An Exponential Function Accelerator with Radix-16 Algorithm for spiking neural networks

Chenxiao Lin12, Qingyang Zeng12, and Delong Shang1(a)

Abstract A range reduction method for shift-and-add algorithms for exponential functions is proposed in this paper. An exponential function accelerator with this method and radix-16 shift-and-add algorithm has been implemented in SMIC 55 nm CMOS process. Compared with the existing method, the proposed method reduces the latency (cycles) by 33% and 20% for 16 and 32-bit precision results, respectively; thereby increasing the throughput to 50M exp/s and reducing the power consumption to 4.6 pJ/exp. In addition, this method saves die area since no arithmetic units are adopted. This exponential accelerator is supposed to be used in a neuromorphic chip for spiking neural network modeling.

Key words: Ercegovac’s algorithm, exponential accelerator, neuron model, range reduction method, spiking neural network

Classification: Integrated circuits (memory, logic, analog, RF, sensor)

1. Introduction

Spiking neural networks (SNNs) have been considered as the next-generation artificial neural networks for machine learning [1]. Compared to convolutional neural networks (CNNs), SNNs incorporate time-domain characteristics, which makes SNNs more similar to creature’s neural systems. Various biological neuron models [2, 3, 4, 5] have been proposed by neuroscientists over the decades. A common characteristic among these neural models is involving exponential functions. In fact, the implementations with analog circuits or dedicated devices can naturally simulate some neural models with $e^x$ terms [6, 7, 8]. For example, the BrainScaleS-1 [8] system is based on physical (analog or mixed-signal) emulations of neuron, synapse and plasticity models with digital connectivity. In terms of more prevailing digital (binary) implementations, multiple designs for neurobiology researches [9] or industrial applications [10, 11] are published. The SpiNNaker-1 system [9] is based on numerical models running in realtime on custom digital multicore chips using the ARM architecture. Intel’s Loihi [10] and IBM’s TureNorth [11] are designed and fabricated for neuromorphic computing or researches. Among the projects based on digital very large-scale integration (VLSI) mentioned above, exponential functions in neuron models remain to be an unsolved problem. For example, TureNorth[11] uses a simplified neuron model that avoids exponential functions; SpiNNaker-1[9] uses look-up tables for exponential calculation, requiring too many RAM resources. A piece-wise linear (PWL) that can approximate the nonlinear behaviours in neural models is commonly used to alleviate computational cost [12, 13, 14], where the accuracy depends on the number of linear segments used.

Numerous implementations for exponential function both in silicon and on FPGA has been published [15, 16, 17, 18, 19, 20, 21, 22, 23, 24]. Commonly used exponential function algorithms for hardware implementation can be categorized into two types: polynomial approximation and shift-and-add algorithm [25]. Gomar et al.[17] converted the exponential term to a term of power of 2, i.e. $e^x = 2^{1.44x}$, which is simply implementable by “logic shift”. Partzsch et al.[18] chose a hybrid algorithm, combining two LUTs with a polynomial approximation. The input is separated into 3 parts: the $\exp(n)$ the integer part $n$ and the $\exp(p)$ of upper fractional part $p$ are evaluated via two LUTs, while the $\exp(q)$ of the lower fractional part $q$ is approximated by a polynomial. Nilsson et al.[22] implemented exponential function using 6 stages Taylor series. Mikaitis et al.[24] used radix-2 shift-and-add algorithm to approximate elementary function for their neuromorphic chip. In addition, Kim et al.[26] used the method called Template-scaling-based exponential function approximation (TS-EFA). It utilizes the property $e^x = e^{a_x} \cdot e^{x'}$, where $a_x$ is the scaling and $e^{x'}$ is the template as $x'$ is in a small range. In [27], the floating-point input is converted to fixed-point format $Z$ beforehand, then divided into 4 segments as the indices for 4 LUTs to obtain $e^Z$.

Ercegovac’s shift-and-add algorithm[28], also known as the radix-16 algorithm, is designed specifically for digital hardware implementation of exponential functions. It has the following characteristics:

1) Need no multiplication operation;
2) Hardware-efficiency;
3) Low latency.

Given these, this algorithm is adopted in our design.
The remainder of the paper is organized as follows. Section 2 briefly introduces Ercegovac’s shift-and-add algorithm and the existing range reduction method; Section 3 discusses the preliminary, introduces the proposed method and compares it with the existing method; Section 4 shows the structure of the whole design; Section 5 analyzes the error in detail; Section 6 presents our implementation results and compares them with peers’ work; The conclusion is presented in Section 7.

