electronics

MDPI

Article

An Efficient Hardware Architecture with Adjustable Precision and Extensible Range to Implement Sigmoid and Tanh Functions

Hui Chen [1], Lin Jiang [1], Heping Yang [1], Zhonghai Lu [2], Yuxiang Fu [1,*] and Li Li [1,*] and Zongguang Yu [1]

[1] School of Electronic Science and Engineering, Nanjing University, Nanjing 210023, China; winnerchan@smail.nju.edu.cn (H.C.); jianglin@smail.nju.edu.cn (L.J.); peace_yang@smail.nju.edu.cn (H.Y.); yuzg58@163.com (Z.Y.)

[2] The KTH Royal Institute of Technology, Kista, 16440 Stockholm, Sweden; zhonghai@kth.se

* Correspondence: yuxiangfu@nju.edu.cn (Y.F.); lili@nju.edu.cn (L.L.); Tel.: +86-138-5158-4190 (Y.F.); +86-189-5167-9299 (L.L.)

Received: 28 September 2020; Accepted: 15 October 2020 ; Published: 21 October 2020

Abstract: The efficient and precise hardware implementations of tanh and sigmoid functions play an important role in various neural network algorithms. Different applications have different requirements for accuracy. However, it is difficult for traditional methods to achieve adjustable precision. Therefore, we propose an efficient-hardware, adjustable-precision and high-speed architecture to implement them for the first time. Firstly, we present two methods to implement sigmoid and tanh functions. One is based on the rotation mode of hyperbolic CORDIC and the vector mode of linear CORDIC (called RHC-VLC), another is based on the carry-save method and the vector mode of linear CORDIC (called CSM-VLC). We validate the two methods by MATLAB and RTL implementations. Synthesized under the TSMC 40 nm CMOS technology, we find that a special case $AR\mid_V^R(3,0)$, based on RHC-VLC method, has the area of 4290.98 μm^2 and the power of 1.69 mW at the frequency of 1.5 GHz. However, under the same frequency, $AR\mid_V^C(3)$ (a special case based on CSM-VLC method) costs 3196.36 μm^2 area and 1.38 mW power. They are both superior to existing methods for implementing such an architecture with adjustable precision.

Keywords: sigmoid and tanh; hyperbolic and linear CORDIC; carry-save method; efficient hardware; adjustable precision; high speed

1. Introduction

Artificial neural networks (ANNs) are widely used in the applications of pattern recognition, image classification, biological systems and so on [1]. In the past, ANNs were generally implemented only in software. But in recent years, with the development of artificial intelligence and integrated circuits, their hardware implementations have become more and more important because of the performance gains compared with software implementations. In ANNs, each layer or component is important, mainly including convolution, pooling, full connection and activation function. The activation functions can introduce nonlinear factors to neurons, so that the ANNs can approximate any nonlinear functions and be applied to nonlinear models. Therefore, an efficient and accurate implementation of non-linear activation functions, such as *tanh* and *sigmoid*, is of high interest.

The *tanh* and *sigmoid* functions both shape like an "S" curve, where the outputs of *tanh* vary between $(-1, 1)$ and the range of *sigmoid* outputs is $(0, 1)$. Mathematically, their functions are defined below ($S(x) \rightarrow$ *sigmoid* function, $T(x) \rightarrow$ *tanh* function).

$$S(x) = \frac{1}{1+e^{-x}}, \; T(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}. \tag{1}$$

In fact, the *tanh* function can be converted to:

$$T(x) = \frac{e^{x}-e^{-x}+e^{-x}-e^{-x}}{e^{x}+e^{-x}} = 1 - \frac{2e^{-x}}{e^{x}+e^{-x}}. \tag{2}$$

By dividing numerator and denominator by e^{-x}, Equation (2) can be rewritten as:

$$T(x) = 1 - \frac{2}{1+e^{2x}} = 1 - 2S(-2x). \tag{3}$$

It follows that they are functionally related, as above, which enables them to be implemented in a unified architecture. From the formulas, we know that the exponential terms are the key factors that generate the nonlinear behavior. Meanwhile, the accuracy and hardware cost of exponential and division operations can influence the performance of whole neural networks [2,3]. Up to now, some methods have been put forward to solve their implementation problems and can be divided into the following types: look-up tables (LUT) or range addressable LUT (RALUT)-based methods [4–7], piecewise linear (PWL)-based methods [8–11], piecewise nonlinear (PWNL)-based methods [12,13] and Taylor series (TS)-based methods [14].

The LUT- or RALUT-based methods take the advantage of look-up tables to store large amounts of points or range values, respectively. Take the LUT-based method as an example it approximates *tanh* or *sigmoid* functions with a finite number of distributed points. In general, these points are evenly distributed, or not evenly distributed. The number of points is related to the number of bits used in hardware implementation and it determines the maximum error or average error of results. The biggest advantage of this method is the short latency and it usually just needs one clock cycle. However, its biggest disadvantage is that it requires a lot of memory to store the lookup table. The larger the input range of the computation, or the higher the precision of the results, the larger the memory requirements, which is obviously not desirable in hardware implementations.

Similarly, PWL-based methods use some straight lines to simplify the nonlinear curves of the functions, and the accuracy depends on the number of lines. Compared with LUT- or RALUT-based methods, this method does not store massive data and requires less memory. However, the essence of transforming non-linear behavior into linear behavior makes its accuracy not very high. Especially at the inflection point of the function curve, the number of segments needs to be increased to obtain better accuracy. More segmentations means more storage, and more complex judgment logic. On the other hand, its fitting linear segment formula is as follows: $y = k_1 + k_2 \times x$, where one multiplier is used. As we know, the multiplier is complicated in hardware implementation because of the negative effect on circuit throughput and area utilization. In many designs, it is difficult to accelerate the working frequency because of the existence of the multiplier. Therefore, it is preferable to avoid using multipliers.

By contrast, PWNL-based methods use nonlinear approximation in each segment. Similar to the PWL-based method, it is fitted by the following nonlinear formula: $y = k_1 + k_2 \times x + k_3 \times x^2 + \cdots + k_n \times x^{n-1}$, where the nth order needs $(n+1)n/2$ multiplication operations. The main advantage of this method is that it can achieve higher precision than the PWL-based method. However, a lot of multiplications result in a low working frequency for the whole architecture [12].

The TS-based method faces the same challenges as LUT- or RALUT-based methods. Its accuracy can not be very high because it is limited by the chosen range.

Therefore, it is difficult for each of the above methods to achieve all the advantages of computing *tanh* and *sigmoid* functions: high hardware efficiency, adjustable precision, extensible input range, and high computation speed. If we use these methods to build such an hardware architecture, they will be very costly. However, when designing a universal reconfigurable accelerator (such as RASP [15]), it usually includes many operators, such as *tanh* and *sigmoid*. If the operator has adjustable precision and the above advantages, it will be very useful. Thus, our proposed architectures to implement *tanh* and *sigmoid* have important application value. On the whole, we make the following contributions:

- For the first time, we propose a hardware architecture with adjustable precision and extensible input range to implement *tanh* and *sigmoid* functions, which is based on the RHC-VLC method. In addition, we propose another architecture with unlimited input range and adjustable accuracy, which is based on CSM-VLC method.
- Of the two proposed methods, the accuracy magnitude based on the CSM-VLC method can reach 10^{-4} best, while the magnitude of the accuracy based on the RHC-VLC method can be much better, such as 10^{-7} or 10^{-8}. At the same time, the proposed methods can change the accuracy by adjusting the iterations of CORDIC without changing the current computing architecture. The lower the accuracy requirement, the lower the overall latency. Other methods of adjusting the precision require architectural changes, which are extremely unfriendly in hardware implementation.
- In hardware implementation, the RHC-VLC-based method only requires shift-and-add (or subtract) operations. Another method based on CSM-VLC requires a constant multiplier, we also optimize it to achieve shorter delay and smaller area. Compared with other existing methods to implement a hardware architecture with adjustable precision, our hardware implementation is more efficient. The proposed architecture has dual computing capabilities (*tanh* and *sigmoid*), which can be determined by the input selection bit.
- Under TSMC 40 nm CMOS technology, the hardware architecture based on our proposed methods can work at 1.5 GHz frequency or even higher. Except for the high frequency, our methods are also compatible with other advantages: efficient hardware, adjustable precision, and extensible input range. With the same adjustable range of precision, our architecture has lower area and power consumption compared with other methods.

The rest of this paper is organized as follows. Section 2 details the first proposed architecture based on the RHC-VLC method to implement *tanh* or *sigmoid* functions. Section 3 presents another proposed architecture based on the CSM-VLC method. Section 4 shows the general hardware implementation and a specific case of these two methods. Then, we make a comparison with other methods. Finally, Section 5 draws the conclusions.

2. Proposed Architecture Based on RHC-VLC Method

To solve the problem of not being able to build a hardware architecture that has all of the above four advantages, we propose two solutions: The RHC-VLC-based method and CSM-VLC-based method. In this section, we first detail the architecture based on the RHC-VLC-method, including basic theory, method process, software simulation and algorithm construction. Another architecture based on the CSM-VLC method will be presented in the next section.

2.1. Overview of RHC and VLC

Before introducing the first method, we need to review what RHC and VLC are. Only based on this theory can we build an architecture with adjustable precision and extensible input range to compute *sigmoid* and *tanh* functions.

In 1959, Volder invented CORDIC to evaluate trigonometric functions, multiplication and division [16]. Later, Walter extended its computation capacities to compute exponentials, logarithms

and square roots [17]. In this paper, only VLC and RHC are adopted, so we omit the introduction to other types of CORDIC. Their iterative formulas can be found in [18].

$$RHC: x_{k+1} = x_k + \zeta \cdot (2^{-k} y_k),\ y_{k+1} = y_k + \zeta \cdot (2^{-k} x_k),\ z_{k+1} = z_k - \zeta \cdot tanh^{-1}(2^{-k}), \tag{4}$$

where $\zeta = sign(z_k)$, $k \geq 1$ and k are integers. Especially, when the iteration number $k = 4, 13, 40, \ldots, n$, one more iteration of RHC should be required. Otherwise, RHC cannot converge.

$$VLC: x_{k+1} = x_k,\ y_{k+1} = y_k - \eta \cdot (2^{-k} x_k),\ z_{k+1} = z_k + \eta \cdot 2^{-k}, \tag{5}$$

where $\eta = sign(y_k)$, $k \geq 0$ and k is an integer.

