the activation functions include Multiquadric, Gaussian, and linear functions [13]. Even though the Multiquadric (1) and Gaussian (2) activation functions offer higher accuracy and faster solution convergence, their implementations are more complicated compared to other activation functions [14]. Therefore, in this work, we use linear RBFs (3) due to their ease of implementation.

$$\phi_i = \sqrt{1 + \beta||\hat{x} - \hat{a}_i||^2} \quad (1)$$

$$\phi_i = e^{-\beta(||\hat{x} - \hat{a}_i||^2)} \quad (2)$$

$$\phi_i = ||\hat{x} - \hat{a}_i|| \quad (3)$$

$$\hat{x} = [x_1, x_2, ..., x_M] \quad (4)$$

In the above equations, $\phi_i$ is the $i^{th}$ RBF of the network, $\hat{a}_i$ is the bias vector for that RBF, $\hat{x}$ is the input (argument) vector, and $\beta$ is a constant. The output of an RBFN can simply be elaborated as a linear combination of a set of RBFs as shown in Fig. 1 where $w_i$ represents the weights connecting the RBFs to the output layer, $x_j$ represents the input to the network, and $y$ is the output of the network.

An RBFN can be either a single hidden layer or multi hidden layer network. However, the traditional RBFN as elaborated by Broomhead and Lowe [15], has only 3 layers $viz.$ input, hidden and output layers. The synaptic weights of the dendrites extending from the inputs towards the hidden layer neurons of the RBFN are set to unity. Our selection for this type of neural network is its simplicity, which leads to our main goal; obtaining power and performance benefits when evaluating elementary functions.

In this work, we propose an RBFN for evaluating sin(), cos(), exp(), log(), sinh(), cosh(), tanh() and square root() (sqrt()) functions. All the weights of the network are forced to be powers of two, replacing all the multiplications in the network by simple bit shifting operations. Such a constraint leads to lower power consumption, albeit with some degradation in accuracy. However, the loss of accuracy can be recovered by the following methods.

- Increasing the number of neurons in a hidden layer.
- Increasing the number of hidden layers in the network.
- Changing the RBF used as the activation function.

In this work, we have only focused on the first method to analyze the trade-offs between, accuracy, power, performance, and area.

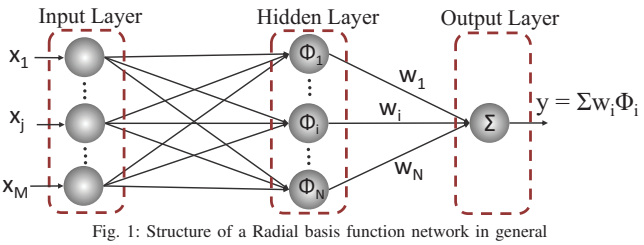


Fig. 1: Structure of a Radial basis function network in general

978-3-9815370-8-6/17/$31.00 ©2017 IEEE

We introduce a new training paradigm designed to identify the optimal weights and bias values for the network, and is explored in detail in section III. The calculated bias values are stored in an LUT. The power consumption, delay and area associated with the LUT were also taken into consideration in this analysis. Our proposed method offers power, area, and performance benefits when compared with traditional function evaluation algorithms. Target applications of this hardware function evaluator will include all image processing and 3D graphics applications, where the highest priority is on performance and power consumption.

## II. Existing Methods To Evaluate Functions

Let us consider three popular methods for evaluating functions; namely, LUT based interpolation [7][8], Polynomial method (based on Remez algorithm) [11] and CORDIC [9]. These methods differ in terms of power, area and memory requirements. Note, all of these methods require LUTs.

### A. LUT Based Interpolation

LUT based interpolation is popular owing to its simplicity and performance. In the direct LUT method, the table stores the outputs corresponding to all possible inputs. Note that the table size can be large since there is a need to store a significant number of data points. However, if the outputs corresponding to all pertinent inputs are not stored, interpolation is necessary to calculate outputs for intermediate inputs. Numerous interpolation methods have been proposed over the past [8]. We have considered linear interpolation [7][8] for all our comparisons.

