ARITHMETIC ACCELERATORS FOR A DIGITAL NEUROMORPHIC PROCESSOR

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF SCIENCE AND ENGINEERING

2020

Mantas Mikaitis

Department of Computer Science

Contents

Word count: 44496

List of Tables

List of Figures

Abstract

ARITHMETIC ACCELERATORS FOR A DIGITAL NEUROMORPHIC PROCESSOR
Mantas Mikaitis
A thesis submitted to The University of Manchester
for the degree of Doctor of Philosophy, 2020

This work explores a programmable accelerator for computing exponential and natural logarithm, functions that are common in Spiking Neural Network (SNN) models, in the context of SpiNNaker neuromorphic chip. The accelerator is integrated in the SpiNNaker2 chip, and the energy, area, and numerical accuracy tradeoffs are evaluated. An early version of the accelerator with *fixed-point* arithmetic was included in a prototype SpiNNaker2 chip and tested in silicon, while the final version with *floating point* is, at the time of writing, in manufacturing as part of another SpiNNaker2 prototype chip. Software techniques for improving the accuracy of the exponential function included in the very first SpiNNaker2 prototype chips are also presented.

Furthermore, a problem of simulation results being different on SpiNNaker from those obtained using floating-point arithmetic is explored. Numerical accuracy of Ordinary Differential Equation (ODE) solvers for the *Izhikevich* neuron model, which was previously shown to be a major challenge in fixed-point arithmetic, is addressed. Any simulation of a physical system has multiple sources of errors, which include errors in measurements, models, numerical methods, and finite-precision computer arithmetic. Here the last source of error is addressed; it is shown, using the Izhikevich neuron on SpiNNaker, that various problems with fixed-point arithmetic caused arithmetic error to be substantially larger than expected with 32-bit data types. Improvements are found by utilizing rounding of the constants (on decimal to fixed-point format conversion), rounding of the multiplier results, and usage of *mixed-precision* operations. The *stochastic rounding* method, which rounds with the probability proportional to the distance between numbers, is shown experimentally to improve the accuracy of a series of ODE solvers beyond the standard rounding routines. As a result, a hardware accelerator for the SpiNNaker2 chip is explored to speed up this rounding method.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

I would like to thank my supervisor David Lester and co-supervisor Steve Furber for support and guidance on this research. I especially appreciated the freedom to explore any research topic that attracted me and the opportunities to contribute to the SpiNNaker2 chip. I am also very grateful to David Lester for providing the constants used in range reduction of the exponential functions and Steve Furber for pointing out ways to parallelize the iteration circuit of the exp-log accelerator.

Gengting Liu and Delong Shang suggested logic optimizations in the iteration circuit of the exp-log accelerator, for which I am thankful for. Also, I am grateful to Andreas Dixius, Sebastian Höppner, and Stefan Scholze at the Technical University of Dresden who answered my queries about the SpiNNaker2 design and test infrastructure. Furthermore, I thank Jim Garside for many useful discussions as well as Garibaldi Pineda García, Michael Hopkins, Jamie Knight, and Oliver Rhodes for the fruitful collaborations in various projects. Many thanks also to Andrew Brown, Steve Furber, Michael Hopkins, Dirk Koch, David Lester, and John V. Woods for reading the early drafts of the thesis — their feedback improved the conciseness and quality of this work significantly.

I acknowledge funding from the Engineering and Physical Sciences Research Council and the European Union that enabled this research to be carried out.

Finally, thanks to Simona for being there and for all the evenings at the movie theaters, which provided opportunities to make brief pauses from bit twiddling; and my family for all the support.

Chapter 1

Introduction

The SpiNNaker chip and its software have been in development since around 2006. A second generation SpiNNaker chip is currently in development. Both chips are based on general-purpose ARM processors, while the neuromorphic property comes from the connectivity of large numbers of these ARM processors using networks-on-chip (NoCs) and routers that control the traffic of neuronal spikes around the machine; and the software — specifically, an event-driven programming model which allows the ARM cores to send signals, enabled by the NoC and router, and trigger some computation at the target ARM processors.

This chapter introduces the main aspects of these projects, with the discussion about neuromorphic computing in general. It is pointed out that the significant cost of using general purpose processors without support for specialized functions is a serious limitation for simulating SNNs with plasticity. The main goal of the thesis is to explore arithmetic hardware accelerators with multiple numerical formats and accuracy control for SpiNNaker2, while another goal of the thesis is to find methods for improving the accuracy of the models when low precision fixed-point arithmetic is used instead of floating point, as is the case with the current generation SpiNNaker machine.

1.1 Neuromorphic computing

Understanding the human brain is one of the biggest challenges facing scientists in recent years. First, with the rise of ideas and need for artificial intelligence (AI), understanding the human brain can help us build better-performing AI. Today's AI algorithms are quite limited compared to the brain: the most common techniques (just to

mention a few of many in AI) are based on large functions that can classify diverse data (*deep neural networks*) and *reinforcement learning* which works on the principle of "learning from experience and feedback". However, these frameworks are rather limited in the sense that they usually do not work when moved to a new, arbitrary environment requiring human intervention to retrain the classifiers or reprogram the reinforcement learning agents to determine what is a mistake and what is a reward.

Another use case for understanding the brain better, and simulating it, is medical. Diverse brain diseases that are not well understood due to the sheer complexity of the brain usually do not have a known, generally applicable cure. Simulating these diseases in a well-defined and controlled environment on a computer would provide researchers with easier access for studies.

However, without even fully understanding the brain, we are now quite sure that using massive clusters of classical general-purpose computers is not going to be feasible in terms of achieving at least biological simulation run-time and manageable electrical power requirements. One of the goals of *neuromorphic computing* field is to enable fast and energy-efficient simulation of large scale neural networks with complex behaviour [1]. This research field was started by Mead [2] with an observation that the brain operates on analogue principles and is quite different from digital computers. He demonstrated silicon neurons, mixed-signal circuits which replicate the properties of the biological neurons directly in circuits rather than numerical simulations on digital computers. Since then, various modern neuromorphic computers have been created following those principles [3] but digital neuromorphic simulators have also been in development [4] for greater programmability, and as platforms for research into neural network models.

Neuromorphic chips and large-scale systems that include them are designed to exploit massive parallelism, using efficient networks-on-chip, to simulate artificial neural networks, optimized for a specific type of neural network called SNN. *SpiNNaker*, completed in 2011, is one such computer, belonging to the digital neuromorphic chip category [5, 6, 7]. Utilizing small, energy-efficient ARM cores, connected in one large network, SpiNNaker can simulate large-scale, fully programmable[1] SNNs in real-time and with low energy usage. The largest network simulated so far on SpiNNaker with Spike-Timing-Dependent Plasticity (STDP) [8] learning rules has 20 000 neurons and

[1] "Fully programmable" refers to the difference between platforms like SpiNNaker which has software neuron and synapse models and analogue neuromorphic platforms that have the models fixed in hardware with minimal configurability.