After a few iterations, RHC and VLC will converge to some common functions, which are shown in Table 1.

Table 1. Outputs of RHC and VLC.

Mode of CORDIC	Outputs
RHC	$x_n = \Gamma(x_0 \cdot coshz_0 + y_0 \cdot sinhz_0)$
	$y_n = \Gamma(y_0 \cdot coshz_0 + x_0 \cdot sinhz_0)$
	$z_n = 0$
VLC	$x_n = x_0$
	$y_n = 0$
	$z_n = z_0 + y_0/x_0$

Γ is computed by $\prod_{k=1}^{n}(\sqrt{1-2^{-2k}})$ and called the scale-factor, where the terms ($k = 4, 13, 40, \ldots$) should be multiplied again. When the iteration number of RHC is determined, Γ will be a constant value, so we do not need to compute it in actual applications.

From Table 1, we know that RHC has the ability of calculating hyperbolic sine (*sinh*) and hyperbolic cosine (*cosh*) if we initialize the inputs as $x_0 = 1/\Gamma$ and $y_0 = 0$, VLC can implement the division operation if z_0 is initialized to be 0. They all lay the foundation to compute the *sigmoid* and *tanh* functions.

2.2. The RHC-VLC-Based Method

This method of computing $S(x)$ contains two steps. The first step deals with the exponential operation of e^{-x}, which can be implemented by RHC. Because

$$cosh(x) + sinh(x) = e^x, \tag{6}$$

we can initialize the inputs of RHC as follows:

$$x_0 = 1/\Gamma,\ y_0 = 0,\ z_0 = -x. \tag{7}$$

Then, we can get the result of e^{-x} through x_n plus y_n.

The second step handles the division operation, which can be calculated by VLC. To this end, the inputs of VLC are initialized to

$$x_0 = 1 + e^{-x},\ y_0 = 1,\ z_0 = 0, \tag{8}$$

so the output z_n will converge to $\frac{1}{1+e^{-x}}$ and it is exactly the result of $S(x)$. From the relationship in Equation (1), we can design the architecture shown in Figure 1 to calculate $S(x)$ or $T(x)$. "t" represents

the functionality to be implemented by the current architecture, $t = 0$ means computing $S(x)$, and $t = 1$ stands for computing $T(x)$.

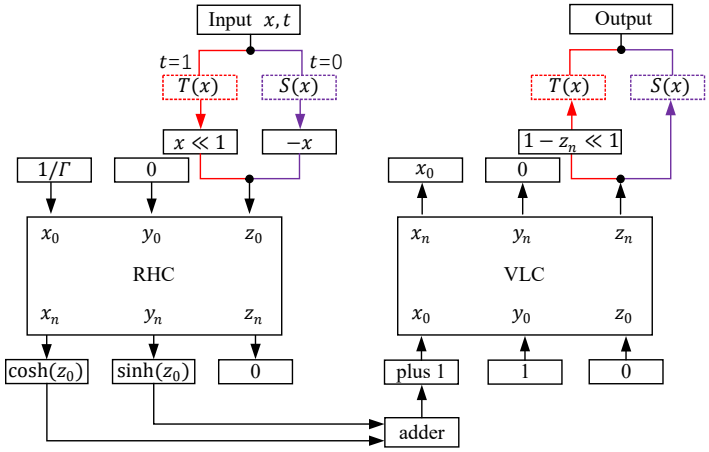

Figure 1. The architecture of computing $S(x)$ and $T(x)$ based on RHC and VLC. The red path is for computing $T(x)$, the purple path is for computing $S(x)$, and the black path is shared.

Since the convergence range of RCH or VLC is limited, we should analyze the range of the input variable x of computing $S(x)$ and $T(x)$ (called $S\&T$). As can be intuitively seen from Figure 1, the convergence range of RHC is affected by VLC and the range of input x is determined by RHC.

As for the standard CORDIC, the convergence range of RHC is $|z_0| \leq 1.1182$, and that of VLC is $\frac{y_0}{x_0} \leq 1$ [19]. Because the output of $T(x)$ varies between $[-1, 1]$ and the $S(x)$ output varies between $[0, 1]$, their ranges happen to be suitable for VLC, and we do not need to expand the converge range of VLC. Then, we analyze the range of the variable x when we use RHC to compute e^{-x} or e^{2x}. x belongs to $[-1.1182, 1.1182]$ for computing $S(x)$ and x only belongs to $[-0.5591, 0.5591]$ for computing $T(x)$, which are both very limited and not suitable for practical applications.

Therefore, we should expand the range of input variables z_0 of RHC. Hu [19] shows that, if we extend the iteration index set from $k = 0, 1, \ldots, n$ to $k = -m, -m+1, \ldots, 0, \ldots, n$, the convergence range of RHC will be expanded. However, the iterative formulas will be changed when $k \leq 0$, where the terms containing 2^{-k} need to be replaced by $1 - 2^{-2^{-k+1}}$. That is

$$\begin{gathered} x_{k+1} = x_k + sign(z_k)(1 - 2^{-2^{-k+1}} y_k),\ y_{k+1} = y_k + sign(z_k)(1 - 2^{-2^{-k+1}} x_k), \\ z_{k+1} = z_k - sign(z_k) tanh^{-1}(1 - 2^{-2^{-k+1}}), \end{gathered} \tag{9}$$

The convergence range of the expanded RHC depends on the maximum sum of rotation angles, which is defined as

$$\theta_{max}^{(m,n)} = \sum_{k=-m}^{0} tanh^{-1}(1 - 2^{-2^{-k+1}}) + \sum_{k=1}^{n} tanh^{-1}(2^{-k}), \tag{10}$$

where the terms of iterative number $k = 4, 13, 40, \ldots$ should be accumulated twice.

Ulterior to this, the convergence range of RHC will be updated to $z_0 \leq \theta_{max}^{(m,n)}$, and we can enlarge or reduce the range through changing the value of m. For example, n is assumed to be 15, then we can change the value of m to get different convergence ranges of RHC, including the corresponding input ranges for $T(x)$ and $S(x)$, as shown in Table 2.

In consideration of the iterations of the expanded RHC are related to two parameters (let them be m and n, $m \leq 0$, $n \geq 1$) and the iterations of VLC are relevant to one parameter (let it be p, $p \geq 1$), we can denote the architecture of computing $S\&T$ as $Ar_{VLC}^{RHC}(m, n, p)$.

Table 2. An Extension of the Input Ranges.

m	$\theta_{max}^{(m,15)}$	x for $S(x)$	x for $T(x)$
0	2.028	$[-2.028, 2.028]$	$[-1.014, 1.014]$
1	3.745	$[-3.745, 3.745]$	$[-1.872, 1.872]$
2	6.863	$[-6.863, 6.863]$	$[-3.431, 3.431]$
3	12.755	$[-12.755, 12.755]$	$[-6.377, 6.377]$
4	24.192	$[-24.192, 24.192]$	$[-12.096, 12.096]$
5	$\to \infty$	$\to \infty$	$\to \infty$

2.3. Software Test for the RHC-VLC-Based Method

Before we go further and design an architecture with adjustable precision, it is necessary to conduct a software simulation for validating and exploring the accuracy of computing $S\&T$. First, we input different data x to verify the correctness of $S\&T$. Then, we explore the accuracy distribution of computing $S\&T$ through changing the iterations of VLC and RHC.

In order to evaluate the accuracy of the proposed architecture, we characterize it with maximum absolute error (MAE) and average absolute error (AAE) and they are defined as:

$$MAE = max(|V_{ori} - V_{pro}|_i),\ AAE = \frac{\Sigma(|V_{ori} - V_{pro}|_i)}{NUM}, \tag{11}$$

where $1 \leq i \leq NUM$, V_{ori} stands for the value of original functions, V_{pro} represents the value of proposed architecture, and NUM means the total test points.

Since the accuracy of RHC and VLC is closely related to the number of iterations, we can first explore the distribution of the respective accuracy of RHC and VLC. All cases set NUM to be 100,000 and the input x is generated by a random number. (Figures 2 and 3 both use logarithmic ordinates.)

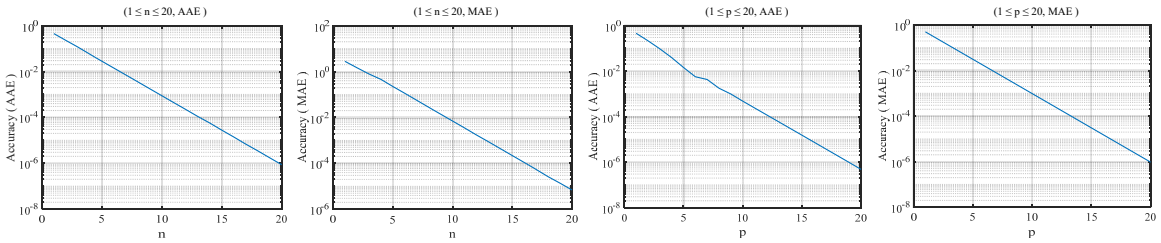

Figure 2. RHC & VLC: Accuracy distribution corresponding to different n or p.

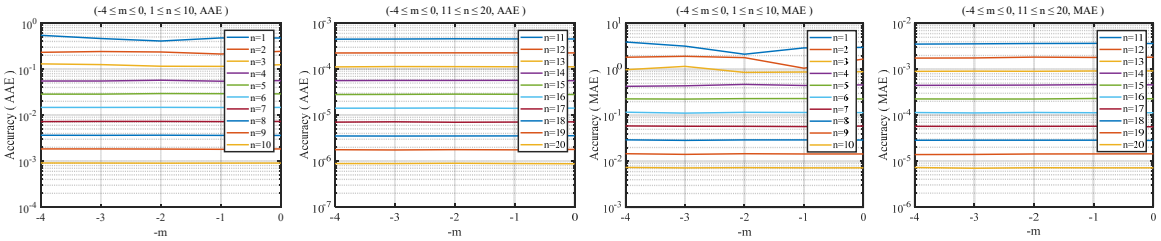

Figure 3. RHC: Accuracy distribution corresponding to different m.