Linear interpolation approximates a value for $f(x)$, using two LUT entries ($[x_1, f(x_1)]$ and $[x_2, f(x_2)]$), where $x_1 < x < x_2$. Each LUT entry is an input ($x_i$) and the corresponding output ($f(x_i)$) of the function $f$. We define these entries as knots. Accuracy increases when the function value of two consecutive knots are close to each other (Fig. 2). Equations (5) and (6) show the estimation of the output $f_{est}(x)$ and the associated error $E(x)$, respectively.

$$f_{est}(x) = \left\{ \frac{f(x_1) - f(x_2)}{x_1 - x_2} \right\} (x - x_1) + f(x_1) \tag{5}$$

$$E(x) = |f(x) - f_{est}(x)| \tag{6}$$

In (5), note that calculating the gradient (i.e. $\left\{ \frac{f(x_1)-f(x_2)}{x_1-x_2} \right\}$) involves division, which is an expensive calculation in terms of power, area and delay. This can be avoided by storing the gradient values in the LUT as well. We have followed this approach for the linear interpolation method, when comparing power, delay and area values with the RBFN based method.

### B. Polynomial Method

Any continuous function $f$ defined in a particular interval can be represented as a polynomial of order $n$ (7) [7]. $x$ is the input.

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0 \tag{7}$$

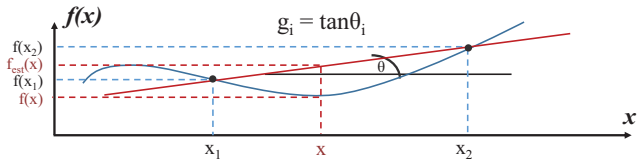


Fig. 2: Linear Interpolation of a function $f$

The coefficients of the polynomials $(a_0, a_1, ..., a_n)$ can be determined by the Remez algorithm [16]. These coefficients are stored in an LUT. Usually, the memory requirement of this method is lower in comparison to other methods since one can store only a few coefficients if accuracy is not the primary concern [12]. Higher order polynomials offer higher accuracy. However, since the algorithm involves evaluation of higher orders of the input, the power consumption is higher than other methods.

### C. CORDIC

The basis for CORDIC algorithm is coordinate rotation in a linear, circular, or hyperbolic coordinate system, depending on which function is to be evaluated [9]. The algorithm utilizes simple operations such as addition, subtraction, bit shift and recall of pre-stored constants. Hence, CORDIC consumes less power when compared with other methods such as polynomial, where several multiplications are required to achieve a considerable accuracy. The CORDIC evaluation is an iterative process, and the number of iterations determines the accuracy of the outcome. These iterative equations are,

$$\begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = \begin{bmatrix} 1 & m\delta_i\alpha_i \\ -\delta_i\alpha_i & 1 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} \tag{8}$$

$$z_{i+1} = z_i + \delta_i \theta_i \tag{9}$$

$$\theta_i = \begin{cases} tanh^{-1}(\alpha_i) & if\ m = -1 \\ \alpha_i & if\ m = 0 \\ tan^{-1}(\alpha_i) & if\ m = +1 \end{cases} \tag{10}$$

where $i$ is the current iteration, $\delta_i$ is $\pm 1$, and $m$ determines the class of functions being evaluated $viz.$ linear ($m$ = 0), circular ($m$ = +1), or hyperbolic ($m$ = -1). $\alpha_i$ is set to $2^{-i}$.

One of the main problems associated with the CORDIC algorithm is the limited convergence domain, in which the functions can be calculated. For an example, $sin(x)$ can only be evaluated within the interval $0 \leq x < \frac{\pi}{2}$. These intervals according to [9] are shown in table 1 and they are smaller than the range of inputs (argument intervals) we have considered in this work for comparison. Two different approaches can be employed to overcome this constraint: first, an argument reduction method [9] and second, an expansion of the CORDIC convergence domain [17]. While the first approach requires processing overhead due to the need for divisions (for certain functions only), the second technique requires more number of iterations than the first method [17]. In this work, we have chosen the argument reduction mechanism originally proposed by Walther [9] when comparing power, delay, and area with the RBFN method we propose.

Among the elementary functions analyzed here, only sin(), cos(), sinh(), and cosh() can be directly solved using CORDIC. To evaluate tanh(), ln(), exp(), and sqrt(), some identities proposed in [9] are used.