51 million synapses [9] while the largest network without learning has 80 000 neurons and 300 million synapses [10]. However, as neuroscience advances, increasingly complex neural network algorithms, especially learning algorithms describing synaptic plasticity, have been discovered; some have been simulated on SpiNNaker with the realization that real-time replication of biological learning models is challenging. In one recent experiment to simulate a reinforcement learning plasticity model (one of the most complex plasticity models in terms of arithmetic requirements run on SpiNNaker to date) [11] it was possible to simulate only 10 000 neurons and 10 million synapses before hitting real-time performance limits. This was mainly due to how the networks were scaled up, which was done in a such a way that the number of incoming synapses per neuron was dependent on the overall network size.

While SpiNNaker is based upon a fully programmable digital neuromorphic chip, other approaches try to replicate neurons and synapses in the analogue domain, utilizing hardware that simulates certain chosen models faster than using software. That being noted, fully programmable neuromorphic chips are well suited for experimental work, at the stage when a modeler of a specific neural network is experimenting with neuron and synaptic plasticity models to achieve some behaviour, whereas analogue neuromorphic chips are suitable for well-defined neural networks when the costs of manufacturing such networks in hardware are known to pay off. Furthermore, some neuromorphic chips are made as hybrids between analogue and digital [12] to allow some part of the whole simulation algorithm to be fully programmable. Other solutions are purely digital, but with limited programmability of supported models and limited numerical precision [13], most likely to provide a faster, but still programmable, solution to simulate neural networks. In summary, in contrast to more classical neuromorphic chips, SpiNNaker is a *fully programmable SNN simulator, with high precision arithmetic available* (where high precision is a 32-bit numerical format and a set of hardware arithmetic operations available on such a format), and this feature allows it to support most of the neuron and synapse models expressed as numerical algorithms.

1.2 Simulation errors

Any simulation of a physical phenomena suffers from various sources of errors: observation error, when experiments are made and data about some physical system are collected; error in a mathematical model that is created to describe the system's behaviour; error in numerical method used to advance the model's state; and error in

the data conversion and operations performed in finite-precision computer arithmetic, such as fixed- or floating-point arithmetics. In this thesis the last source of error is addressed, for three reasons.

The first is that there are studies that show major differences between the results of fixed- and floating-point arithmetics in simulating neuron models with equivalent numerical methods [14, 15]. These differences are more significant than expected with a 32-bit arithmetic and this source of error is dominating when compared with the error caused by the numerical method used to solve differential equations [14]. The second reason is that by default, the absolute rounding error of fixed-point arithmetic is $2^{-15} \approx 0.3 \times 10^{-4}$, which is a significant error and will accumulate if the rounding mode is unidirectional, as shown in Chapter 3. The third reason is reproducibility — while most of the neural network simulators enjoy single- or double-precision floating-point arithmetics, SpiNNaker uses fixed point, and to increase reproducibility, fixed-point arithmetic can be optimized in various ways to produce results that are closer to floating point, making SpiNNaker results closer to those of other simulators [10, 15]. This is important both when the same numerical method is used (very close reproducibility possible as arithmetic error is close to zero) and when a different numerical method is used on SpiNNaker (numerical method error dominating, instead of arithmetic as is the case currently).

1.3 Arithmetic in digital computers

It was established that SpiNNaker is a digital computer and that it should provide a full accuracy option for simulating SNNs. Digital computers, including SpiNNaker, require arithmetic operations to be implemented in an algorithmic way, working on a defined length of input vectors containing bits laid out in a manner dictated by a chosen numerical format to approximately represent various physical quantities. This is in contrast to analogue chips, that can implement specific model behaviours physically. A set of the most common operations on integers, such as $+$, $-$, $\times$, *shift* are usually provided in hardware on general-purpose processors. However, when real numbers are required, those instructions might not always work in the same way and might require some extra operations such as *round* and *saturate*. If a fixed-point numerical representation is used, most integer operations available on a processor will work, with slight modifications and algorithmic implementation of rounding and saturation.

On the other hand, if floating-point arithmetic is chosen, none of the hardware integer operations will work directly and therefore the arithmetic library either has to be implemented using integer instructions or new hardware has to be introduced.

More complex operations such as sin, cos, exp, log, or pow are usually implemented algorithmically in software using in the order of hundreds of processor instructions, depending on numerical format and accuracy [16]. In fields outside neuroscience modelling and simulation it has been shown that some of the complex mathematical functions dominate the overall computational time of certain algorithms. One example is CERN's Large Hadron Collider (LHC) Physics experiments [17, 18, 19] which demonstrate high utilization of *dividers* for trigonometric functions in software, and elementary functions; the exponential function is reported to take up to 60 % of run time in some physics experiment reconstruction algorithms. Similarly, datacenter applications reported by Microsoft Research [20] require a lot of log, exp and other functions. If measured on a complex neuron and plasticity simulation with a small timestep, SpiNNaker would most likely show a high usage of exponential function as well (no such model is currently available on SpiNNaker — 1 ms timestep and fixed synapses are used in most networks). Exponential is used in many places in SNN models to exponentially decay various quantities on each timestep update, and sometimes on each spike. On SpiNNaker2 prototype [21], a plasticity model was explored which was shown to use approximately 43 % of processor cycles available per simulation timestep for computing software exponential function.

SpiNNaker is based on an ARM968 processor, which supports integer-only operations. Fixed-point arithmetic is implemented by interpreting integers as fixed-point real numbers with appropriate shift on conversion and, on certain arithmetic operations, to control where the binary point is. SpiNNaker2, on the other hand, will support single-precision floating-point numbers with a hardware Floating-Point Unit (FPU). However, one limitation of such a unit is that it keeps the intermediate results of any arithmetic operation inside the FPU and therefore does not allow for experimenting with rounding options apart from those that it supports (which are IEEE 754 standard [22] compliant) or mixed-precision arithmetic algorithms. On the other hand, when working with fixed-point values, one can control the desired rounding quite easily.

Both SpiNNaker and SpiNNaker2 are 32-bit processors, and all arithmetic operations work on 32-bit operands. In some cases, it might be useful to hold intermediate results in a wider length, across two registers, and utilize rounding at a chosen point in the algorithm — the core idea of the aforementioned mixed-precision algorithms.

In other cases, memory requirements have to be considered and a word length lower than 32 bits chosen, in which case rounding is utilized again before writing the value to memory.