The first case: We set the number of iterations (n) of RHC to range from 1 to 20 and the input range is $[-1.118, 1.118]$. Since all iteration numbers are positive, each iteration of RHC will adopt the iterative formulas, described by Equation (4). As can be seen from Figure 2, when n is greater than 10, the order of magnitude of AAE changes from -4 to -7. The magnitude of MAE varies from -4 to -6 when n is greater than 11. The more times of positive iterations, the higher the accuracy.

The second case: We set the number of negative iterations (m) of RHC to vary from 0 to 4 and the input range is $[-2.028, 2.028]$. The maximum positive iteration number (n) ranges from 1 to 20. The terms with positive iteration numbers still adopt the iterative formulas, shown as Equation (4),

but those with negative iterations will use Equation (9). From Figure 3, we can see that the negative iterations basically do not affect the accuracy of RHC and the order of magnitude of AAE or MAE is the same as Figure 2.

The third case: In order to evaluate the computational accuracy of VLC, we set the number of iterations (p) to scale from 1 to 20 and the input range is [1, 100]. The iterative formulas used in the simulation are shown in Equation (5). The following conclusions can be drawn from Figure 2: The larger the p value, the higher the calculation accuracy of VLC. When $p \geq 9$, the order of magnitude of AAE is smaller than -4. However, when MAE reaches the same magnitude, p must be greater than 10.

Next, based on the accuracy of the above two computation modules—RHC and VLC—we use the control variable method to explore the precision distribution of computing $S\&T$. All cases set NUM to be 100,000 and the input x is generated by random number. From Figure 3, we know that the parameter m of RHC has little effect on the accuracy, so we set it to 0 first. In this case, the accuracy of computing $S\&T$ is only affected by the iterations n and p. In order to obtain n and p, corresponding to different magnitudes of MAE and AAE, we divide the discussion into three cases. (Input range x: $S(x) \rightarrow [-2, 2]$ and $T(x) \rightarrow [-1, 1]$.)

Case 1: $m = 0, 1 \leq n \leq 20, 1 \leq p \leq 7$. The simulation result is shown in Figures 4 and 5. They tell us that, for the same iterations (n and p), the accuracy of *tanh* is slightly lower than that of *sigmoid*. Then, when $3 \leq p \leq 7$ and $1 \leq n \leq 10$, the accuracy is improved with the increase in p. Finally, if p is small and n is large, the precision stays pretty much the same.

Case 2: $m = 0, 1 \leq n \leq 20, 8 \leq p \leq 14$. The simulation result is shown in Figures 6 and 7. Similar to Case 1, when $8 \leq p \leq 14$, no matter how n varies in $1 \leq n \leq 10$, the accuracy remains basically the same. However, when n is increased in the range of $11 \leq n \leq 20$, the accuracy is greatly improved. On the whole, the precision under the condition of Case 2 is better than that of Case 1.

Case 3: $m = 0, 1 \leq n \leq 20, 15 \leq p \leq 21$. The simulation result is shown in Figures 8 and 9. Except for some features common to Case 1 and Case 2, the overall accuracy of Case 3 is higher than that of Case 1 and Case 2 (n in the same range). As p increases, even if n is small or large, the accuracy remains basically the same. As n and p both increase gradually, the precision tends to be of the same magnitude.

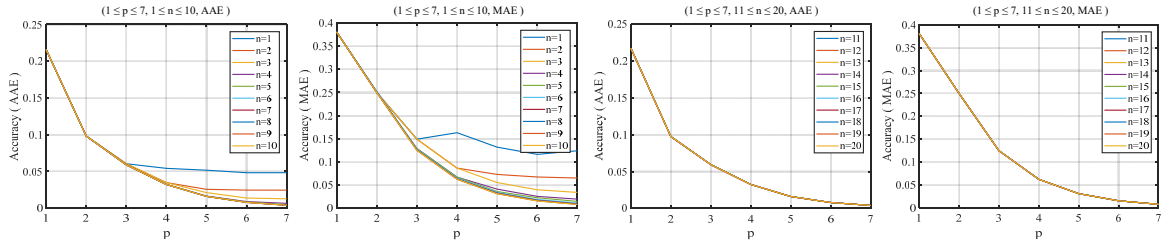

Figure 4. *Sigmoid*: Accuracy distribution corresponding to Case 1.

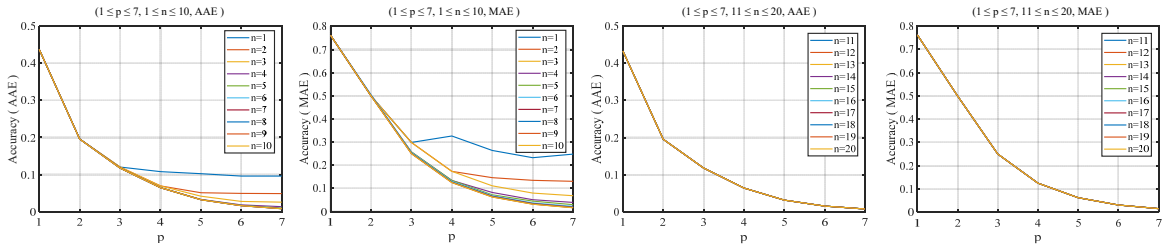

Figure 5. *Tanh*: Accuracy distribution corresponding to Case 1.

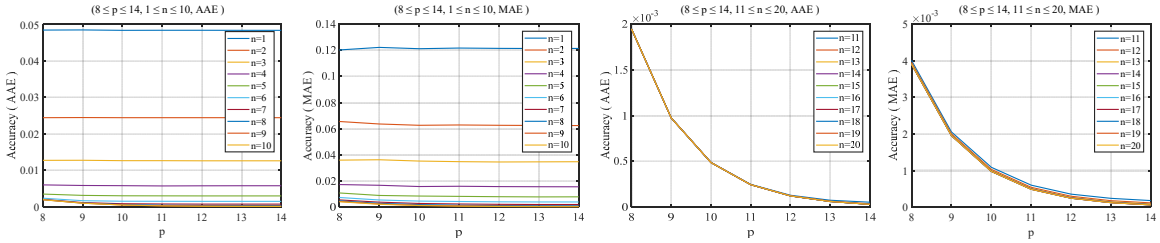

Figure 6. *Sigmoid*: Accuracy distribution corresponding to Case 2.

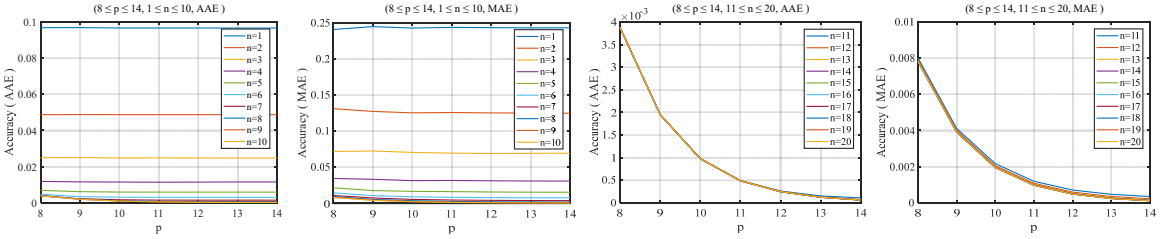

Figure 7. *Tanh*: Accuracy distribution corresponding to Case 2.

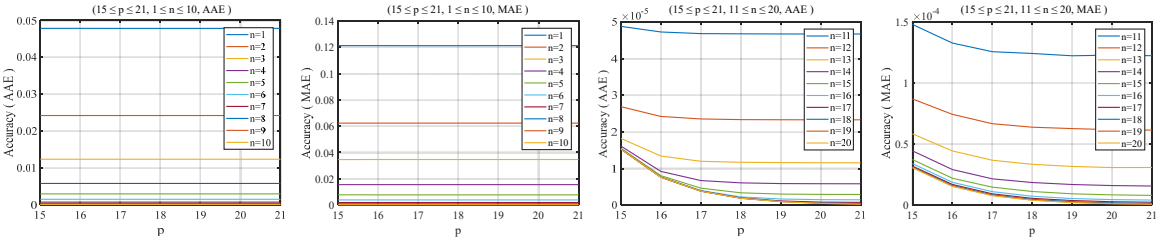

Figure 8. *Sigmoid*: Accuracy distribution corresponding to Case 3.

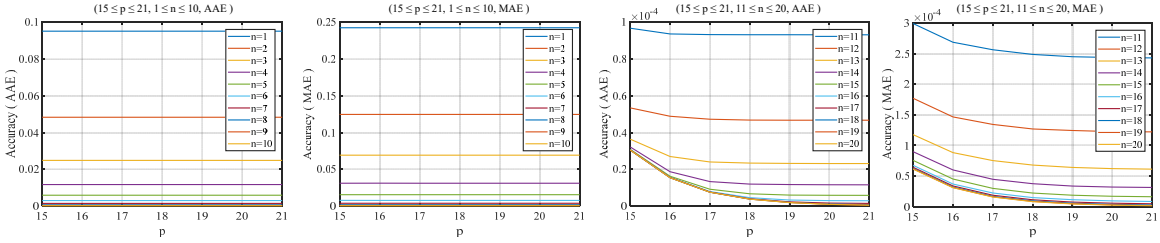

Figure 9. *Tanh*: Accuracy distribution corresponding to Case 3.

From the above experimental results, it can be seen that the computation accuracy of *S&T* is strongly associated with the combination of RHC and VLC iterations. As long as the iterations of RHC and VLC are determined, the output precision of these two activation functions will be determined accordingly. In this way, the architecture based on the RHC-VLC method is not only with adjustable precision, but is also easy to implement and extend in hardware.

2.4. Proposed Algorithm with Adjustable Precision

Through the simulation verification in the previous subsection, we know the relationship between the computation precision of *S&T* and the iterations of RHC and VLC. Since different precision magnitudes correspond to multiple combinations of RHC and VLC iterations, we further classify and screen them according to a principle of fewer total iterations (Principle: AAE and MAE are expressed as "$K \times 10^E$", $K \in (1, 5]$ and E is a negative integer), as shown in Tables 3 and 4. Here, only partial results corresponding to the order of magnitude of precision are shown. If the magnitude is less than -6, it can be obtained according to the above method, which will not be detailed in this paper.

Table 3. Sigmoid: Relationship Between the Magnitudes of Accuracy and Iterations.