## III. Proposed RBFN

### A. Basic Structure of the RBFN

RBFN is a type of an artificial neural network for supervised learning applications. The RBFN used in this work is a single input single output network (since the elementary functions are also single input single output) with one hidden layer as shown

TABLE I: Range Of Argument In CORDIC And RBFN Methods

| $f(x)$ | CORDIC | RBFN | |
|---|---|---|---|
| | Range of $x$ | Range of $x$ | Range of $f(x)$ |
| sin(x) | $[0,\frac{\pi}{2})$ | [0, 2π] | [-1, 1] |
| cos(x) | $[0,\frac{\pi}{2})$ | [0, 2π] | [-1, 1] |
| sinh(x) | [0,ln2) | [0, 2π] | [0, 267.75] |
| cosh(x) | [0,ln2) | [0, 2π] | [1, 267.75] |
| tanh(x) | [0,ln2) | [0, 2π] | [0, 1) |
| exp(x) | [0,ln2) | [0, 7] | [1, 1096.63] |
| ln(x) | [0.5,1) | [0.01, 100] | [-4.61,4.61] |
| sqrt(x) | [0.5,1) | [0, 100] | [0,10] |

in Fig. 3. The weights between the input and hidden layer are set to unity while the weights between the hidden layer and the output are constrained to powers of two. In Fig. 3, $a_i$ and $\phi_i$ represent the bias value and the RBF, associated with the $i^{th}$ hidden layer neuron respectively. $ri$ is the exponent of 2, corresponding to the synaptic weight that connects $i^{th}$ hidden layer neuron to the output. We have utilized linear RBFs of the form illustrated in (11), as activation functions for our network.

$$\phi_i(x) = |x - a_i| \tag{11}$$

The output of the RBFN is the weighted sum of the hidden layer outputs as depicted in (12). This output is referred to as $f_{est}(x)$ and it is an estimate of the actual function output value $f(x)$.

$$f_{est}(x) = \sum_{i=1}^{N} w_i |x - a_i| \tag{12}$$

### B. Error Estimation of the RBFN output

The output error $E_i$ corresponding to an argument $x_i$ is defined as the magnitude of the difference between the exact and estimated values (13).

$$E_i = |f(x_i) - f_{est}(x_i)| \tag{13}$$

In this experiment, MATLAB's inbuilt function evaluator output (the IEEE double precision floating-point format) was considered as the reference for error calculations. We define the average error percentage as shown in (14), by selecting a number of uniformly sampled points ($N_e$) within the desired argument interval $[\lambda_0, \lambda_1]$. The selected intervals for all the functions are tabulated in table 1.

$$Average\ Error\% = \frac{\sum_{i=1}^{N_e} E_i}{\sum_{i=1}^{N_e} f(x_i)} \times 100\% \tag{14}$$

$$x_i = \lambda_0 + (i-1)(\lambda_1 - \lambda_0)/(N_e - 1) \tag{15}$$

The error in the output vary with the number of hidden layer neurons of the network and the values of the synaptic weights. In order to achieve energy and performance benefits, we force the synaptic weights ($w_i$) to be powers of two, which results in some accuracy degradation. To realize this multiplier-less network structure, additional steps were introduced to the conventional training methodology of an RBFN. The training steps are elaborated in detail in the following section. Note that the entire training process is carried out, considering the word length restrictions of weights and bias values, to suit the digital implementation of the network. However, for better understanding, we do not discuss such restrictions in section III. They will be separately discussed in section IV, in detail.

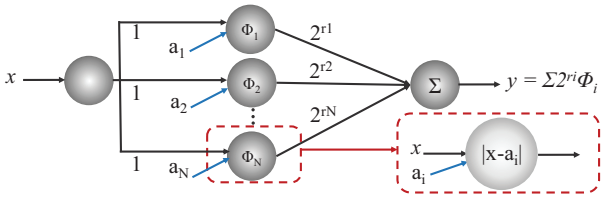


Fig. 3: Structure of the RBFN used in this work

### C. Training the RBFN

During the training phase, a discrete set of input-output pairs ($N_t$ number of pairs) is fed in to the RBFN. After training, the network is capable of estimating the output for an arbitrary input within the argument interval $[\lambda_0, \lambda_1]$. The proposed training process consists of two steps, namely, 'unrestricted exact interpolation' and 'restricted bias evaluation'.