2. Algorithm and Existing Range Reduction Method

2.1 Algorithm
This section gives a brief description of the radix-16 shift-and-add algorithm and a comparison between the existing and the proposed range reduction methods.

The algorithm works by iteration. Assuming that exp(t) is supposed to be computed. The iteration starts from an initial pair of L1 and E1. Make L1 = t, consider the iteration

\[
L_{n+1} = L_n - \ln(1 + d_n 16^{-n}),
\]

\[
E_{n+1} = E_n (1 + d_n 16^{-n}),
\]

where \(d_n\)s are a sequence of integers, \(d_n \in \{-a, -a+1, \ldots, 0, 1, \ldots a\}\).

When \(t \in [\sum_{n=1}^{\infty} \ln(1-a16^{-n}), \sum_{n=1}^{\infty} \ln(1+a16^{-n})]\), if a sequence \(d_n\) can be found such that \(L_n\) goes to 0, then \(E_n \rightarrow E_1 \exp(t)\) is obtained:

\[
n \rightarrow \infty, \\
L_n \rightarrow 0, \\
E_n \rightarrow E_1 \exp(t).
\]

For the iteration starting from \(n = m\), the convergence domain is \([\sum_{n=m}^{\infty} \ln(1-a16^{-n}), \sum_{n=m}^{\infty} \ln(1+a16^{-n})]\), denoted as \([s_m, r_m]\). The value of \(a\) will affect the convergence domain. Following Ercegovac [28], \(a = 10\) is selected in this design. Thus the convergence domain starting from \(n = 1\), i.e. \([s_1, r_1]\), is \([-1.02, 0.53]\).

In order for \(L_0\) to converge to 0, the value of \(d_n\) should be chosen to meet \(L_{n+1} \in [s_{n+1}, r_{n+1}]\). Fig. 1 is called Robertson diagram, showing an example for \(a = 1\) and depicting the relation between \(L_n\) (X-axis) and \(L_{n+1}\) (Y-axis) under different value of \(d_n\), and their convergence domain \([s, r]\).

If \(L_n\) is in an overlapping region between \(d_n = -1\) and \(d_n = 9\) (P point in Fig. 1), \(d_n\) can take both 0 or -1 so that \(L_{n+1}\) falls in \([s_{n+1}, r_{n+1}]\). The projections on the X-axis of adjacent values of \(d_n\) overlap with each other for \(n > 2\). In other words, for \(n > 2\), there are always a \(d_n\) that makes \(L_{n+1}\) falls in its convergence domain \([s_{n+1}, r_{n+1}]\). However, when \(n = 1\), there are non-overlapping regions in \([s_1, r_1]\). When \(d_1\) makes \(L_2\) falls in \([s_2, r_2]\). Thus for \(n = 1\) of the iteration has to differ from those for \(n > 1\). Changing the step for \(n = 1\) to:

\[
L_2 = L_1 + \ln(1 + d'_1 32^{-1}),
\]

where the range of \(d'_1\) depends on the reduction method, i.e., the range of \(L_1\). Thus, for the full range of \(L_1\), \(L_2\) falls in \([s_2, r_2]\) for a certain value of \(d'_1\). The method will be further discussed in section 3.

2.2 Existing Range Reduction Method

The existing method utilizes the following expression:

\[
\exp(x + k \ln(2)) = \exp(x) \times 2^k,
\]

where \(k\) is a integer. Define reduced argument \(x^* = x + k \ln(2)\). The reduction is to find the \(x^*\) in the convergence domain. The computation can be decomposed into 3 steps:

1) Find the \(k\) and compute \(x^*\) such that \(x^* \in [0, \ln(2)]\); 2) Compute \(\exp(x^*)\); 3) Compute \(\exp(x) = \exp(x^*) \times 2^{-k}\) by shifting.

The 1) and 3) are called reduction and reconstruction, respectively. In order to find a \(k\) such that \(x^* \in [0, \ln(2)]\), make \(k = \lfloor x \times 23^{-1} \rfloor\) since \(23^{-1} \approx 1/\ln(2)\). Note that this method doesn’t utilize \(E_1\) and set it to 1.

In this method: step 1) needs a multiplier; step 2) needs another multiplier and an adder; step 3) needs a shifter. Reduction and reconstruction consume one clock cycle each.

3. Preliminary and Proposed Range Reduction Method
3.1 Preliminary
While 8-bit fixed-point for inference and 16-bit floating-point for training has become common practice in deep neural networks (DNNs) [29], 32-bit data are more popular in SNNs modeling [17, 24, 30]. These data are used to characterize various parameters of the neuron model such as membrane potential, current and conductance.