Accuracy Magnitude	Accuracy Type	RHC&VLC (m,n,p)	K $Value$	MTI [1] $m+n+p$
$E=-2$	AAE	(0,2,4)	3.49	6
	MAE	(0,3,6)	3.99	9
		(0,4,5)	4.13	
$E=-3$	AAE	(0,5,7)	4.84	12
	MAE	(0,8,8)	4.77	16
		(0,9,7)	3.78	
		(0,10,6)	4.53	
$E=-4$	AAE	(0,8,11)	4.33	19
	MAE	(0,10,12)	4.72	22
$E=-5$	AAE	(0,11,15)	4.77	26
		(0,12,14)	3.59	
	MAE	(0,14,15)	4.51	29

(1) MTI: Minimum Total Iterations. (2) Assuming $m=0$; Input Range: $[-2, 2]$.

Table 4. Tanh: The Corresponding Relationship between the Magnitudes of Accuracy and Iterations.

Accuracy Magnitude	Accuracy Type	RHC&VLC (m,n,p)	K $Value$	MTI $m+n+p$
$E=-2$	AAE	(0,2,6)	4.95	8
		(0,3,5)	4.18	
	MAE	(0,4,7)	3.39	11
		(0,5,6)	4.49	
$E=-3$	AAE	(0,6,8)	4.75	14
	MAE	(0,8,10)	3.84	18
		(0,9,9)	4.72	
$E=-4$	AAE	(0,9,12)	4.35	21
	MAE	(0,11,13)	4.78	24
$E=-5$	AAE	(0,12,16)	4.77	28
		(0,13,15)	3.63	
	MAE	(0,15,16)	4.48	31

(1) Assuming $m=0$; Input Range: $[-1, 1]$.

Next, according to the Sections 2.2 and 2.3, we propose an algorithm (Algorithm 1) to compute $S\&T$ with adjustable precision, which is called the $RVST$ algorithm in this paper. It can be seen from Tables 3 and 4, that the accuracy of the same magnitude may correspond to multiple iterative combinations, and it is better to take n smaller and p larger than the criterion. The reason is that according to Equations (4) and (5), the hardware implementation of VLC is less complex than that of RHC. Therefore, the combinations of m, n and p in the $RVST$ algorithm are determined and they are also the best combinations. The input range is shown in Table 2 and it varies with the value of m. However, the value of m has little effect on the accuracy. In the $RVST$ algorithm, MAE is used as the accuracy standard. Besides, AAE can also be adopted.

Algorithm 1 *RVST* algorithm.

Input: The parameter x that needs to be calculated; the calculated type t ($t = 0 \Rightarrow S(x)$ and $t = 1 \Rightarrow T(x)$); the required order of magnitude (rm) of MAE (all are positive numbers, for example, 2 actually represents "-2"); m_i determines the input range of x

Output: Final result FR

if $t = 0$ **then**
  if $rm = 2$ $m = m_i, n = 3, p = 6$
  else if $rm = 3$ $m = m_i, n = 8, p = 8$
  else if $rm = 4$ $m = m_i, n = 10, p = 12$
  else if $rm = 5$ $m = m_i, n = 14, p = 15$
  else $m = 0, n = 0, p = 0$
  Return $FR = S(x, m, n, p)$
else
  if $rm = 2$ $m = m_i, n = 4, p = 7$
  else if $rm = 3$ $m = m_i, n = 8, p = 10$
  else if $rm = 4$ $m = m_i, n = 11, p = 13$
  else if $rm = 5$ $m = m_i, n = 15, p = 16$
  else $m = 0, n = 0, p = 0$
  Return $FR = T(x, m, n, p)$
end if

As can be seen from Algorithm 1, in addition to the conditional statements, there are two functions: $S()$ and $T()$ (more on that below). In general, the algorithm is clear and simple in hardware implementation.

Further, we introduce the algorithm of function $S()$. The principle of the algorithm is based on Figure 1 and it is termed as S algorithm. It should be noted that the values of $1/\Gamma$ in Figure 1 are different under different iterations of RHC, as shown in Table 5. Another thing to note is that when RHC contains non-positive indexed iterations, the new scale-factor Γ should be redefined as

$$\prod_{k=-m}^{0}(\sqrt{1-(1-2^{-2^{-k+1}})^2}) \cdot \prod_{k=1}^{n}(\sqrt{1-2^{-2k}}) \tag{12}$$

Table 5. The Parameter Value $1/\Gamma$ in S Algorithm.

No.	(m, n)	$1/\Gamma$	*Value*
1	(0,3)	$0.66143783 \cdot \prod_{k=1}^{3}(\sqrt{1-2^{-2k}})$	0.55028235384
2	(0,4)	$0.66143783 \cdot \prod_{k=1}^{4}(\sqrt{1-2^{-2k}})$	0.54920653198
3	(0,8)	$0.66143783 \cdot \prod_{k=1}^{8}(\sqrt{1-2^{-2k}})$	0.54885034814
4	(0,10)	$0.66143783 \cdot \prod_{k=1}^{10}(\sqrt{1-2^{-2k}})$	0.54884903958
5	(0,11)	$0.66143783 \cdot \prod_{k=1}^{11}(\sqrt{1-2^{-2k}})$	0.54884897415
6	(0,14)	$0.66143783 \cdot \prod_{k=1}^{14}(\sqrt{1-2^{-2k}})$	0.54884895268
7	(0,15)	$0.66143783 \cdot \prod_{k=1}^{15}(\sqrt{1-2^{-2k}})$	0.54884895243

(1) Assuming $m = 0$; (2) Input Range: $[-2, 2]$.

Similarly, we can derive the algorithm for the function $T()$ from Figure 1, which is called T algorithm. In the hardware implementation, multiplying by 2 is equivalent to moving one bit to the left. For RHC and VLC, only shift-add (or subtract) operations are required. Therefore, Algorithms 1–3, there is no complicated hardware logic, which ensures the efficiency of the whole architecture. Specific hardware implementation of the proposed architecture is detailed in Section 4.

Algorithm 2 *S* algorithm.

Input: The parameter x that needs to be calculated; m, n, and p correspond to the iterations of RHC and VLC, respectively
Output: $S()$ result SR

$R_{y0} = 0$
$R_{z0} = -x$
if $rm = 2$ $R_{x0} = 0.55028235384$
else if $rm = 3$ $R_{x0} = 0.54885034814$
else if $rm = 4$ $R_{x0} = 0.54884903958$
else if $rm = 5$ $R_{x0} = 0.54884895268$
else $R_{x0} = 0$
$(R_{xn}, R_{yn}, R_{zn}) = RHC(R_{x0}, R_{y0}, R_{z0}, m, n)$
$V_{x0} = R_{xn} + R_{yn} + 1$, $V_{y0} = 1$, $V_{z0} = 0$
$(V_{xp}, V_{yp}, V_{zp}) = VLC(V_{x0}, V_{y0}, V_{z0}, p)$
Return $SR = V_{zp}$

Algorithm 3 *T* algorithm.

Input: They are the same as *S* algorithm
Output: $T()$ result TR

$R_{y0} = 0$
$R_{z0} = x \cdot 2$
if $rm = 2$ $R_{x0} = 0.54920653198$
else if $rm = 3$ $R_{x0} = 0.54885034814$
else if $rm = 4$ $R_{x0} = 0.54884897415$
else if $rm = 5$ $R_{x0} = 0.54884895243$
else $R_{x0} = 0$
$(R_{xn}, R_{yn}, R_{zn}) = RHC(R_{x0}, R_{y0}, R_{z0}, m, n)$
$V_{x0} = R_{xn} + R_{yn} + 1$, $V_{y0} = 1$, $V_{z0} = 0$
$(V_{xp}, V_{yp}, V_{zp}) = VLC(V_{x0}, V_{y0}, V_{z0}, p)$
Return $TR = (1 - V_{zp}) \cdot 2$

3. Proposed Architecture Based on CSM-VLC Method

In this section, we introduce another method based on the CSM (carry-save method) and VLC. The difference between this method and the RHC-VLC-based method is that the latter uses RHC to compute the exponential directly, whereas here the exponential is calculated based on the CSM. Therefore, we focus on how to calculate e^{-x} and e^{2x} in the following.

3.1. Compute E^{-X} and E^{2X}

Generally speaking, the expansion of exponential functions in accordance with Taylor's formula requires a large number of multiplication operations, while the hardware implementation of multiplication not only consumes a lot of logical resources, but it also leads to a long data path delay. In addition, the input range of the exponential functions should not be large, otherwise the data bit width is too large or the precision is not guaranteed in the hardware implementation. Therefore, in order to overcome the above disadvantages, we design a special hardware architecture for e^{-x} and e^{2x} in *tanh* and *sigmoid* functions. First, by means of the mathematical formula transformation, we convert e^{-x} and e^{2x} into:

$$e^{-x} = 2^{(-x)\cdot log_2 e} = 2^{r_1},\ e^{2x} = 2^{(2x)\cdot log_2 e} = 2^{r_2}. \tag{13}$$

When we get the values of r_1 and r_2, then we can easily separate their integer and decimal parts, assuming: $r_1 = I_1 + D_1,\ r_2 = I_2 + D_2$, where

$$I_1 = floor((-x) \cdot log_2 e),\ I_2 = floor((2x) \cdot log_2 e), \tag{14}$$

and

$$D_1 = (-x) \cdot log_2 e - floor((-x) \cdot log_2 e),\ D_2 = (2x) \cdot log_2 e - floor((2x) \cdot log_2 e), \tag{15}$$

$floor(x)$ is a function that returns the greatest integer less than or equal to x, which is usually denoted as "$floor(x) = [x]$"—for example, $floor(-1.2) = -2$ and $floor(1.2) = 1$. Further, we can simplify Equation (13) as follows:

$$\begin{aligned} e^{-x} = 2^{I_1+D_1} = 2^{I_1} \cdot 2^{D_1} = \begin{cases} 2^{D_1} << I_1, & for\ \ I_1 > 0, \\ 2^{D_1} >> (-I_1), & for\ \ I_1 \leq 0, \end{cases} \\ e^{2x} = 2^{I_2+D_2} = 2^{I_2} \cdot 2^{D_2} = \begin{cases} 2^{D_2} << I_2, & for\ \ I_2 > 0, \\ 2^{D_2} >> (-I_2), & for\ \ I_2 \leq 0, \end{cases} \end{aligned} \tag{16}$$

"$<< X$" means to move X bit to the left and "$>> Y$" means to move Y bit to the right.