*1) Unrestricted Exact Interpolation:* During this step, the weights are calculated for a given set of input-output pairs $[x_i, f(x_i)]$, without any restriction. The number of input-output pairs (training sets) are selected to be equal to the number of hidden layer neurons, $N_t$ to obtain the invertible RBF matrix $G$, which will be discussed momentarily. Initial training set data are selected by uniformly sampling the output range $[f(\lambda_0), f(\lambda_1)]$ into $N_t$ points for all the functions except for sin() and cos() functions. Unlike other functions, within our selected argument range, outputs for sin() and cos() functions neither continuously increase nor decrease, leading to complications while sampling the output. Therefore, for those two functions, the argument interval $[\lambda_0, \lambda_1]$ is uniformly sampled into $N_t$ points to obtain the initial training set. The argument $x_i$ of the training data set is used as the bias value $a_i$ of the $i^{th}$ hidden neuron (i.e. $a_i = x_i$). After the aforementioned initial selections, the corresponding synaptic weights are calculated using (16), where $b$ denotes the set of known values of the function $f$ in the training set ($f(x_i)$), and $w$ is the weight vector ($[w_1, w_2, , w_{N_t}]^T$). At the end of this step, the network is capable of evaluating the data points in the training set, with zero error, and the in between points with nonzero errors. This is the standard training process of an RBFN and it is known as 'exact interpolation'.

$$w = G^{-1} b \tag{16}$$

$$b = [f(x_1) ... f(x_{N_t})]^T \tag{17}$$

$$x_i = \begin{cases} \lambda_0 + \frac{(i-1)(\lambda_1 - \lambda_0)}{(N_t - 1)} & f = sin(), cos() \\ f^{-1}[f(\lambda_0) + \frac{(i-1)(f(\lambda_1) - f(\lambda_0))}{N_t - 1}] & otherwise \end{cases} \tag{18}$$

$$G_{i,j} = |x_i - a_j| \tag{19}$$

$$a_i = x_i \tag{20}$$

At this stage, the average error given by (14) is significant since the optimum bias and weight values are yet to be calculated. In order to find the optimum values, adjustments are made to the bias values $a_i$ and to the training data points $[x_i, f(x_i)]$. Each bias $a_i$ and argument $x_i$ is changed by a $\pm\Delta_1 k_1$ amount ($k_1 = 1, 2, ..., N_{adjust1}$), one at a time, and the average error is recorded. $\Delta_1$ is a small change and $\Delta_1 N_{adjust1}$ is the largest change made to $x_i$ or $a_i$. These changes are accepted if they reduce the average error of the output.

The power, delay and area values for each function evaluation method are compared at a target accuracy $A_t$ = 97%

(average accuracy = 100% - average error). The aforementioned adjustment process is repeated for each function, till the average accuracy does not improve any further. If this maximum average accuracy is below $A_t$, then the number of hidden layer neurons are increased by one and the process is repeated. If the maximum average accuracy of the network is well above ($+2\%$) $A_t$, the number of hidden layer neurons is reduced by one and the process is repeated. Step one ends after determining the weights, bias values, and the minimum size of the network that meets the accuracy constraint $A_t$.

*2) Restricted Bias Evaluation:* The second step of the training process is reverse calculation of the bias values for restricted weights. This step can be divided into three sub steps *viz.* 'weight replacement', 'synaptic weight adjustment', and 'bias evaluation'. During synaptic weight replacement, the weights obtained in step one were replaced with their closest powers of two (21). The powers can be either positive or negative leading to left or right shifting when evaluating the post synaptic states.

$$w_i = 2^{ri} \tag{21}$$
$$\text{with } (2^{ri-1} + 2^{ri-2}) \leq w_{i,step1} \text{ or } (2^{ri} + 2^{ri-1}) > w_{i,step1}$$
$$\text{and } ri = ... - 2, -1, 0, 1, 2, ...$$

where $w_{i,step1}$ is the $i^{th}$ synaptic weight obtained in step one. It is possible that the adjusted weights are not the ideal (that gives the minimum error) set of powers of two for the bias values obtained in step one. In order to check if there exists another set of weights that gives better accuracy, in weight adjustment sub step, each weight is individually changed to another power of two, and the change in accuracy is observed. If a new weight improves the accuracy, then that adjustment is accepted. After exhaustive adjustments to the weights, they are made fixed for the last step of the training process, which is bias evaluation. During bias evaluation, an optimization criterion is followed to find the set of bias values that gives the minimum average error corresponding to the calculated weights. Considering the nonlinear relationship between the average error and the bias values, we have selected the Nelder Mead optimization criteria [18] to identify the optimum bias values. The initial conditions for the Nelder Mead method are the $\hat{a}$ ($[a_1, a_2, ..., a_{N_t}]^T$) and $\hat{x}$ ($[x_1, x_2, ..., x_{N_t}]^T$) vectors determined in step one. At the end of optimization, a new $\hat{a}_n$ and a new $\hat{x}_n$ that offers a smaller error could be obtained. However, for different initial points, the optimization can converge to different solutions. Therefore, each element in $\hat{a}$ and $\hat{x}$ are first changed by a $\pm\Delta_2 k$ amount ($k = 1, ..., N_{adjust2}$) before applying the aforementioned optimization step. The changes are accepted if they reduce the average error at the networks output. This initial conditions adjustment is carried out until no further improvement on the average accuracy can be observed. At this point, if the network accuracy ($A_n$) is not close (within 0.5%) to $A_t$, the process should be repeated from step one with different network sizes. When repeating step one, the network size determining phase will be skipped since it is already being decided by step two.