Lastly, the terms *accuracy* and *precision* should be addressed here. In this thesis we will usually refer to precision as being *the number of bits in a numerical format* or *the number of bits in a fractional part of the numerical format*. On the other hand accuracy is addressed after precision is defined, that is, in terms of numerical error analysis, accuracy is *how well does an algorithm utilize a given numerical precision to represent some output when compared to some ideal algorithm using the same or higher precision data type*; in other words, how well does it set up the number of bits in a data type to minimize the numerical error. It is worth commenting that definitions can differ, depending on the context. Higham [23, p. 6], for example, defines *precision* as the "accuracy with which basic arithmetic operations $+, -, *, /$ are performed". Note that, in terms of neuromorphic computing, accuracy can also be used to discuss neural network performance and this should not be confused with numerical accuracy.

1.4 Energy consumption and the role of accelerators

While biological neural networks run in real time irrespective of their scale, SNN simulation performance in a digital computer is directly related to the size and complexity of the network. This is easy to see when imagining that each neuron is mapped into Central Processing Unit (CPU) instructions — the more neurons one tries to update in each timestep the more computation time is required. Neuron processing can be spread across multiple CPUs, but there is always some level of sequential execution (for example updating a single neuron) which cannot be parallelized and eventually is a limiting part of the overall parallel system's performance (Amdahl's law). Similarly, if each spike has the cost of a certain number of CPU instructions attached to it, scaling the network without repartitioning where the neurons are laid out will result in more spikes (instructions) arriving into CPUs.

As discussed by Jin [24], most of the methods to speed up simulations of neural networks generally lie in two categories: 1) developing simpler neuron and synapse models, and 2) increasing the performance of the simulation hardware; most effort in the first category was placed in developing simpler neuron models. One of the most biologically plausible neuron models is the *Hodgkin-Huxley* model, which takes 1200 arithmetic instructions per update [25]. Izhikevich [25] demonstrated increasing

biological implausibility with decreasing neuron model complexity and cost in arithmetic operations, eventually reaching a simple 5-arithmetic-operation *Integrate-and-Fire* neuron model. The second category is usually a choice between a neuromorphic analogue/digital hardware platform, a large-scale high-performance CPU or a Graphics Processing Unit (GPU) clusters, or Field-Programmable Gate Array (FPGA)s, with multiple choices of software simulators on each platform.

With the growing complexity of neuron and synaptic plasticity models, and new information from neuroscience about the sparsity of networks and neuron fan-in/out, machines such as SpiNNaker will need to meet increasing low energy and real-time performance demands. The next major step is to investigate specialized compute engines [26] to be placed next to the Processing Elements (PEs) inside SpiNNaker chips to accelerate different parts of the neural network algorithms, most importantly, serial parts of the simulation flow that cannot be parallelized by using more cores. Here *accelerated* means computation not run in a general-purpose CPU but specifically cast into an (algorithm-specific) hardware unit that runs more efficiently than the CPU (with higher accuracy, lower latency and energy), but without compromising the flexibility of the algorithms that can be executed (otherwise the accelerator risks becoming deprecated where users would start replacing it with software for programmability).

It is not always the case that high accuracy and a complex number representation format such as floating point is required. Some studies are already being done in terms of moving away from a CPU that computes everything to the last bit of accuracy [27], and error tolerance of the applications, both at circuit level and algorithmic level, is explored to minimize energy use [28]. On a digital machine with fixed-length instructions, such as SpiNNaker with a 32-bit datapath and registers, trading off accuracy for energy is not easy to achieve. It might be possible to reduce accuracy by modifying the algorithms, for example an iterative algorithm with controllable number of iterations to run, that computes slightly inaccurate 32-bit results. However, the lowest level operations such as $\times$ and $+$ cannot be controlled for reduced accuracy — they are evaluated in the processors' hardware components without the user control, and the actual hardware component has the area and energy usage of a component which computes a fully accurate result to a given precision.

Therefore, in SpiNNaker terms it is possible to trade off accuracy for speed and energy mainly in two ways: SpiNNaker and SpiNNaker2 algorithmic arithmetic functions can be controlled to use fewer atomic integer or floating-point operations, and arithmetic accelerators for SpiNNaker2 can be built with accuracy control in them for

providing an option for making trade-offs. But it is not possible to control the energy used by the ARM processors' multipliers, adders, and other basic functional units.

Another major contributor to energy usage in SpiNNaker is moving data from external memory into the processors, unpacking it, repacking it, and sending it back to the memory. When a spike event arrives at the core, it needs to get all the information about the synapses which is stored in an off-chip memory as it would not fit into the limited space on the core. The data is fetched using a Direct-Memory-Access Controller (DMA) call, and used to compute contributions of synapses to the neuron. Then it is modified if plasticity is enabled in the simulation, and put back into the off-chip memory using a DMA call again. The moving of data and unpacking and packing it back into the form that they are stored in uses a lot of energy in the simulation and some possibilities exist for accelerating this part of the SpiNNaker simulation algorithm. For example, DMA could have some programmable support for unpacking streams of data so that the processor would not need to do it using multiple shift and logical bit masking instructions. These types of accelerators are out of the scope of this work but are worth noting.

1.5 Conclusion

This thesis explores the fixed-point arithmetic of SpiNNaker, and fixed- and floating-point arithmetic, including hardware accelerators, of SpiNNaker2. To summarise the previous sections, the main work will have the following constraints.

- SpiNNaker and SpiNNaker2 are digital neuromorphic computers where neural models are cast into ARM instructions that use underlying full-precision arithmetic hardware to evaluate mathematical equations.

- Both machines are experimental, fully programmable platforms as opposed to analog neuromorphic machines with fixed neuron models. Neuron, synapse and synaptic plasticity models are described using the ARM instruction set.

- The core building block of the SpiNNaker chip is the ARM968 integer processor, whereas SpiNNaker2 is based on the ARM Cortex-M4F containing an FPU.

- The SpiNNaker2 FPU does not provide the user access to the intermediate values in higher precision than single-precision floating point. Any operation rounds the

intermediate values to floating-point using one of the rounding modes from the IEEE 754 standard [22].

Working with these constraints, the main goal of this work is to address the arithmetic of SpiNNaker to understand how accuracy-energy tradeoffs can be made in large-scale simulations of SNNs. The main contributions are as follows.

- Techniques to improve the accuracy of neuron models on the current generation SpiNNaker, when fixed-point arithmetic is used (Chapter 3).
- The design and evaluation of a *stochastic rounding* accelerator for SpiNNaker2, supporting fixed-point and floating-point rounding with saturation in single cycle plus two memory operations. Stochastic rounding is demonstrated to provide substantial accuracy improvements in fixed-point neural ODE solvers on SpiNNaker (Chapter 4).
- The design and evaluation of an exponential and logarithm function accelerator for SpiNNaker2, in fixed- and floating-point, with programmable accuracy for energy-accuracy tradeoffs. The first version (fixed-point only) of this accelerator was manufactured as part of a prototype SpiNNaker2 test chip (Chapter 5).