Therefore, two natural exponential functions are transformed into a same exponential function with base 2, and the computation range is changed from "$(-\infty,\ +\infty)$" to "[0, 1)", which just needs an extra shift operation. For "2^{D_1} or 2^{D_2}, $D_1, D_2\ \epsilon\ [0,\ 1)$", we can easily implement them according to the required precision by taking advantage of the LUT-based or PWL-based methods.

After introducing the above transformation process, we can obtain the hardware architecture shown in Figure 10, which contains a constant multiplier, an exponential calculator (EC) and a shift unit (SU).

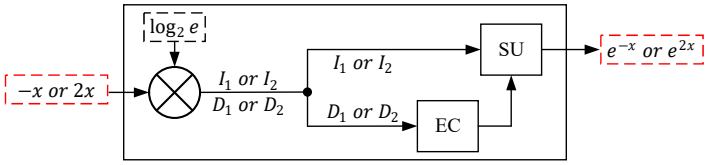

Figure 10. The architecture of computing exponential function.

As we mentioned earlier, a multiplier not only consumes a lot of resources in hardware implementation, but also causes long data path delay. So we introduce a method of optimizing the constant multiplier used in this paper, which can greatly reduce its hardware complexity and its critical path. The optimization method includes algorithmic level and architectural level.

First, we talk about the optimization at the algorithmic level. We know that $(-x) \cdot log_2 e$ and $(2x) \cdot log_2 e$ can be written as

$$(-x) \cdot log_2 e = x \cdot log_2 \frac{1}{e},\ (2x) \cdot log_2 e = x \cdot log_2 e^2, \tag{17}$$

where $log_2 \frac{1}{e}$ is approximately equal to -1.442695040888963 and $log_2 e^2$ approximates to 2.885390081777927. We can further get

$$\begin{aligned} x \cdot (-1.442695040888963) &= x \cdot (-2 + 0.557304959111037) \\ &= -2x + 0.557304959111037x, \\ x \cdot (2.885390081777927) &= x \cdot (2 + 0.885390081777927) \\ &= 2x + 0.885390081777927x. \end{aligned} \tag{18}$$

Then, we introduce the optimization at the architectural level. In binary terms, we know that 0.557304959111037 is approximately $(0.1000111)_2$ (=0.5546875) and 0.885390081777927 is approximately $(0.1110001)_2$ (=0.8828125). If we design a constant multiplier in the normal way, there will be four shifts and three additions. However, we can simplify it to three shifts or two shifts, one addition and one subtraction, which is called the "3-1-1" or the "2-1-1" method in this paper. The implementation process of this method is as follows.

The above two binary numbers can be further rewritten as:

$$\begin{aligned}(0.1000111)_2 &= (0.1000000)_2 + (0.0001000)_2 - (0.0000001)_2,\\ (0.1110001)_2 &= (1.0000000)_2 + (0.0000001)_2 - (0.0010000)_2.\end{aligned} \tag{19}$$

In this case, that is

$$\begin{aligned}x_1 \cdot (0.1000111)_2 &= x_1 >> 1 + x_1 >> 4 - x_1 >> 7,\\ x_2 \cdot (0.1110001)_2 &= x_2 + x_2 >> 7 - x_2 >> 3.\end{aligned} \tag{20}$$

The top one is "3-1-1", and the bottom one is "2-1-1" in Equation (20). In general, "one plus and one minus" can be converted to "three plus"; that is, subtracting a number is equal to adding its two's complement. Then, we turn Equation (20) into:

$$\begin{aligned}x_1 \cdot (0.1000111)_2 &= x_1 >> 1 + x_1 >> 4 + \sim (x_1 >> 7) + 1,\\ x_2 \cdot (0.1110001)_2 &= x_2 + x_2 >> 7 + \sim (x_2 >> 3) + 1.\end{aligned} \tag{21}$$

Supposing that x_1 and x_2 are all M bits, if we use a traditional method to compute the above formula, $3M$ full adders (FAs) are needed and the critical path time is $3M\tau$ (τ is the operation time of a full adder). However, we use a method called CSM to reduce both the number of FAs (from $3M$ to $2M$) and the critical path time (from $3M\tau$ to $(M+1)\tau$). Of course, a reduction in the number of FAs means that the area and power consumption of the whole architecture will be also reduced. In fact, we can also consider using the carry select adders [20–22] to implement Equation (21), but for the sake of balancing area and speed, we do not adopt it in this paper.

Let the equations for "$x_1 \cdot (0.1000111)_2$" and "$x_2 \cdot (0.1110001)_2$" look like "$I^1 + I^2 + I^3 + 1$" and let their results be expressed in terms of "R", the optimized architecture for computing $0.5546875x_1$ or $0.8828125x_2$, as is shown in Figure 11.

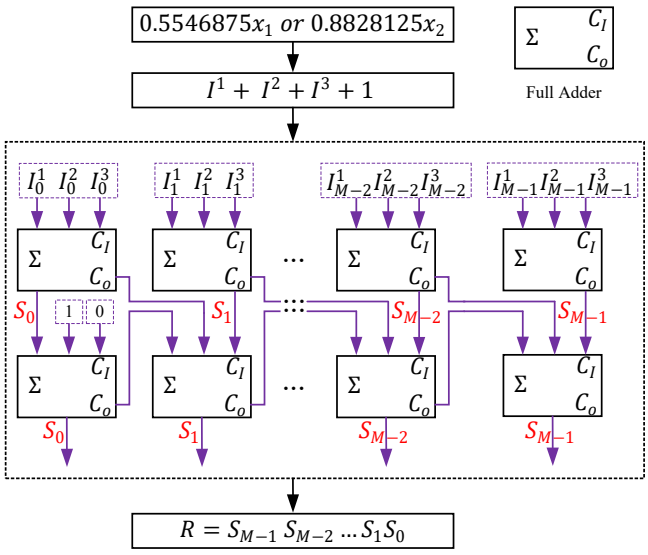

Figure 11. The optimized constant multiplier proposed by carry-save method.

3.2. Software Test for the CSM-VLC-Based Method

Now, we use the accuracy criteria (AAE and MAE) to evaluate the precision of this method. Compared with the method based on RHC-VLC, the CSM-VLC-based method has a wider input range. Although there is no scope limitation in nature, we still set two limited test input ranges: $[-2, 2]$ and $[-12, 12]$.

Since the iterations (p) of VLC can be set flexibly, the precision of division operation will also change accordingly. Therefore, we set the parameter p from 1 to 20 to explore the accuracy of computing $S\&T$ with the CSM-VLC-based method. The final simulation results can be obtained through MATLAB, as shown in Figure 12. (It uses logarithmic ordinate.)

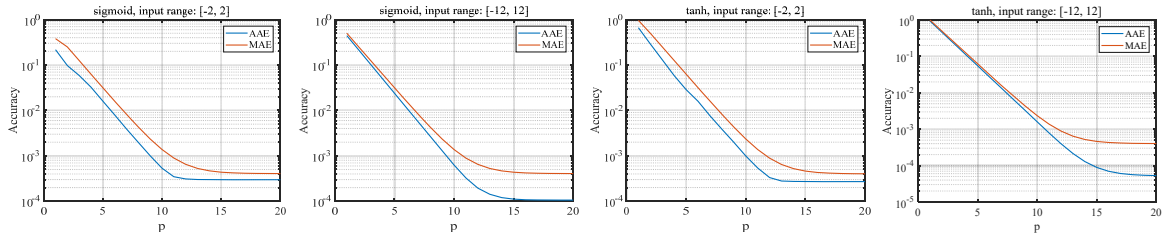

Figure 12. Accuracy simulation of computing $S\&T$ based on CSM-VLC method.

From Figure 12, we can draw the following conclusions: (1) with the increase in p, the accuracy of computing $S\&T$ with this method becomes higher; (2) the greater the p value, the more the accuracy tends to remain unchanged; (3) the larger the input range and the larger the p value, the higher the accuracy; (4) compared with the method based on RHC-VLC, the overall computation accuracy of this method is not dominant; (5) the input range of this method is essentially unlimited, but the RHC-VLC-based method needs to adjust the iterations of RHC to expand the input range, so the architecture based on CSM-VLC is superior in this respect.

3.3. Proposed Algorithm with Adjustable Precision

Through the previous simulation verification, we obtain the relationship between the computation accuracy of $S\&T$ and the iterations of VLC. Since different precision magnitudes correspond to different iterations of VLC, we first classify and screen them according to a principle of fewer iterations. (Principle: AAE and MAE are expressed as "$K \times 10^E$", $K \epsilon (1, 5]$ and E is a negative integer), as shown in Table 6. From Figure 12, we know that even if the p value becomes larger, it is difficult to improve the accuracy by another magnitude. Therefore, only MIV corresponding to three kinds of accuracy magnitude are shown in the Table 6, which also lays the foundation for another proposed algorithm based on CSM-VLC with adjustable precision.

According to the final optimization of e^{-x} and e^{2x} and Table 6, we can obtain another new algorithm (based on CSM-VLC) to compute $S\&T$, which is called the $CVST$ algorithm (Algorithm 4) in this paper. The specific implementation of $VLC()$ in this algorithm refers to the previous algorithm. In the $CVST$ algorithm, we use MAE as the accuracy standard an,d of course, we can also adopt AAE.

Table 6. Sigmoid: Relationship between the Magnitudes of Accuracy and the Iterations.

Accuracy Magnitude	Accuracy Type	Function Type	K *Value*	MIV [1] *p*
E = −2	AAE	Sigmoid	3.24	4
		Tanh	2.84	5
	MAE	Sigmoid	3.16	5
		Tanh	3.16	6
E = −3	AAE	Sigmoid	3.96	7
		Tanh	3.89	8
	MAE	Sigmoid	4.31	8
		Tanh	3.29	9
E = −4	AAE	Sigmoid	3.52	11
		Tanh	3.26	12
	MAE	Sigmoid	4.66	14
		Tanh	4.61	15

(1) MIV: Minimum Iterations of VLC. (2) Input Range: $[-2, 2]$.

Algorithm 4 $CVST$ algorithm.