## IV. Simulation Methodology

We analyzed the implementation of eight functions, namely sin(), cos(), exp(), log(), sinh(), cosh(), tanh() and sqrt(). Separate RBF networks were designed for each function, following the process mentioned in section III. The design process was done in MATLAB and all the accuracy calculations were carried out assuming the MATLAB inbuilt function evaluators output as the baseline. The representation used for the arguments ($x_i$) of the function and the bias values ($a_i$), employ a 16 bit fixed point fractional number system, with 11 integer bits and 5 fractional bits as shown in Fig. 4 (floating-point primitive is unattractive due to its high energy cost implementation). The most significant bit of the internal number system is a sign bit; '0' and '1' represent a positive and a negative number, respectively. Since the internal calculations require more number of bits, the output employs a 32 bit fixed point representation. For example, the resultant after multiplying a 16 bit wide number with $2^5$ require a 21 bit wide register.

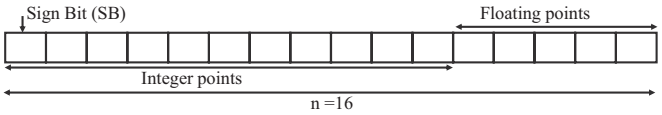

Fig. 4: Fixed point number system

The created MATLAB emulator is a bit-exact version of the actual hardware model. Finite precision effects for fixed-point can be effectively simulated within MATLAB by proper rounding after each arithmetic operation. However, for certain functions, rounding produces a zero synaptic weight when training the RBFN. This occurs when the synaptic weights are too small to be represented using the fixed point number system. In order to avoid this, we have introduced a pre-factor $F_{pre}$ for all the bias ($a_i$) and argument ($x_i$) values. $F_{pre}$ is chosen to be a negative power of two so that no multiplication is required for that step. This will increase each weight by a factor $1/F_{pre}$ as elaborated in (22) and (23):

$$G_{i,j} = |x_i - a_j| F_{pre} \tag{22}$$

$$w_{new} = G^{-1} b = w F_{pre}^{-1} \tag{23}$$

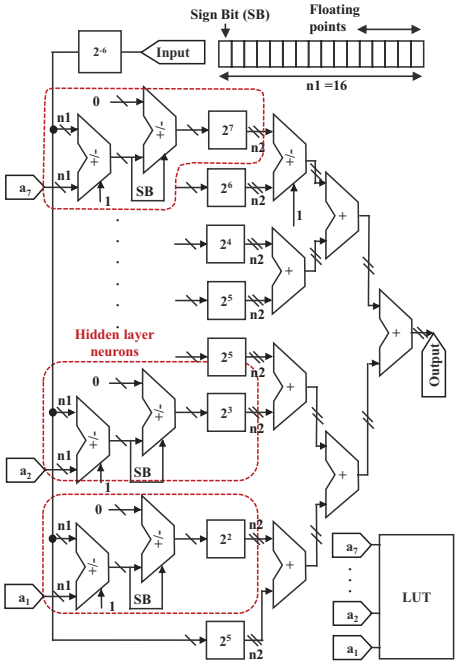

Fig. 5: Schematic of the RBFN that evaluates cosh() function.

The RBFN that evaluates cosh() function is illustrated in Fig. 5. The network consists of 8 neurons and the bias values are stored in an LUT. The pre factor is applied to the incoming arguments.