Our neuromorphic chip is a 32-bit processor-based system, and using the fixed-point data format s16.15 (A sign bit, 16 integer bits and 15 fractional bits). The biggest and smallest positive values of this format are 65535 (2^{16}) and 0.000031 (2^{-15}), respectively, so the full range of input x is ln(0.000031) to ln(65536):

\[ x \in [-10.38, 11.09]. \]

Since the negative binary number are in two’s complement, for an x of s16.15 format that falls in this range, its upper 13 bits, \( x[31:19] \), are all 0s if positive, all 1s if negative.

As the previous lines mentioned, radix-16 algorithm has a convergence domain of \([-1.02, 0.53]\) when \( a = 10 \), which is much narrower than the full input range. Hence, the input \( x \) must be reduced.

3.2 Proposed Range Reduction Method
An low-latency and hardware-efficient range reduction method for shift-and-add algorithms is proposed in this paper. The comparison between the existing method and this proposed method is illustrated in Fig. 2. In this method, the reduced argument \( x^* \) is obtained by truncating the upper 18 bits of input \( x \), including the sign bit, all 16 integer bits and 1 fractional bit, so the remaining 14 fractional bits are:

\[ x^* = x \& (0000\_3FF\_F)_{16}, \quad (3) \]

where \& means bitwise AND, and subscript 16 means hexadecimal. Define \( \delta \) so that:

\[ x^* = x + \delta \quad (4) \]

It’s obvious that \( x^* \in [0, 0.5) \) regardless of whether \( x \) is positive or not; meanwhile, it is in the convergence domain \([-1.02, 0.53]\) of the aforementioned algorithm. Then \( x^* \) is used as the input of the shift-and-add algorithm to obtain \( E_1 \ exp(x^*) \). It can be found that

\[ E_1 \ exp(x^*) = E_1 \ exp(x + \delta) = E_1 \ exp(x) \ exp(\delta). \quad (5) \]

If make \( E_1 = 1/\exp(\delta) \), the output of the algorithm turns out to be:

\[ E_1 \ exp(x^*) = \exp(x). \quad (6) \]

\( E_1 \) is obtained from a LUT. Because \( x \in [-10.38, 11.09] \) and \( x[31:19] \) are always the same, the design uses 6 bits in the upper segment of \( x \) (\( x[19:14] \)) as the index, including 1 sign bit (\( x[19] \)), 4 integer bits (\( x[18:15] \)) and 1 fractional bit (\( x[14] \)). As a result, the computation process is reduced into two steps:

1) Take the lowest 14 bits of the input \( x \) as \( x^* \), and the upper bits (\( x[19:14] \)) as the index of \( E_1 \) LUT;

2) Compute \( E_1 \ \exp(x^*) \), which equals \( \exp(x) \).

To sum up, the proposed method requires only a 43×40-bit LUT of \( 1/\exp(\delta) \), where \( \delta = \{ -10.5, -10, \ldots, 10.5, 11 \} \). Besides, the number of entries in the \( \ln(1 + d_i/32) \) table is reduced by about one-third since the proposed method reduces \( x \) to \([0, 0.5) \), which is smaller than \([0, \ln(2)] \) in the existing method. In addition, two clock cycles are saved for each exp calculation compared with [24] as its reduction step doesn’t consume any extra cycles and it doesn’t involve a reconstruction step.

4. Implementation

Fig. 3 shows the block diagram of the design. In the diagram, the left part shows the reduction module and the iteration of \( L_n \), i.e. \( L_{n+1} = L_n - \ln(1 + d_n 16^{-n}) \), while the right half is responsible for the iteration of \( E_n \), i.e. \( E_{n+1} = E_n (1 + d_n 16^{-n}) \).

In the solutions of neuron model equations for some popular models like [2] and [3], \( x \) in the \( \exp(x) \) terms is always minus, making the results less than 1. For the results less than 1 in s16.15 format, only the lowest 15 of fewer bits would be non-zero. So 15-bit precision, i.e., the upper 15 non-zero bits of the results are accurate, meets the requirements. In this case, we choose 4 iterations, which can achieve 16-bit precision. The relationship between the number of iterations and accuracy will be discussed in section 6..

Fig. 3 omits the clock, reset and other control signals. There are only one input \( x \) and one output \( \exp(x) \) in the figure, both are in s16.15 format. The whole design consists of 3 parts: the reduction, \( L_n \)'s iteration and \( E_n \)'s iteration.
Input $x$ first passes through the reduction module. The module divides the input $x$ into 2 segments: the upper segment $\times[19:14]$ serves as the index of $E_1$ LUT, and the lower segment $\times[13:0]$ is the reduced argument $x^\ast$. Note that $x^\ast = L_1$.