Input: They are the same as $RVST$ algorithm
Output: Final result FR

if $t = 0$ **then**
  $V = -2x + 0.5546875x$
  if $rm = 2$ $p = 5$
  else if $rm = 3$ $p = 8$
  else if $rm = 4$ $p = 14$
  else $p = 0$
else
  $V = 2x + 0.8828125x$
  if $rm = 2$ $p = 6$
  else if $rm = 3$ $p = 9$
  else if $rm = 4$ $p = 15$
  else $p = 0$
end if
$Int = floor(V)$
$Fra = V - floor(V)$
$V_{x0} = 2^{Int} \cdot 2^{Fra}$, $V_{y0} = 1$, $V_{z0} = 0$
$(V_{xn}, V_{yn}, V_{zn}) = VLC(V_{x0}, V_{y0}, V_{z0}, p)$
if $t = 0$ **then**
  Return $FR = V_{zn}$
else
  Return $FR = 1 - V_{zn} \cdot 2$
end if

4. Hardware Implementation

After proposing two algorithms with adjustable accuracy to calculate $S\&T$ functions, and learning their advantages and disadvantages through software simulation, we further compare their pros and cons through specific hardware implementations, including the comparison with other methods.

For the convenience of expression, we denote the hardware architecture of $RVST$ algorithm as $AR\ |_V^R\ (rm, m_i)$. Similarly, the $CVST$ algorithm in hardware implementation is named $AR\ |_V^C\ (rm)$. Here, rm refers to the magnitudes of MAE and m_i determines the range of input x.

4.1. General Architecture Based on RHC-VLC Method

According to the $RVST$ algorithm, we can design two structures: one is a non-pipelined structure, which can save a lot of hardware area. The other is a pipelined structure, which can obtain high throughput. In this paper, all hardware designs adopt the pipelined structure.

The general architecture $AR\ |_V^R\ (rm, m_i)$ is shown in Figure 13, which mainly includes PS (precision selector) module, RHC module, VLC module and Adder. When the inputs (x, t, rm, m_i) are valid, the sign bit of x will flip if $t = 0$, otherwise, x will be shifted left for one bit as with the initial input z_0 of the RHC module. The PS module selects the corresponding iterations (m, n, p) according to the values of rm, m_i and t, then it transmits them to the RHC module and the VLC module, respectively. After a fixed number of iterations, the output of RHC module is valid, x_n and y_n of RHC will be added up by adder and used as the initial input x_0 of the VLC module. Similarly, when the output of the VLC module is valid, if $t = 1$, z_p will convert its sign bit and then be shifted one bit to the left, the result is added up by 1 as the final result of $T(x)$. However, if $t = 0$, z_p will be directly the final result of $S(x)$.

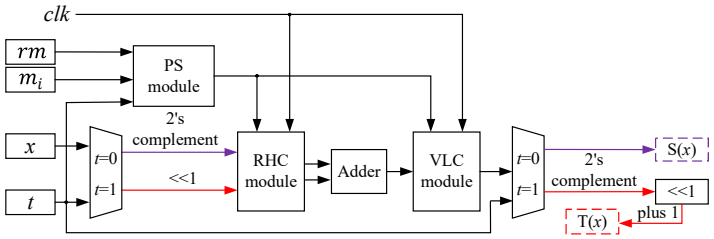

Figure 13. The general architecture ($AR\ |_V^R\ (rm, m_i)$) of computing $S\&T$.

Next, we detail the architecture of each module, namely the PS module, RHC module and VLC module. First, we introduce the PS module, as shown in Figure 14. Since m_i determines the input range of x, which is independent of the current function to be calculated and does not affect the calculation accuracy, it is directly equal to m. The values of rm and t determine the different combinations of n and p. m and n are transmitted to the RHC module and p to the VLC module. See Tables 3 and 4 for "$case\ of\ sigmoid$" and "$case\ of\ tanh$".

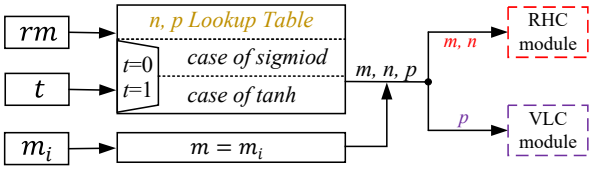

Figure 14. The architecture of PS module in $AR\ |_V^R\ (rm, m_i)$.

Figure 15 shows that the inputs and outputs of each iteration of VLC are cascaded back and forth. The kth stage includes the shift operations, which needs to shift x_k by k bits to the right. The constants contained in the look-up table are calculated by 2^{-k}. In the same way, Figure 16 describes a pipelined architecture of RHC module. Unlike the VLC module, there are two kinds of architecture in the RHC module. One is the RHC of positive iterative number (called P-RHC), another is that of the non-positive iterative number (called NP-RHC). For the P-RHC, the kth stage needs to shift x_k and y_k by k bits to the right and the constants in Lookup Table 2 are computed by $tanh^{-1}(2^{-k})$. However, the kth stage of NP-RHC should shift x_k and y_k by 2^{1-k} bits to the right and the elements in Lookup Table 3 are calculated by $tanh^{-1}(1 - 2^{-2^{-k+1}})$. There are seven cases of the initial value x_0, the specific constant values are shown in Table 5 and stored in Lookup Table 1.

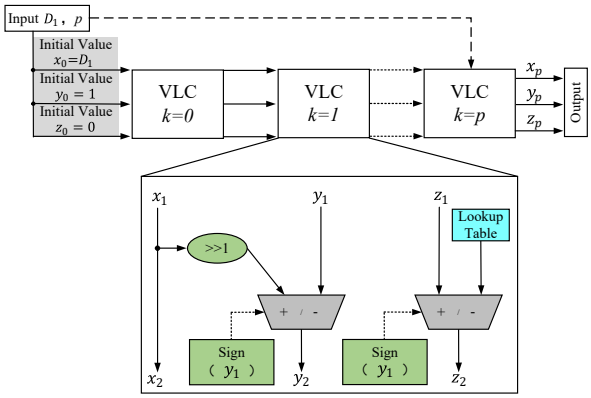

Figure 15. The pipelined architecture of VLC module.

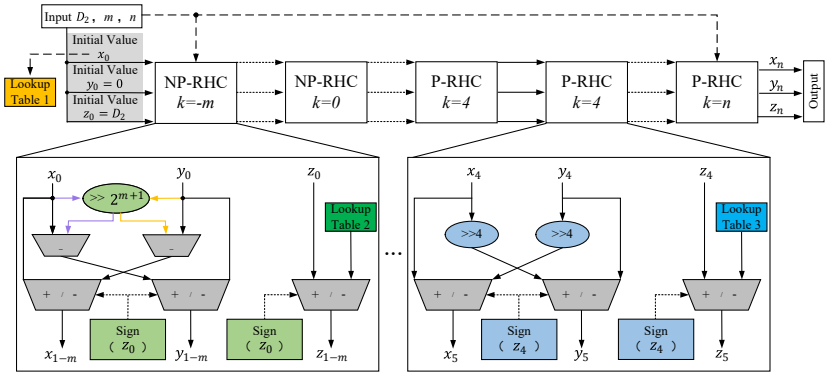

Figure 16. The pipelined architecture of RHC module.

4.2. General Architecture Based on CSM-VLC Method

Different from $AR\ |_V^R\ (rm, m_i)$, the input m_i is no longer needed in $AR\ |_V^C\ (rm)$. In the PS module, there can be no case of $rm \geq 5$ because the upper limit of rm is 4 (See Table 6 for details). There only exists one parameter p in the PS module and it is passed to the VLC module. The RHC module will be replaced by a CSM-based exponential computing module (called the CEC module), which is shown in Figures 10 and 11 and not further described here. The Adder is no longer needed because the output of the CEC module is equivalent to the output of it.

4.3. Implementation of a Specific Case

We design a specific case $AR\ |_V^R\ (3,0)$ and $AR\ |_V^C\ (3)$, respectively, using Verilog HDL, and synthesize them under the TSMC 40 nm CMOS technology. The report shows that the area of $AR\ |_V^R\ (3,0)$ consumes 4290.98 μm^2 and the power of that costs 1.69 mW at the frequency of 1.5 GHz. Under the same frequency, $AR\ |_V^C\ (3)$ costs the area of 3196.36 μm^2 and the power of 1.38 mW. Compared with $AR\ |_V^R\ (3,0)$, $AR\ |_V^C\ (3)$ can save 25.51% area and 18.34% power. In fact, both architectures can operate at a frequency of more than 2 GHz. In the following section, we present the details of the hardware design, including word length, latency analysis and error analysis.

First, let us look at the word lengths required for each module (Input Range: $S(x) \rightarrow [-2,2]$, $T(x) \rightarrow [-1,1]$). According to Section 2.3, we know that the AAE of $AR\ |_V^R\ (3,0)$ are 4.77×10^{-3} for the *sigmoid* function, and 3.84×10^{-3} for the *tanh* function, respectively. The AAE of $AR\ |_V^C\ (3)$ are 4.31×10^{-3} for the *sigmoid* function, and 3.29×10^{-3} for the *tanh* function, respectively. To achieve the accuracy, we set the fractional part of the input or output data of top-level module to be nine bits. The minimum number that nine fractional bits can represent is $1/(2^9) = 1.95 \times 10^{-3}$, which is lower than MAE, mentioned above ($1/(2^8) = 3.91 \times 10^{-3} > 3.84 \times 10^{-3}$).

From Figures 2 and 3, we know that the MAE of the RHC module ($m = 0, n = 8$) is 2.82×10^{-2} for the *sigmoid* or *tanh* functions. For $AR\ |_V^R\ (3,0)$, the MAE of the VLC module is 3.91×10^{-3} ($p = 8$)

for the *sigmoid* function, and 9.76×10^{-4} ($p = 10$) for the *tanh* function, respectively. For $AR\ |_V^C\ (3)$, the MAE of VLC module is the same as $AR\ |_V^R\ (3,0)$ for the *sigmoid* function, and 1.95×10^{-3} ($p = 9$) for the *tanh* function. Therefore, we set the fractional part of input or output data of RHC module to be six bits, whose minimum number is $1/(2^6) = 1.56 \times 10^{-2}$. The fractional part of input or output data of the VLC module are set to be 11 bits ($1/(2^{11}) = 4.88 \times 10^{-4}$). Since the MAE of computing e^{-x} using the CEC module (Input Range: $[-2, 2]$) is 2.68×10^{-2}, and that of computing e^{2x} using CEC module (Input Range: $[-1, 1]$) is 1.32×10^{-2}, respectively, we set the fractional part of input or output data of CEC module to be seven bits ($1/(2^7) = 7.81 \times 10^{-3}$). The fractional part of input or output data of Adder is the same as the RHC module. The required integer bits of input or output data of each module can be calculated from the input range of the top-level module. The word length setting of each module is outlined in Table 7.