The advantages of SNNs are still debated in the neuromorphic community. It is generally assumed that because the brain can perform much more substantial tasks than any current machine learning algorithms, SNNs, that very closely resemble biological neural networks, will be the most popular approach in the future. In this work this assumption is also followed and the question of SNN superiority over other approaches is not addressed. As a result, the thesis is written from a computer science and engineering perspective, having in mind that any algorithms that are currently run on neuromorphic machines are simplifications of some brain activities that are described by neuroscience. The general strategy of the work is to *survey the neuron and synapse models available in computational neuroscience literature, focusing on mathematical detail and algorithms, and provide methods to improve numerical accuracy as well as identify the parts to accelerate in hardware*.

Chapter 2

Background

This chapter contributes two things to the thesis. First, it introduces definitions and properties of fixed- and floating-point arithmetics, ways to measure accuracy of results in each, and rounding modes that are explored in the thesis. The subject of digital computer arithmetic is not covered extensively here and we refer the interested reader to [16, 29, 30, 31, 32] for more details. Both numerical formats are used in Chapters 3–5. The accuracy of the implemented functions (Sections 5.4.1.1–5.4.1.3) are measured using Unit of Least Precision or Unit in the Last Place (ulp) that is discussed here. Also, various rounding methods defined in this chapter are a core part of Chapters 3 and 4. Furthermore, carry-save adders, whose symbols are defined here, are used in the exponential and logarithm function algorithms (Section 5.2.2).

Secondly, the chapter introduces an overview of state-of-the-art neuromorphic simulators (analogue, digital and software simulators on the conventional hardware) in order to give a landscape of this field which forms a major part of the context of this thesis. Only brief details of each simulator are given, focusing on the available neuron and learning models, numerical precision, energy and computational performance. A more detailed SpiNNaker hardware and software review is given, focusing on the aspects and algorithms that provide background for the results in Chapter 3.

Some material in this chapter was reproduced from the material that was published in the Philosophical Transactions of the Royal Society A journal [33].

2.1 Digital computer arithmetic

In this section some definitions and properties of fixed- and floating-point numerical formats are provided.

2.1.1 Fixed-point arithmetic

A generalized fixed-point number can be represented as shown in Figure 2.1. Define a signed fixed-point number $< s, i, p >$ and an unsigned fixed-point number $< u, i, p >$ of word length w (which usually is $8, 16, 32$ or 64) bits where i denotes the number of integer bits, p denotes the number of fractional bits and s is a binary value depending on whether a specific number is signed or unsigned (where u means that there is no sign bit and s means there is a sign bit, and then we have to use 2's complement in interpreting the numbers). p also tells us the *precision* of the representation. Given a signed fixed-point number $< s, i, p >$, the range of representable values is $[-2^i, 2^i - 2^{-p}]$. Whereas, given an unsigned fixed-point number $< u, i, p >$, the range of representable values is $[0, 2^i - 2^{-p}]$.

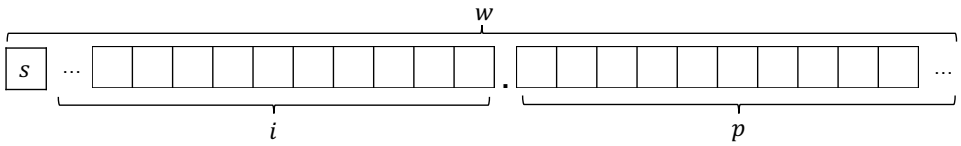

Figure 2.1: The general form of a fixed-point number that is made out of three parts: the most significant bit which is a sign, an integer part with i bits, and a fractional part with p bits. The word length is $w = i + p + 1$.

To measure the *accuracy* of a given fixed-point format (or more specifically some function that works in this format and we want to know how well, how *accurately*, it performs in a given precision), we define *machine epsilon* $\varepsilon = 2^{-p}$ (sometimes also referred to as a value of Least Significant Bit (LSB)). Here ε gives the smallest positive representable value in a given fixed-point format and therefore represents a *gap* or a *step* between any two neighbouring values in the representable range. Note that this gap is absolute across the representable range of values and is not scaled by the *exponent* as in floating-point or similar representations. This requires us to consider only absolute errors when measuring the accuracy of functions that are implemented in fixed-point representation. Accuracy is sometimes also measured as LSB — a value represented by the least significant bit in a fixed-point word, which is the same as machine epsilon. Note that the maximum error of a fixed-point number, when round-to-nearest is used in conversion, is $\frac{\varepsilon}{2}$.

Lastly, it is worth noting how to convert a general fixed-point number into a decimal number. Given a 2's complement fixed-point number of radix 2 (a binary vector) $< s, i, p >$: $sI_{i-1}I_{i-2}\cdots I_0.F_1F_2\cdots F_p$, if the number is signed the decimal value is given

Table 2.1: Minimum and maximum positive numbers of 32-bit fixed-point numerical formats.

Property	s16.15	u0.32	s0.31
Accuracy (abs.)	2^{-15}	2^{-32}	2^{-31}
Min (exact)	2^{-15}	2^{-32}	2^{-31}
Min (approx.)	0.0000305	2.32831×10^{-10}	4.65661×10^{-10}
Max (exact)	$2^{16} - 2^{-15}$	$1 - 2^{-32}$	$1 - 2^{-31}$
Max (approx.)	65535.999969	0.999999999767169	0.999999999534339

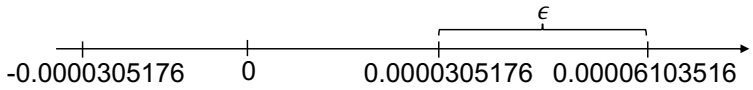

Figure 2.2: Values close to zero in the s16.15 representation.

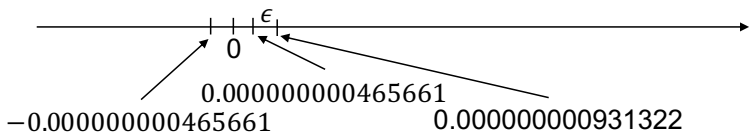

Figure 2.3: Values close to zero in the s0.31 representation.

by summing the non-zero elements multiplied by corresponding weights

$$value = \sum_{k=0}^{i-1} I_k 2^k + \sum_{j=1}^{p} F_j 2^{-j} - s2^i. \tag{2.1}$$

Otherwise, if the number is not signed (so bit s becomes integer bit I_i) the decimal value is given by

$$value = \sum_{k=0}^{i} I_k 2^k + \sum_{j=1}^{p} F_j 2^{-j}. \tag{2.2}$$

SpiNNaker software mostly uses two fixed-point formats standardized by the International Organization for Standardization (ISO) 18073 [34] standard and available in the GNU Compiler Collection (GCC): *accum*, which is $< s, 16, 15 >$ and *long fract* which is $< s, 0, 31 >$ (further referred to without brackets, for example s16.15). Table 2.1 shows some properties of various fixed-point formats. The values close to 0 for each format are shown in Figures 2.2 and 2.3. The s16.15 format has a range of