As mentioned earlier, in order to compare our proposed network, three commonly used function evaluating methods were selected. They are, LUT based linear interpolation, polynomial method and CORDIC. The accuracy of the aforementioned methods depends on the number of knots [8], the order of the polynomial [11], and the number of iterations [9] respectively. These parameters were changed in the MATLAB emulators (separate MATLAB emulators were designed for eight functions, per evaluation method), in such a way that all the function evaluating methods approximately (within $\pm 0.5\%$ of $A_n$) offer the same average accuracy as the RBFN based method. The proposed and existing methods were implemented in Register Transfer Level in Verilog, and mapped to the IBM 65nm technology library using Design Compiler to obtain power, delay and area values. Depending on the structure of each design, the memory requirement was estimated.

### A. Simulation Framework for Memory

All function evaluating methods discussed in this work require fetching data from the memory (LUT). Read power, leakage power, read latency and area are some of the important parameters associated with such LUTs. All these factors were considered in this analysis, assuming a 6T-SRAM of size 1Kibit. The 6T-SRAM was designed to have 208mV read noise margin and a 344mV of write margin in 65nm CMOS technology, under 1.1V nominal supply voltage conditions. The sense amplifier of choice was the full Complementary Positive Feedback Sense Amplifier [19].

## V. Results

### A. RBFN Size and Accuracy

The accuracy of the RBFN output can be improved by increasing the number of hidden layer neurons ('size'). Here we observe the performance, area and power tradeoffs of the RBFN with increasing accuracy. Fig. 6 (a) shows the variation of the highest achievable average accuracy of the RBFNs evaluating cosh(), sinh(), tanh(), exp(), log() and sqrt() functions, when the number of hidden layer neurons are increased from 3 to 12. According to Fig. 6 (a), the accuracy improves with the increasing network size. However, the rate of accuracy improvement reduces. Fig. 6 (b) illustrates the normalized area, power and delay values (as a ratio of the baseline values) for increasing number of hidden layer neurons. The baseline considered here is the RBFN with 3 hidden layer neurons.

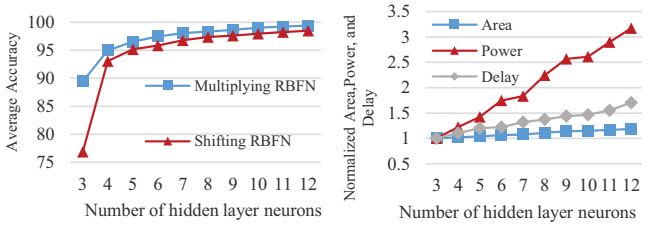


Fig. 6: (a) Average percentage accuracy of multiplying and shifting RBFNs, and (b) Normalized average power, delay and area of the shifting RBFN, for different number of hidden layer neurons. The RBFNs evaluate sinh(), cosh(), tanh(), exp(), log(), and sqrt() functions, and the average is elaborated here.

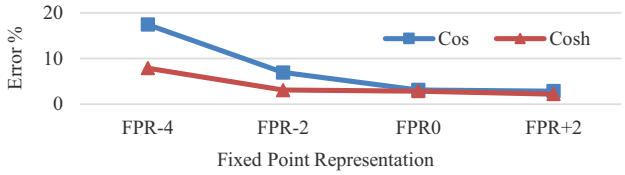


Fig. 7: Average error percentage of RBFNs that evaluate cos() and cosh() functions for different fixed point representations.

### B. Accuracy Degradation Due To Synaptic Weight Restrictions

The synaptic weights of the RBFNs in this design are constrained to powers of two. Such restrictions are applied to replace the multiplications required for function evaluation by bit shifting. When a multiplying RBFN (RBFN with unrestricted weights) is compared with a similar sized shifting RBFN (RBFN with restricted weights) which evaluates the same function, the multiplying network offers higher accuracy than the counterpart (Fig. 6 (a)). The difference between the accuracy values of the two types of RBFNs reduces as the number of hidden layer neurons are increased. From our results, it can be concluded that, by switching from multiplying RBFN to shifting RBFN (both with 7 hidden layer neurons), it is possible to obtain a 56.2% of power reduction and an 11.6% of delay reduction with an accuracy degradation of 1.3%, when evaluating functions.