Table 7. Word Length Setting.

Module	Data	Sign bit	Integral bit	Fractional bit	Total bit
top-level	Input	1	2	9	12
	Output	1	0	9	10
RHC	Input	1	2	6	9
	Output	1	3	6	10
CEC	Input	1	2	7	10
	Output	0	3	7	10
Adder	Input	1	3	9	13
	Output	0	3	9	12
VLC	Input	0	3	11	14
	Output	0	1	11	12

Next, we analyze the latency of $AR\ |_V^R\ (3,0)$ and $AR\ |_V^C\ (3)$. In these two cases, $m = 0$, n and p vary with rm and t, the RHC module has $(n+1)$ cascaded stages. Because n is smaller than 13 in this special case, the RHC module only just needs 1 repeated stages ($n = 4$). Each stage needs one clock cycle. Thus, the RHC module requires $(n+2)$ clock cycles. The VLC module has $(p+1)$ cascaded stages and it needs $(p+1)$ clock cycles. The Adder also needs additional one clock cycle. On the whole, the latency of computing $S\&T$ based on RHC-VLC method is $n+p+4$ clock cycles. For the method based on CSM-VLC, since the CEC module can compute the value of e^{-x} or e^{2x} in one clock cycle, the latency of this method is $(p+2)$ clock cycles in total. From Tables 4 and 6, it can be concluded that 20 clock cycles are needed to calculate the *sigmoid* function by the RHC-VLC method, and 22 clock cycles are needed to calculate the *tanh* function. If the method based on CSM-VLC is used, 10 clock cycles are needed to calculate *sigmoid* function and 11 clock cycles are needed to calculate *tanh* function.

As for arbitrary $AR\ |_V^R\ (rm, m_i)$ and $AR\ |_V^C\ (rm)$, different order of magnitude of accuracy correspond to different number of iterations (n and p) of CORDIC, as shown in Tables 4 and 6, and the total latency is also different. It is important to note that when $n = 13$, the RHC module requires two more clock cycles according to the convergence.

Finally, we make an error analysis of the two cases. We generate three groups of 10,000 random numbers by MATLAB and take them as the test benchmark of the example circuit. The accuracy of the specific cases are evaluated by comparing the outputs from Vivado with the results of MATLAB's original functions. After the comparison, we find that the order of magnitude of AAE and MAE keeps the same for both hardware implementation and software validation. Additionally, we use another metric "bit position error" [23] to compute the probability error (PE) of each bit. Taking the AAE of

computing *sigmoid* function as an example, Figure 17 shows that the PE of all bits is lower than 0.5. Of all the bits, the PE of the last two is close to 0.5, which indicates that these two bits are not very accurate. Overall, however, the accuracy of the example circuits is as expected. In addition, we can also find that $AR\,|_V^R\,(3,0)$ and $AR\,|_V^C\,(3)$ have roughly the same probability of error, and the phenomenon is basically consistent with the software simulation results.

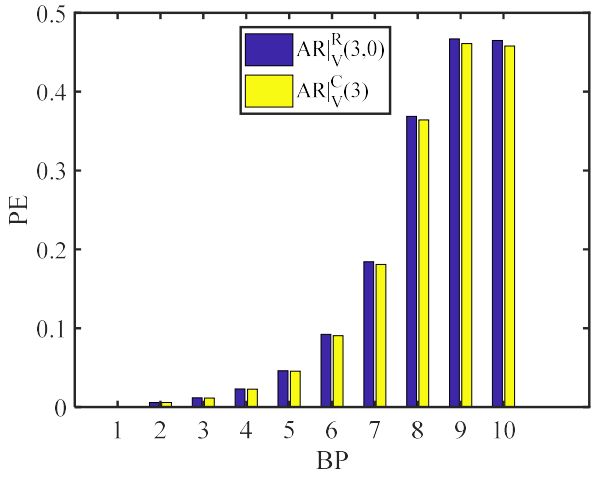

Figure 17. Bit position error of computing *sigmoid* function.

4.4. Comparison with Existing Methods

In this section, in order to illustrate that our proposed methods have the advantages of efficient hardware architecture with adjustable precision and high speed, we analyze the cost of implementing $S\&T$ with adjustable precision in other methods from the perspective of hardware implementation.

4.4.1. Comparison with LUT-Based Method

The LUT-based method typically stores all possible results in memory. The more data that needs to be stored in memory, the larger the hardware area. Therefore, it will consume a lot of logic and storage resources when using this method to implement a hardware architecture with adjustable precision and input range. If the input range is $[M, N]$ and the accuracy is required to be $RA = 2^{-a}$, and the amount of data to be stored will be

$$DaTo = (N - M) \times RA^{-1} + 1. \tag{22}$$

Each piece of data is composed of an m integer bit and n fractional bits. In that case, all of the data need $BitTo = DaTo(m + n)$ bits. That is

$$BitTo = [(N - M) \times 2^a + 1](m + a), \tag{23}$$

where n is equal to a. It can be seen from the formula that the hardware area required to store these data is positively correlated with the accuracy or input range. For example, if $a = 5$, $M = -1$, $N = 1$, RA will be 2^{-5} (3.125×10^{-2}), $DaTo$ will be 65. m is at least 1 and n is 5. $BitTo$ will be 390. However, as the accuracy increases from 3.125×10^{-2} to 4.883×10^{-4}, a ($n = a$) is at least 11. $DaTo$ will be 4097 and $BitTo$ will be 49,164. Thus, it can be seen that when order of magnitude of accuracy is changed from -2 to -4, the required hardware area will be increased by approximately by 126 (49,164/390) times.

Although the LUT-based method has a low latency at low precision, a large amount of data needs to be stored at high precision. For example, to reach the precision level of 10^{-4}, the number of node data and result data is 2048, respectively ($1/2048 = 4.88 \times 10^{-4}$). In order to find the result data, the latency of serial search mode will be huge. Although the parallel search mode can be adopted, the number of parallel paths should not be large considering the area overhead. Assuming that

a 32-way parallel search is adopted, and 64 nodes data of each way are compared in a serial way. Each comparison consumes one clock cycle, then at least 64 cycle clocks are required, which is higher than our methods. Obviously, the low latency advantage of this method will not exist when we need high precision.

Since memory in hardware costs a lot of area, it would be costly and undesirable to use this method to compute $S\&T$ functions with adjustable precision and variable input ranges. This method is only applicable to the case of narrow input range and low precision requirement, so as to give full play to the maximum benefit of short delay. The flexible precision and input ranges mean that the data are diverse and it is hard to be compatible with the method's data fixation.

4.4.2. Comparison with PWL-Based Method

The PWL-based method approximates the target functions into many linear segments. In each input interval, the results are computed according to the corresponding linear segment. Suppose the target function is $f(x)$ and the input range is $[M, N]$. After PWL segmentation, the target function is divided into a number of small linear line segments $S_i(k_i)$, each k_i belongs to $[M_i, N_i]$ and $[M_i, N_i]$ belongs to $[M, N]$. In this method, the linear segmentation of $f(x)$ is usually carried out by software simulation in advance according to the accuracy requirement. Then, in hardware implementation, the slope and intercept of the linear segment are stored in memory. In other words, if the method is used to design a hardware architecture with adjustable precision or flexible input range, the segmentation process at the software level must be implemented in hardware. Otherwise, when the accuracy or input range needs to be changed, we must first use the software for linear segmentation, which is not only inefficient, but also tedious.

Further, we analyze the cost of designing a hardware architecture with adjustable precision or flexible input range based on the PWL method. From [24], we learn that, in order to obtain the slope and intercept of all linear segments, the PWL-based method needs division and multiplication operations for many times. Moreover, if we want to achieve the optimal segmentation results, the number of iterations is also difficult to determine, because it is related to the required precision and input range. In the hardware implementation, the multiplier is also required to compute the final result.

Finally, we make an analysis of the latency. In order to build an adjustable-precision hardware architecture, we need to use dividers and multipliers to obtain all slopes and intercepts. Assuming that the latency of multiplier and divider is one clock cycle each (in order to improve the working frequency, it is actually far more than one clock cycle), the input range is divided into N intervals. Each interval requires at least one multiplication and division operation, that is, to achieve the slope and intercept of the fitting line segment of each interval, it requires at least $2N$ clock cycles. From [24], in order to obtain the maximum absolute error of 1×10^{-3}, 19 segments ($N = 19$) are needed to calculate *sigmoid* function and 27 segments ($N = 27$) are needed to calculate *tanh* function. Therefore, the total latency of the PWL-based method is much higher than that of our proposed methods.

In basic mathematical operations, the hardware implementation of division and multiplication is the most complicated, which not only consumes a lot of logic resources, but also reduces the working frequency of the whole architecture. Compared with our proposed architecture to implement $S\&T$ functions with adjustable precision and extensible input range in this paper, the PWL-based method is worse and incomparable.

4.4.3. Comparison with Other Related Methods

In addition to the above mainstream methods, there are also some other methods to compute *tanh* and *sigmoid*, such as stochastic computing [25] or a mixture of stochastic logic and PWL [11].

In [25], we learn that the $S\&T$ functions can be implemented in stochastic computing according to the Horner's rule for Maclaurin expansion. It is shown that, if the coefficients are alternately positive and negative and their magnitudes are monotonically decreasing, a polynomial can be implemented using multiple levels of NAND gates based on Horner's rule. Truncated Maclaurin series expansions

of arithmetic functions are used to generate polynomials which satisfy these constraints. In contrast to the PWL-based method, it does not require the multipliers, but rather a stochastic number generator to convert digital values to stochastic bit streams. The most obvious disadvantages are that the precision is not high and the input range is limited, so it is difficult to build an architecture with adjustable precision and flexible input range to compute $S\&T$.