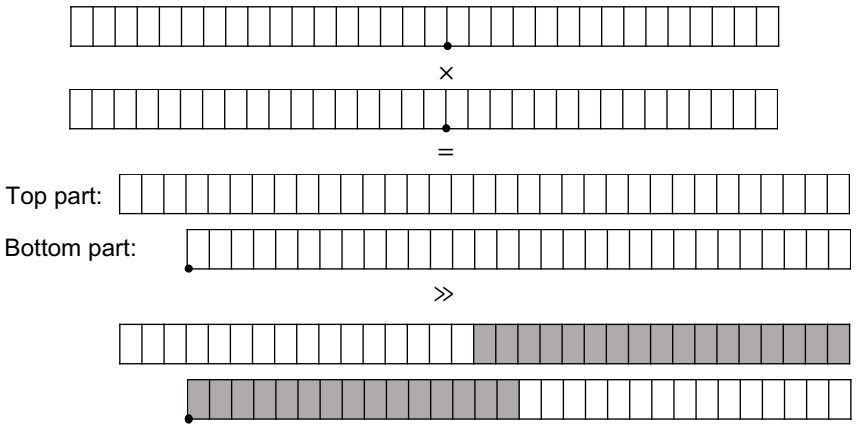

Figure 2.4: Example of the multiplication of two s16.15 variables with the binary point after the 15th bit marked in the arguments and after the 30th bit in the long answer. The shaded part in the long answer is a 32-bit result that has to be extracted, discarding the bottom and top bits.

representable values of

$$[-2^{16} = -65536,\, 2^{16} - 2^{-15} = 65535.99996948...],$$

with the gap between neighbouring values of $\varepsilon_{s16.15} = 2^{-15} = 0.000030517578125$. The s0.31 format has a range of

$$[-1,\, 1 - 2^{-31} = 0.99999999953433...],$$

with the gap of $\varepsilon_{s0.31} = 2^{-31} = 0.00000000046566...$ between neighbouring values. Based on the values of machine epsilon, *long fract* is a more precise fixed-point data type. However, *long fract* has a very small range of representable values compared to *accum*. Which format should be used depends on the application requirements, such as the required precision and the bounds of all variables. Sometimes intermediate values can be held in *long fract* as long as possible and only rounded into *accum* when it is known that certain operations will cause *long fract* to overflow or that the subsequent algorithmic steps require an input number to be an *accum*.

The three main *arithmetic operations* on fixed-point numbers are of interest:

- Addition: $< s, i, p > + < s, i, p > = < s, i, p >$,
- Subtraction: $< s, i, p > - < s, i, p > = < s, i, p >$, and
- Multiplication: $< s, i_a, p_a > \times < s, i_b, p_b > = < s, i_a + i_b, p_a + p_b >$.

Note that $+, -$ and $\times$ denote integer operations available in most processors' arithmetic logic units (ALU), including the ARM968. Therefore, for addition and subtraction, no special steps are required and these operations are exact if there is no underflow or overflow. However, for multiplication, if the operands a and b have the word lengths w_a and w_b, then the result will have the word length of $w_a + w_b - 1$. Therefore after the integer multiplication we have to shift the result right to convert it to the same fixed-point representation as the operands (the example in Figure 2.4 shows this for the *accum* format). This is done because subsequent addition/subtraction operations need the inputs to be in *accum*, or the result from multiplication is an input into another multiplication which, if not rounded, would require a 64-bit multiplication instruction (not always available) and yield results longer than 64 bits, resulting in *bit growth*. Support for storing and operating on such longer data types is not widely available without extended precision libraries such as GNU MPFR [35]. This shifting in the multiplication operation results in *loss of precision* and therefore an appropriate rounding step can be done to minimize the error. Similar issue arises when two numbers in different fixed-point formats are added: if the result from the addition needs to be in the format of the less precise fixed-point input (which it should, otherwise overflow is likely — the input argument with more integer bits can already have a magnitude which will overflow the other argument's format), the bottom bits of the more precise input will not be preserved and have to be rounded.

2.1.2 Floating-point arithmetic

The IEEE 754-2019 standard [22] defines a radix-2 (binary) normalized single-precision floating-point (called binary32) number with a sign bit S, 8-bit exponent E and a 23-bit integer significand T to have the numerical value (slightly modified from [32, p. 51])

$$(-1^S) \times 2^{E-127} \times (1 + T \cdot 2^{-23}), \tag{2.3}$$

whereas a double-precision (binary64) number with a sign bit S, 11-bit exponent and a 53-bit integer significand T has the numerical value

$$(-1^S) \times 2^{E-1023} \times (1 + T \cdot 2^{-52}). \tag{2.4}$$

Table 2.2 shows the minimum and maximum values of various floating-point numerical formats (including bfloat16 [36] which is a floating-point non-IEEE format

equivalent to binary32 in structure but without 16 bottom bits of the significand). As discussed previously, fixed-point formats have absolute accuracy denoted by the corresponding machine epsilon, which means that any two neighbouring numbers in the representable range have a fixed gap between them. On the other hand, floating-point formats have accuracy relative to the exponent, which means that the gap between any two neighbouring numbers (or a real value of the least significant bit) is relative to the exponent that those numbers have — machine epsilon multiplied by 2 to the power of the biased exponent.

This can be imagined by a long axis which contains markers for all possible floating-point numbers (Figure 2.5). Such an axis would have very small gaps between the numbers close to zero and increasingly larger gaps as numbers get further away from zero towards $\pm\infty$. Whereas such an axis for fixed-point would look like all gaps between the markers, where the representable numbers are placed, are all of the same length. For example, the next number after $0.5\,(E = 126)$ in binary32 floating-point arithmetic is $0.5 + 2^{-23} \times 2^{-1}$, while the next number after $1.0\,(E = 127)$ is $1 + 2^{-23} \times 2^{0}$. Due to this, the accuracy of floating-point numbers is measured relative to the exponent (more on this in Section 2.1.3).

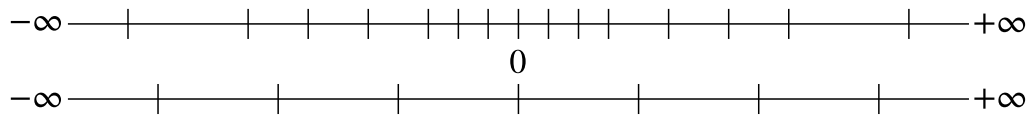

Figure 2.5: A demonstrative example of the difference between the axis of real numbers represented in floating- and fixed-point (bottom) arithmetics.