### C. Word Length and Accuracy of the RBFN

We consider four fixed point arithmetic representations (FPR), each with different word lengths. First representation (FPR0) is the baseline which is 16 bits wide with 5 fractional points and 11 integer points (Fig. 4). Second representation (FPR$^+$2) has 7 fractional points and 11 integer points. Third (FPR$^-$2) and fourth (FPR$^-$4) representations have 11 integer points each, and 3 and 1 fractional points, respectively. Note that we have only changed the number of fractional points and the networks were trained with the restrictions applied. Since the output range of the functions considered in this work are not the same (table 1), the outcome of each function may have different effects when the number of fractional points are changed. For an example, as shown in Fig. 7, when we decrease the number of fractional bits, the accuracy degradation of *cos* function is larger than that of *cosh* function. This is due to the fact that, most of the outputs of the *cos* function have zero integer bits, and thus altering/removing one fractional bit will result in more percentage accuracy degradation than that of cosh function.

On average, there is a 14.3% power reduction and a 7.8% delay reduction, when we decrease the number of fractional bits by two from our baseline representation (FPR0).

### D. Proposed RBFN vs. Other Traditional Methods

In this section, we compare the power, delay, area, and memory requirements of each function evaluation circuit, for iso-accuracy conditions. All values presented are normalized with respect to the proposed RBFN. Fig. 8 (a) demonstrates the power requirements to evaluate each function using the four different methodologies. It is evident that the polynomial method consumes more power for most of the functions compared to other methods. Polynomial method involves finding higher orders of a given input [12]. Due to the higher number of multiplications associated with the polynomial method, the power

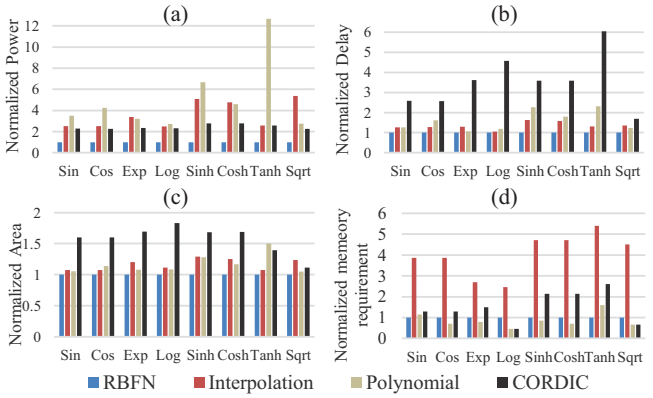


Fig. 8: Normalized (a) Power, (b) Delay, (c) Area, and (d) Memory requirement of different function evaluation algorithms and our proposed RBFN based method, when evaluating sin(), cos(), sinh(), cosh(), tanh(), exp(), log(), and sqrt() functions

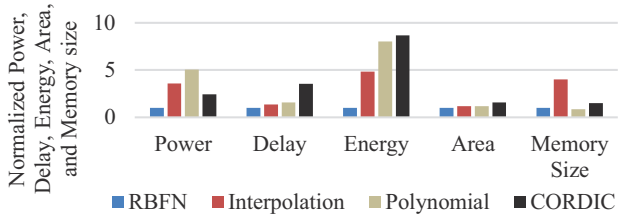


Fig. 9: The normalized power, delay, area, and memory requirements of different function evaluation methods, averaged over all the functions

consumption is relatively higher. However, for some functions like square root(), a smaller order polynomial is sufficient to achieve the target accuracy and thus the power consumption is lower. As Fig. 8 (b) illustrates, the delay for the CORDIC circuit is the largest for all the functions considered. This is due to the iterative nature of the algorithm [7][12]. Polynomial method offers the second largest delay for most of the functions since it requires more multiplications than other methods.

The relative area and memory requirements of different implementations are illustrated in Fig. 8 (c) and Fig. 8 (d), respectively. CORDIC, being an iterative process, consumes more area than other methods. For the linear interpolation circuit, the gradient values are also stored in the LUT along with the sampling knots to avoid redundant calculations. This results in higher memory requirement for the interpolation method compared to other methods. In contrast, the polynomial method requires smaller amount of memory for most of the functions since only the coefficients of the polynomial are stored [12]. For all the functions considered, the proposed RBFN offers power, area, energy, and delay benefits when compared with polynomial method, linear interpolation, and CORDIC (Fig. 9). Further, the proposed RBFN based method requires a smaller sized LUT with respect to linear interpolation and CORDIC.

We also observed the resource requirement to improve the accuracy of each function evaluation method. When improving the accuracy by 1% from the baseline (97%), other techniques show more than 4× power overhead, more than 1.6× delay overhead and more than 2× area overhead (averaged over all the functions) with respect to our proposed method.