The method [11] can improve the accuracy of the calculation results through linear feedback registers, but it needs to produce sufficient and uncorrelated random numbers. Compared with [25], it can also reduce the area. However, it is still hard to build a hardware architecture with adjustable precision and extensible input range compared with our methods.

In order to reveal the strengths and weaknesses of the above methods and our methods, we implement the hardware architectures based on these methods. Synthesized under the TSMC 40 nm COMS technology, their performance indicators are shown in Table 8.

Table 8. Hardware Performance Comparison Between Our Methods and Existing Methods.

Purpose: To Build a Hardware Architecture with Adjustable Precision and Extensible Input Range.					
Method	**LUT [6]**	**PWL [10]**	**MSC-PWL [1] [11]**	**RHC-VLC (Proposed)**	**CSM-VLC (Proposed)**
Area	62,376.21 μm^2	11,521.68 μm^2	8638.49 μm^2	4364.57 μm^2	4048.64 μm^2
Power	7.75 mW	5.14 mW	3.29 mW	1.89 mW	1.75 mW
Frequency	1 GHz	700 MHz	800 MHz	1.5 GHz	1.5 GHz
Input Range	$[-2, 2]$	$[-1, 1]$	$[-1, 1]$	$S(x) \rightarrow [-2, 2]$ $T(x) \rightarrow [-1, 1]$	$[-2, 2]$
Range of MAE	$10^{-1} \sim 10^{-4}$	$10^{-1} \sim 10^{-3}$	$10^{-1} \sim 10^{-3}$	$10^{-1} \sim 10^{-4}$	$10^{-1} \sim 10^{-4}$
Range of AAE	$10^{-1} \sim 10^{-4}$	$10^{-1} \sim 10^{-3}$	$10^{-1} \sim 10^{-3}$	$10^{-1} \sim 10^{-4}$	$10^{-1} \sim 10^{-4}$
Has multipliers or dividers?	No	Multipliers and Dividers	Multipliers and Dividers	An optimized constant multiplier	No

[1] MSC-PWL: Mixture of Stochastic Computing and PWL.

In conclusion, our methods have the following characteristics, which are difficult for other methods to be compatible with for all advantages.

- Efficient hardware: Our RHC-VLC-based method only requires shift-and-add (or subtract) operations, which can avoid the direct use of inefficient multiplication and division. For the constant multiplier used in the CSM-VLC-based method, we also made specific optimization and design to improve the hardware efficiency.
- Adjustable precision: Both of our methods can easily adjust the accuracy of calculation results by increasing or decreasing the number of CORDIC iterations.
- Extensible range: According to the application requirements, the method based on RHC-VLC can adjust the negative iterations of RHC freely to expand or narrow the input range. In theory, the method based on CSM-VLC even has no limitation of input range.
- High speed: The hardware architecture based on our proposed methods can work at the frequency of 1.5 GHz, or even higher, such as 2 GHz.

5. Conclusions

In this paper, an efficient hardware architecture with adjustable precision and extensible input range is proposed to compute *tanh* and *sigmoid* functions for the first time. At first, we introduce two methods in detail—RHC-VLC based and CSM-VLC based. Then, we use MATLAB to conduct accuracy simulation to further find out the relationship between accuracy distribution and CORDIC iterations. Next, at the RTL level, we implement a special case of these two methods and compare them under the TSMC 40 nm COMS technology. Finally, we analyze the costs and drawbacks of other

approaches to building an architecture with adjustable precision, and demonstrate that our proposed methods combine the following three advantages: efficient hardware, adjustable precision and high working frequency.

Author Contributions: H.C. conceived and designed the methodology and architecture to implement the sigmoid and tanh functions; H.C. performed the experiments with support from L.J. and H.Y.; H.C. analyzed the experimental results; H.C., L.J., and H.Y. contributed task decomposition and corresponding implementations; H.C. wrote the paper; Z.L. and Z.Y. helped to revise the manuscript; L.L. and Y.F. supervised the project. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the National Nature Science Foundation of China under Grant No. 61176024, Nanjing University Technology Innovation Fund.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Namin, A.H.; Leboeuf, K.; Muscedere, R.; Wu, H.; Ahmadi, M. Efficient Hardware Implementation of the Hyperbolic Tangent Sigmoid Function. In Proceedings of the IEEE International Symposium on Circuits and Systems, Taipei, Taiwan, 24–27 May 2009; pp. 2117–2120.
2. Sartin, M.A.; da Silva, A.C.R. Approximation of Hyperbolic Tangent Activation Function Using Hybrid Methods. In Proceedings of the International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), Darmstadt, Germany, 10–12 July 2013.
3. Gomar, S.; Mirhassani, M.; Ahmadi, M. Precise Digital Implementations of Hyperbolic Tanh and Sigmoid Function. In Proceedings of the Asilomar Conference on Signals, Systems and Computers, Pacific Grove, CA, USA, 6–9 November 2016; pp. 1586–1589.
4. Piazza, F.; Uncini, A.; Zenobi, M. Neural Networks with Digital LUT Activation Functions. In Proceedings of the International Conference on Neural Networks, Nagoya, Japan, 25–29 October 1993; Volume 2, pp. 1401–1404.
5. Leboeuf, K.; Namin, A.H.; Muscedere, R.; Wu, H.; Ahmadi, M. High Speed VLSI Implementation of the Hyperbolic Tangent Sigmoid Function. In Proceedings of the International Conference on Convergence and Hybrid Information Technology, Busan, Korea, 11–13 November 2008; Volume 1, pp. 1070–1073.
6. Meher, P.K. An Optimized Lookup-Table for the Evaluation of Sigmoid Function for Artificial Neural Networks. In Proceedings of the IEEE/IFIP International Conference on VLSI and System-on-Chip, Madrid, Spain, 27–29 September 2010; pp. 91–95.
7. Low, J.Y.L.; Jong, C.C. A Memory-Efficient Tables-and-Additions Method for Accurate Computation of Elementary Functions. *IEEE Trans. Comput.* **2013**, *62*, 858–872. [CrossRef]
8. Myers, D.J.; Hutchinson, R.A. Efficient Implementation of Piecewise Linear Activation Function for Digital VLSI Neural Networks. *Electron. Lett.* **1989**, *25*, 1662–1663. [CrossRef]
9. Basterretxea, K.; Tarela, J.M.; del Campo, I. Approximation of Sigmoid Function and the Derivative for Hardware Implementation of Artificial Neurons. *IEE Proc. Circuits Devices Syst.* **2004**, *151*, 18–24. [CrossRef]
10. Armato, A.; Fanucci, L.; Scilingo, E.; Rossi, D.D. Low-Error Digital Hardware Implementation of Artificial Neuron Activation Functions and Their Derivative. *Microprocess. Microsyst.* **2011**, *35*, 557–567. [CrossRef]
11. Nguyen, V.; Luong, T.; Le Duc, H.; Hoang, V. An Efficient Hardware Implementation of Activation Functions Using Stochastic Computing for Deep Neural Networks. In Proceedings of the IEEE International Symposium on Embedded Multicore/Many-Core Systems-on-Chip (MCSoC), Hanoi, Vietnam, 12–14 September 2018; pp. 233–236.
12. Zhang, M.; Vassiliadis, S.; Delgado-Frias, J.G. Sigmoid Generators for Neural Computing Using Piecewise Approximations. *IEEE Trans. Comput.* **1996**, *45*, 1045–1049. [CrossRef]
13. Xie, Z. A Non-Linear Approximation of the Sigmoid Function Based on FPGA. In Proceedings of the IEEE Fifth International Conference on Advanced Computational Intelligence (ICACI), Nanjing, China, 18–20 October 2012; pp. 221–223.
14. Zamanlooy, B.; Mirhassani, M. Efficient VLSI Implementation of Neural Networks with Hyperbolic Tangent Activation Function. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2014**, *22*, 39–48. [CrossRef]
15. Feng, F.; Li, L.; Wang, K.; Fu, Y.X.; Pan, H.B. Design and Application Space Exploration of a Domain-Specific Accelerator System. *Electronics* **2018**, *7*, 45. [CrossRef]

16. Volder, J.E. The CORDIC Trigonometric Computing Technique. *IRE Trans. Electron. Comput.* **1959**, *EC-8*, 330–334. [CrossRef]
17. Walther, J.S. The Story of Unified CORDIC. *J. VlSI Signal Process. Syst. Signal Image Video Technol.* **2000**, *25*, 107–112. [CrossRef]
18. Chen, H.; Jiang, L.; Luo, Y.Y; Lu, Z.H.; Fu, Y.X.; Li, L.; Yu, Z.Z. A CORDIC-Based Architecture with Adjustable Precision and Flexible Scalability to Implement Sigmoid and Tanh Functions. In Proceedings of the IEEE International Symposium on Circuits & Systems, Sevilla, Spain, 10–21 October 2020.
19. Hu, X.; Harber, R.G.; Bass, S.C. Expanding the Range of Convergence of the CORDIC Algorithm. *IEEE Trans. Comput.* **1991**, *40*, 13–21. [CrossRef]
20. Bedrij, O.J. Carry-Select Adder. *IRE Trans. Electron. Comput.* **1962**, *EC-11*, 340–346. [CrossRef]
21. Ramkumar, B.; Kittur, H.M. Low-Power and Area-Efficient Carry Select Adder. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2012**, *20*, 371–375. [CrossRef]
22. Kesava, R.B.S.; Rao, B.L.; Sindhuri, K.B.; Kumar, N.U. Low Power and Area Efficient Wallace Tree Multiplier Using Carry Select Adder with Binary to Excess-1 Converter. In Proceedings of the Conference on Advances in Signal Processing (CASP), Pune, India, 9–11 June 2016; pp. 248–253.
23. Mopuri, S.; Acharyya, A. Low-Complexity Methodology for Complex Square-Root Computation. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2017**, *25*, 3255–3259. [CrossRef]
24. Sun, H.Q.; Luo, Y.Y.; Ha, Y.J.; Shi, Y.H.; Gao, Y.; Shen, Q.H.; Pan, H.B. A Universal Method of Linear Approximation With Controllable Error for the Efficient Implementation of Transcendental Functions. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2020**, *67*, 177–188. [CrossRef]
25. Parhi, K.K.; Liu, Y. Computing Arithmetic Functions Using Stochastic Logic by Series Expansion. *IEEE Trans. Emerg. Top. Comput.* **2019**, *7*, 44–59. [CrossRef]