Due to these features of floating-point arithmetic, it depends on the application which data types will provide more accuracy overall. For example, if the application works with positive numbers below 1 only, u0.32 is a more accurate data type as it gives smaller steps between adjacent numbers (2^{-32}) than the binary32 floating point (2^{-23}). On the other hand, if the application works with numbers that are as small as $0.001953125\,(E = 117$, which gives a gap between numbers of 2^{-32}), binary32 floating-point representation becomes as accurate as u0.32 and increasingly more accurate as the numbers decrease beyond that point, eventually reaching the accuracy of 2^{-126} (normalized) and $2^{-126-23}$ (subnormal) near zero.

Table 2.2: Minimum and maximum positive numbers of floating-point formats. Smallest exact subnormals (sn) are shown in brackets.

Property	binary64	binary32	bfloat16
Accuracy (rel.)	2^{-52}	2^{-23}	2^{-7}
Min (exact)	$2^{-1022}(\text{sn} : 2^{-1074})$	$2^{-126}(\text{sn} : 2^{-149})$	$2^{-126}(\text{sn} : 2^{-133})$
Min (approx.)	2.225×10^{-308}	1.175×10^{-38}	1.175×10^{-38}
Max (exact)	$(2-2^{-52}) \times 2^{1023}$	$(2-2^{-23}) \times 2^{127}$	$(2-2^{-7}) \times 2^{127}$
Max (approx.)	1.798×10^{308}	3.403×10^{38}	3.39×10^{38}

2.1.3 Measuring accuracy: machine epsilon (LSB) and unit of least precision (ulp)

To measure the accuracy of functions implemented in fixed- and floating-point arithmetics we have to consider what is the best that the function can do given the numerical format and the number of bits of precision. Any perfectly performing function can only give answers from the set of numbers that are available in the numerical format of the outputs, and we should choose a number that is as close as possible to an exact answer, without knowing it. Measuring errors is simpler in fixed-point arithmetic, as the set of numbers that it can represent is uniformly distributed in the available dynamic range. So we only have to consider how many steps, that is machine epsilons $\varepsilon = 2^{-p}$, the answer from a given function is from the exact answer of the same function with the same inputs. This measure is sometimes also called LSB, for a *value represented by the least significant bit*. For simplicity, in this thesis this measurement will be called ulp as used in floating-point arithmetic.

In floating-point arithmetic it is not as straightforward, as the set of available numbers is not uniformly distributed. Smaller numbers are separated by smaller steps and larger numbers are separated by larger steps. Due to this, reporting how many machine epsilons we are away from the exact mathematical answer will not be meaningful anymore without taking into account where in the dynamic range the answer is in a given floating-point format. We need to use a measurement that tells us how many *steps* away we are from the exact answer — one step away will have a meaning across the full dynamic range even though one step can be a very large number at the edges of the dynamic range and very small fractional number near 0.

The ulp is one such measure. Multiple definitions of ulp for a general floating-point number are discussed by Muller [37]. For the purposes of this thesis the definition slightly rephrased from William Kahan's [38] definition will be used: $\text{ulp}(x)$, where

x is some exact number of which approximation in floating point we are considering, is *a magnitude of the gap between the finite two nearest floating-point numbers on both sides of* x. However, in most cases we do not need to measure the magnitude of the gaps but just want talk about how many ulps one or the other number is away from some exact value (the real size can be reconstructed for a specific situation if needed for understanding numerical error). Therefore there are two uses of ulp, one as a function $\text{ulp}(x)$ that returns the magnitude of the gap where x lies and one as a quantity for error measurement: 0.5ulp, 1ulp, 2ulps and so on. Note that for numbers that are powers of 2, ulp magnitudes are different on the left and on the right of those numbers due to the exponent change; this is addressed in more detail by Muller [37]. Some interesting scenarios are demonstrated in Figure 2.6 for visualization. Also note that the maximum error of a perfect implementation of some function with correct rounding is 0.5ulp, which means that we always calculate the floating-point number nearest to the theoretical exact value.

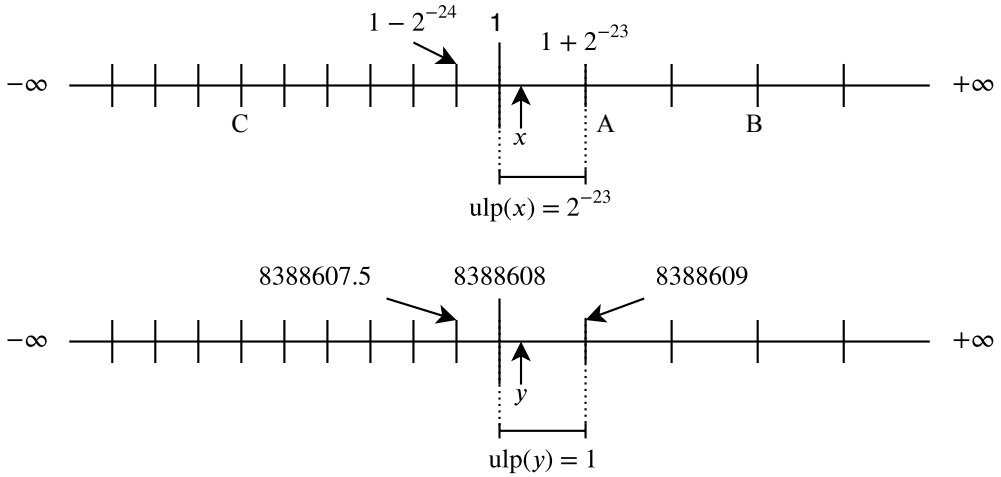

Figure 2.6: Some interesting numbers from the binary32 floating-point representation. Top: some number x is not representable in floating-point exactly, so either 1 has to be returned or the next representable number $A = 1 + 2^{-23}$. If 1 is returned, the accuracy is within 0.5ulp, if one of these is returned, we say that the accuracy is within 1ulp. If B is returned, accuracy is within 3ulps; if C is returned, then accuracy is 7ulps. Bottom: $2^{23} = 8388608$ is the first number going from 0 to $+\infty$ that loses the fractional part. Notice that the gaps between the floating-point numbers after it are equal to 1. However, when some real value y, that cannot be represented in floating-point, is desired and 8 388 608 or 8 388 609 is returned, we still say that we are within 1ulp and therefore very accurate, as accurate as in the top scenario, even though the absolute error is much larger. As long as we are not introducing "GULP" errors (Giga-Ulp) (see the post by Kahan [38] for history) we are fine.