The mission of $L_n$’s iteration is to offer $d_n$. For $n \geq 2$, $d_n$ can be obtained by rounding to the nearest integer the number obtained by truncating $16^n L_n$ after its fifth fractional digit, referring to [25]; for $n = 1$, $d'_1$ belongs to $\{0, \ldots, 15\}$ and is obtained from a LUT ($14\times 5b$) with $\times[13:10]$ as index. Specifically, $d'_1$ can take 14 values, ranging from 0 to 20; $d_2$ belongs to $\{-8, \ldots, 8\}$; $d_3$ and $d_4$ belong to $\{-10, \ldots, 10\}$. The value of $\ln(1+d_n 16^{-n})$ is obtained from precalculated LUTs, whose size are shown in Table I. Note that the $\ln(1+d3/4096)$ LUT, drawn in Fig. 3, is omitted in practice for the reason that $\ln(1 + d3/4096)$ is small enough in our data width so that:

$$\ln(1 + d3/4096) = d3/4096.$$

Thus its value can be obtained directly from $d3$. LUTs are implemented with logic gates rather than SRAM for 3 reasons: first, the memory compiler in the library doesn’t support generating such small SRAMs; second, for small LUTs, using logic gates is much more hardware-efficient than using SRAM; and last, RAM is volatile and needs to be initialized at power-up. Table I compares the area of these two approaches. Note that the SRAM area in the table doesn’t involve overheads, such as decoders and buffers, which are actually much larger than the array area in a small-capacity SRAM.

**Table I.** Area of LUTs Implemented with Logic gates vs. with SRAM ($\mu$m$^2$)

<table>
<thead>
<tr>
<th>LUT Description</th>
<th>Logic Gates</th>
<th>SRAM$^*$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E_1$ LUT ($43\times 40b$)</td>
<td>1294</td>
<td>901</td>
</tr>
<tr>
<td>$\ln(1+d1/32)$ LUT ($16\times 14b$)</td>
<td>99</td>
<td>117</td>
</tr>
<tr>
<td>$\ln(1+d2/256)$ LUT ($21\times 10b$)</td>
<td>57</td>
<td>110</td>
</tr>
</tbody>
</table>

$^*$: SRAM cell array only, not involving overheads.

The right half, responsible for $E_n$’s iteration, is reused temporally and acts like a multiply-and-accumulator (MAC). The dashed box is similar to a multiplier, which takes $d_n$ as the multiplier and the pre-shifted $E_n$ as the multiplicand. Carry-saved adders (CSAs) are adopted in the adder chain to shorten the critical path. To simplify the multiplication process, $d_n$ is a signed binary number, instead of a two’s complement number. $d_n$ determines which partial sums are output from the shifter to be accumulated. In the case that $d_n$ is negative, the intermediate result is bitwise inverted at the Inverter, and added one at the clocked adder. At the last step, the final result $E_5$ is output after being converted from carry-save to binary by a binary adder.

**5. Error Analysis**

This section analyzes the error of the design theoretically and compares it with simulation results.

As mentioned in section 3., we choose 4 iterations and $E_5$ is the final result. Define $\Delta E = |\exp(x) - E_5|$. Because $d_n 16^{-n} \ll 1$,

$$L_5 = L_4 - \ln(1 + d_4 16^{-4}) \approx L_4 - d_4 16^{-4}.$$

As mentioned in section 3., for $n \geq 2$, $d_n$ is obtained by rounding $16^n L_n$ to the nearest integer, i.e., $16^n L_n - d_n \leq 1/2$, thus:

$$L_{n+1} \leq \frac{1}{2 \times 16^n} = 2^{-(d_n+1)}.$$  \(7\)

As a result, $L_5 \leq 2^{-17}$, the maximum absolute error is

$$\Delta E_{\text{max}}(x) = |\exp(x) - \exp(x \pm 2^{-17})|.$$  

Since the derivative of $e^x$ is itself, it’s easy to find that for $x$ in the full input range:

$$\frac{\exp(x)}{\Delta E_{\text{max}}(x)} \equiv 2^{17}.$$  \(8\)

It can be found from Eq. (7) and Eq. (8) that each iteration of the radix-16 algorithm obtains a 4-bit more accurate result, forming the basis for choosing 4 iterations, which reach 16-bit precision. For $x = 1.4$, $\exp(x) \approx 2^1$, the max error $\Delta E_{\text{max}} \approx 2^{-15}$, which is the resolution of s16.15 format; for $x = 11.09$, the upper limit of $x$, $\exp(x) \approx 2^{16}$, $\Delta E_{\text{max}} \approx 0.5$.