## VI. Conclusion

Our proposed function evaluation method is an RBF network with a minimal sized LUT. The synaptic weights of the network were constrained to be powers of two (shifting RBFN) to eliminate any multiplications in the network. This was achieved through major modifications to the traditional training methods for RBFNs. Replacing multiplication with shifting makes the network evaluation process more efficient for digital implementation. However, the shifting RBFN offers lower accuracy than the multiplying RBFN. The accuracy of the network can be improved by increasing the number of neurons in the hidden layer. Increasing the number of fractional bits also improves the accuracy of the RBFN, albeit with energy cost. The digital implementation of the proposed RBFN based method consumes at least ∼2.5 times less power with ∼1.2 times less area, and is at least ∼1.4 times faster than traditional methods of evaluating elementary functions such as LUT based interpolation, CORDIC, and Polynomial approximations.

## Acknowledgment

The work was supported in part by, Center for Spintronic Meterials, Interfaces, and Novel Architectures (C-SPIN), a MARCO and DARPA sponsored StarNet center, by the Semiconductor Research Corporation, the National Science Foundation, Intel Corporation and DoD Vannevar Bush Fellowship.

## References

[1] H. J. Kwon et al.,"Luminance adaptation transform based on brightness functions for LDR image reproduction" *DSP, 30*, pp.74-85, July 2014
[2] A. Mukhtar et al., "Vehicle detection techniques for collision avoidance systems: A review", *IEEE Transactions on Intelligent Transportation Systems* 16.5 pp.2318-38, Oct 2015
[3] P. Kassanos et al., "An integrated analog readout for multi-frequency bioimpedance measurements, *IEEE Sensors Journal*. pp. 2792-800, 2014
[4] A. Jonghun et al., "A Reconfigurable Lighting Engine for Mobile GPU Shaders", *Journal Of Semiconductor Technology And Science*, 15.1 pp. 145-149, Feb 2015
[5] W. Cody, "Software Manual for the Elementary Functions. *Prentice-Hall series in computational mathematics*, Prentice-Hall, Inc Jul. 1980
[6] J. Pieiro et al., "Algorithm and architecture for logarithm, exponential, and powering computation, *IEEE Trans. on Computers*. pp. 1085-96, Sep. 2004
[7] M. Sadeghian et al., "Optimized Linear, Quadratic and Cubic Interpolators for Elementary Function Hardware Implementations, *Electronics*, 5.2 pp. 17, Apr 2016
[8] E. Meijering, "A chronology of interpolation: from ancient astronomy to modern signal and image processing. *Proc. of the IEEE*. pp. 319-342, 2002
[9] S. Aggarwal et al., "Concept, Design, and Implementation of Reconfigurable CORDIC, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24.4 pp. 1588-92, Apr 2016
[10] Muller, J.M., "Elementary Functions: Algorithms and Implementation, 2nd ed. 2006
[11] A. Strollo et al., "Elementary functions hardware implementation using constrained piecewise-polynomial approximations, *IEEE Transactions on Computers*, pp. 418-432, Mar. 2011
[12] D. Lee et al., "Hardware implementation trade-offs of polynomial approximations and interpolations, *IEEE Transactions on Computers*. pp. 686-701, May 2008
[13] Y. Lei et al., "Generalization performance of radial basis function networks, *IEEE transactions on neural networks and learning systems*, 26.3 pp. 551-64, Mar. 2015
[14] R. Dash et al., "A comparative study of radial basis function network with different basis functions for stock trend prediction, *PCITC* pp. 430-435, Oct. 2015
[15] D. Broomhead et al., "Radial basis functions, multi-variable functional interpolation and adaptive networks, *Royal signals and radar establishment Malvern*, Mar. 1988
[16] R. Pachn et al., "Barycentric-Remez algorithms for best polynomial approximation in the chebfun system, *BIT Numerical Mathematics*, 49.4 pp. 721-41, Dec. 2009
[17] X. Hu et al., "Expanding the range of convergence of the CORDIC algorithm, *IEEE Transactions on Computers*, pp. 13-21 Jan. 1991
[18] J. Nelder et al., "A simplex method for function minimization, *The computer journal*, pp. 308-313, Jan. 1965
[19] J. Rabaey et al., "Digital integrated circuits (Vol. 2). Englewood Cliffs: Prentice hall, Dec. 2002