2.1.4 Rounding

Anytime we convert a number to a lower precision number, we have to consider rounding instead of throwing away the bottom bits. Here two known rounding approaches are described, for simplicity using fixed-point numbers: Round-to-Nearest (RN) and Stochastic Rounding (SR) [39, 40]; the latter is named stochastic due to the requirement for random numbers. Given a real number x, an output fixed-point format to round the value to $< s, i, p >$ with $\varepsilon = 2^{-p}$ and defining $\lfloor x \rfloor$ as the truncation operation (cancelling a number of bottom bits and leaving p fractional bits) which returns a number in $< s, i, p >$ format less than or equal to x, RN is defined as

$$\mathrm{RN}(x, < s, i, p >) = \begin{cases} \lfloor x \rfloor & \text{if } \lfloor x \rfloor \leq x < \lfloor x \rfloor + \frac{\varepsilon}{2}, \\ \lfloor x \rfloor + \varepsilon & \text{if } \lfloor x \rfloor + \frac{\varepsilon}{2} \leq x < \lfloor x \rfloor + \varepsilon. \end{cases} \tag{2.5}$$

Note that for numerical error reduction in Chapter 3, the choice in this work was to implement round up on the tie $x = \lfloor x \rfloor + \frac{\varepsilon}{2}$; this was done because it results in a simple rounding routine that requires checking only the Most Significant Bit (MSB) of the truncated part to make a decision to round up or down. Other tie breaking rules such as round to even can sometimes yield better results, but a cheaper tie-breaking rule in this work is preferred.

Another rounding routine used in this thesis is stochastic rounding. It is useful to reduce the error in some applications, as is shown in Chapter 4, since it takes into account all of the round-off bits instead of only a single bit as in RN and rounds probabilistically in such a way that over many roundings the expected error is zero (due to errors with different signs cancelling out when added). Instead of always rounding to the nearest number as in RN, the decision about which way to round is non-deterministic and the probability of rounding up is proportional to the value in the round-off bits (called a *residual* here). Given all the values as in RN and, additionally, given a random value $P \in [0, 1)$, drawn from a uniform random number generator, SR is implemented as (similarly to [39])

$$\mathrm{SR}(x, < s, i, p >) = \begin{cases} \lfloor x \rfloor & \text{if } P \geq \frac{x - \lfloor x \rfloor}{\varepsilon}, \\ \lfloor x \rfloor + \varepsilon & \text{if } P < \frac{x - \lfloor x \rfloor}{\varepsilon}. \end{cases} \tag{2.6}$$

For efficiency and reproducibility it is expected that P is generated with a Psuedo-random Number Generator (PRNG).

Lastly, the cases without rounding (bit truncation) will be denoted as Round Down

(RD). For floating-point numbers it is very similar, as it comprises a significand which is essentially a fixed-point value scaled by $2^{exponent}$. However, note that in 2's complement fixed-point representation bit truncation results in RD, while in floating-point arithmetic, which does not use 2's complement, bit truncation results in Round-towards-Zero (RZ).

2.1.5 Carry-save adders

In Chapter 5, algorithms for exponential and logarithm are implemented using carry-save number representation. In Figure 2.7A, a symbol for a carry-save adder is shown (also called 3-to-2 compressor, as labelled). Figure 2.7B demonstrates a carry-save adder which can add four binary vectors (which could be two carry-save numbers) and produce a carry-save number — this is called a 4-to-2 compressor. Figure 2.7C demonstrates a symbol for it and note that it has two bits each for carry in and carry out. A lot of textbooks do not consider carry in signal in carry-save adders, but it is sometimes useful to add a bit in, for example when we want to perform $x+y-z$, which requires inverting z in 2's complement.[1]

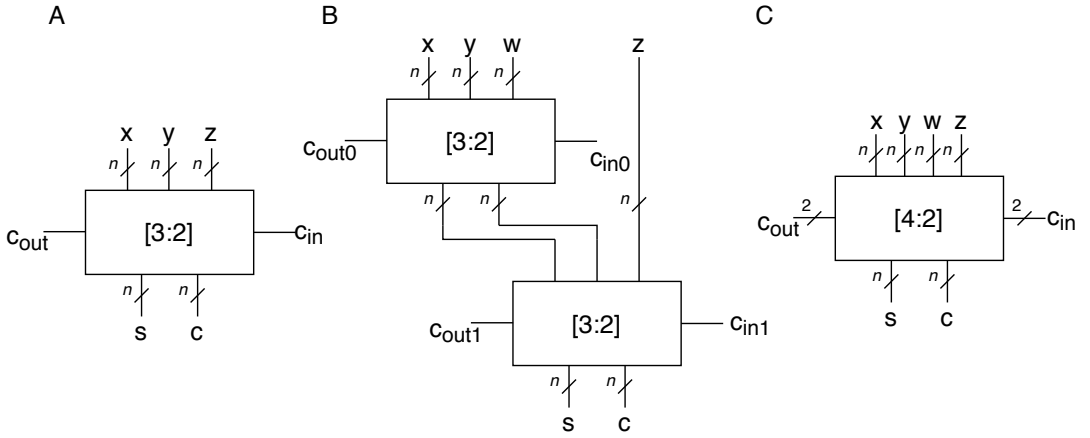

Figure 2.7: A: Bit-vector level carry-save adder. B: 4:2 carry-save adder built from two 3:2 carry-save adders. C: Bit-vector level 4:2 carry-save adder.

Some issues with the right shifts of the carry-save numbers were relatively recently discovered [41]. However, as discussed by the authors, the problems do not occur in most of the algorithms and hardware implementations in modern processors and this

[1] Even more useful is doing $x+y-(w+z)$ for subtracting a carry-save number from another carry-save number. In this case, because we have two binary vectors w and z which we want to sign invert in 2's complement, we need to bit-invert both vectors and add 2 to the total, which can be done by using the two carry inputs in the 4-to-2 compressor.

issue did not appear in any of the algorithms used in Chapter 5 even though right shifts of carry-save numbers are performed.

2.2 Analogue neuromorphic computers

Analogue neuromorphic processors implement neuron, synapse and learning models using physical components that can approximate their behaviour in integrated circuits (ICs) [3]. Three analogue processors are discussed here: *BrainScaleS* (Heidelberg), *BrainScaleS 2* (Heidelberg) and *DYNAPs* (ETH Zurich).

The BrainScaleS is based on the direct translation of neuron and synapse model equations to electronic circuits that have measurable quantities representing different variables. Time is scaled in BrainScaleS by a factor of 10^4 of real-time, which means that equations are solved 10 thousand times faster than in biology and this translates into the firing times and rates of the neuron. The neuron model used in BrainScaleS is AdEx [42, 43] which is implemented with 23 analogue parameters for configuration. The neuron model is solved in *continuous time* rather than in the discrete time steps like in a digital neuromorphic chips. Each neuron can receive up to 16 000 incoming synapses [44]. The chip also contains a substantial digital component, namely an infrastructure for propagating neuron action potentials, and control of weight updates for implementing STDP [45]. Chips are interconnected using *wafer-scale integration*, which means that the chips manufactured on silicon wafers are not used separately but left on the wafer with some interconnection between the chips added by post-processing the wafer.