**Fig. 4.** Error simulation for the full input range. The 1 in the y-axis scale represents the value of 1 LSB ($2^{-15}$).

The design is verified in the full input range with Synopsys VCS to observe the error, shown in Fig. 4. The x-axis represents the input $x$ increasing from -10.38 to 11.09 by fine steps, and the y-axis represents the absolute error of the result in log10, i.e., $\lg |\exp(t) - E_5|$. The simulation shows that for $x > 1.4$, the error conforms to Eq. (8); for $x < 1.4$, the error is less than one least significant bit (LSB). The simulation results indicate that this accuracy of the design meets our requirement of SNN neuron modeling, where input $x$ is always less than 0. For $x > 1.4$, the upper 16 non-zero bits of results are accurate (Eq.(8)), thus the results are still reasonable. The larger-than-LSB error when $x > 1.4$ is due to only 4 iterations are performed, which achieves 16-bit precision. It’s feasible to reach higher precision by performing more iterations, but it doesn’t match our needs for SNN modeling.

**6. Results and Comparison**
6.1 Implementation Results

The design is fabricated in SMIC 55 nm CMOS process (SCC55NLL_HD_RVT) with Synopsys’ Design Compiler (DC) and IC Compiler (ICC). The layout of the design takes up a core area of $110 \times 110 \mu m^2$ at 87% utilization. To find the optimal speed, the clock period in timing constraints of DC decreases from 9 ns to 4 ns and the area is recorded after syntheses, as shown in Fig. 5. It fails to meet the timing constraint when the period is less than 4 ns. Fig. 5 indicates that the area increases sharply when the period is less than 5 ns. As a result, 5 ns, i.e., 200 MHz is chosen as the speed of this design to attain the balance between speed and area. The power consumption is analyzed in the typical process corner at the worst condition of 1.08 V, 125 °C.

6.2 Comparison

This work is compared with several of peers’ work in recent years, tabulating in Table II. The comparison shows that with the proposed method, this work achieves the highest throughput and the lowest power consumption for 32-bit data. The area and speed are also outstanding despite the difference in processes. In terms of latency, compared with [24] which uses a radix-2 shift-and-add algorithm with the existing range reduction method, our design is 33% and 20% faster for 16 and 32-bit precision results with the proposed range reduction method. Specifically, for 16-bit precision, our design consumes 4 cycles, while [24] consumes 6; for 32-bit precision, ours consumes 8 while [24] consumes 10 cycles.

Kim’s work [26] achieves a very high speed and throughput. However it’s source-consuming since it includes two 256×16b LUTs and one 64×12b LUT, and two multipliers (16b×16b and 12b×12b). Besides, the input $t$ in $e^{-\tau t}$ is limited in $[0, \infty)$, and only when $\tau = 1$ can it achieve 4 cycles per exp and the accuracy to $2.1 \times 10^{-5}$. By contrast, our work uses rather small LUTs and an equivalent 40x5b multiplier. Jadhav’s work [27] firstly converts the single-precision input to a 32-bit fixed-point number $Z$, then slices $Z$ into 4 segments of the same width. By utilizing $e^Z = e^{Z_a} \cdot e^{Z_b} \cdot e^{Z_c} \cdot e^{Z_d}$, the implementation is source-consuming for employing 4 256x32b LUTs and 3 floating-point multipliers.

Nilsson [22] employs 6 stages Taylor series, with 5-stage multiply-add units in series, to obtain 16-bit $e^x$. However this polynomial approximation method limite $x \in [0, 1]$, thus the output $e^x \in [1, e]$.

7. Conclusion

This paper proposes a novel range reduction method that can increase throughput and save die area for shift-and-add algorithms for exponential functions. An accelerator with the proposed method and radix-16 algorithm is implemented. The results show that compared with the existing reduction method, this method reduces the latency (cycles per calculation) by 33% and 20% for 16 and 32-bit precision results, respectively. CSAs are adopted to speed up the adder chain so that it achieves 200 MHz in SMIC 55 nm CMOS process. It also achieves the lowest power consumption of 4.6 pJ/exp, and excellent area.

For future work, we are going to optimize the accuracy and introduce a dynamic iteration count mechanism to balance the latency and accuracy. Speed improvement and RTL design streamlining are also meaningful, by adopting Booth
multiplication algorithm, the data path in the shift-and-add block can be further shortened.

Acknowledgments

This work was supported in part by the Science and Technology Innovation 2030 – Major Project, 2020AAA0109102.

References