The latest version of the BrainScaleS has a general purpose digital processor embedded into the chip, for implementing programmable STDP rules [12, 46]. A prototype BrainScaleS2 chip was recently used by Wunderlich et al. [47] for implementing a Pong playing agent in spiking neural networks. It is running with a speedup of 10^3 relative to biological time and contains an analogue part for neuron and synapse models and a digital processor running at 98 MHz interfacing with the analogue part for programmable plasticity rules. While the digital part is running relatively slowly, the speed-up of the analogue part which implements neurons and synapses allowed Wunderlich et al. [47] to draw the following speed and power figures compared with the simulation running on a general purpose Intel CPU: 50 000 network learning iterations took 25 s on BrainScaleS2 versus 40 min on a conventional Intel processor and used at least three orders of magnitude less energy. While the network that was simulated

is quite small (32 neurons and 1024 synapses), these speed and energy numbers show what will be possible in the future generations of analogue hardware; the authors mention plans to build a new prototype chip with 512 neurons and 130k synapses which might be used for more complex networks. At the moment, the main challenge of these platforms is that every synapse and neuron has to be implemented physically and therefore the size of networks is limited to the size of system built. Furthermore, since neuron models and synapses are fixed in hardware, the simulations are less customizable than the simulations on a general purpose CPU or a digital neuromorphic platform.

Another notable neuromorphic processor that is based on analogue circuits is *DYNAPs* [48].

2.3 Digital neuromorphic computers

Digital neuromorphic chips such as SpiNNaker implement classical von Neumann processors, containing adders, multipliers, program counters and other usual components, communicating with instruction and data memory and in some, accelerators to speed up the simulations of neural networks. Usually the processors are general purpose, but have an efficient interconnect (using routers and network-on-chip, NoC) and software optimized for spiking neural network (event-driven) algorithms. Therefore, the main components that can be called *neuromorphic* are interconnect fabric to transport spike packets and software, with the processor hardware being quite a standard multi-core architecture. Other digital neuromorphic platforms, such as *TrueNorth* or *Intel Loihi* have digital circuits to do specific tasks with minimum programmability due to the lack of the general purpose CPU. In this section provided are brief overviews of Intel Loihi, True North, SpiNNaker, and SpiNNaker2 digital neuromorphic platforms.

2.3.1 Intel Loihi

The Intel Loihi digital neuromorphic chip contains 130000 neurons, 130M synapses simulated in real-time across 128 digital neuromorphic cores and three standard x86 cores [13]. The inclusion of x86 cores seems to be for routing packets and for modifying the synaptic parameters. In addition to a digital on-chip Leaky Integrate and Fire (LIF) neuron, Loihi includes: pre- and post-synaptic spike traces with configurable time constants for learning rules; two additional state variables per synapse apart from

weight and delay, which can be used for reinforcement learning for example; reward traces that correspond to special reward spikes that can be sent to sets of synapses; pseudo-random number generators; configurable delays, and multi-compartment neuron support (dendritic trees). Synaptic weights are 9 bits and traces 7 bits, which means that Loihi is a highly reduced-precision computer and there is some indication in Davies et al. [13] that it is a custom floating-point format. Davies et al. [13] also mention that stochastic rounding (more in Chapter 4) is used when computing spiking history traces — likely to avoid large errors due to reduced 7-bit precision. Furthermore, instead of globally distributed periodic timer ticks to update all the neurons' states in a network, Loihi uses a handshaking mechanism between the cores to decide when it is safe to proceed to the next time step.

2.3.2 TrueNorth

IBM's TrueNorth is another digital chip for spiking neural networks [49, 50]. It contains 4096 cores with each core capable of simulating 256 integrate-and-fire neurons with 256 synapses each and requires only 63 mW of power. Differently from SpiNNaker and Intel Loihi (which is not based on a general purpose CPU but still has customizable plasticity rules), TrueNorth implements digital circuits optimized to replicate specific neuron and synapse behaviours.

2.3.3 SpiNNaker

SpiNNaker can be most clearly explained by following the bottom-up hierarchy: SpiNNaker cores and chips, large SpiNNaker systems (with interconnected chips) and SpiNNaker software. The SpiNNaker chip is presented next.

2.3.3.1 SpiNNaker chip

The SpiNNaker chip (Figure 2.8) is the main building component of a million-core computing cluster for simulating large-scale SNNs. The SpiNNaker chip comprises 18 ARM968 processors as well as a 128 MB block of shared off-chip memory. Typically each ARM968 in the SpiNNaker chip will be allocated up to 255 neurons so a single chip can model approximately 4000, where the exact number depends on the complexity of the simulation models. Each core has 64 kB of data storage memory called Data Tightly-Coupled Memory (DTCM) and also 32 kB of instruction memory, called Instruction Tightly-Coupled Memory (ITCM). The executable program is

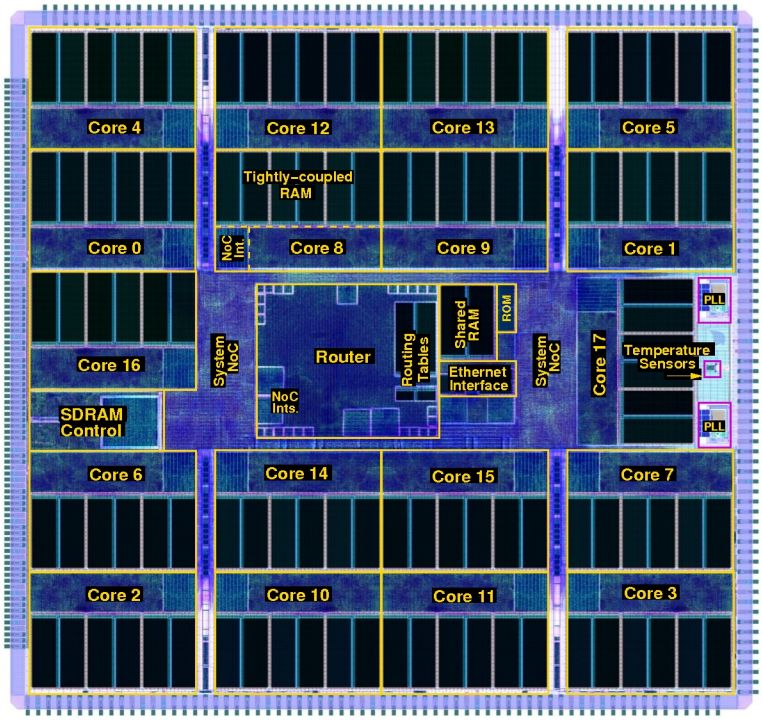

Figure 2.8: Image of the SpiNNaker chip [6].

constructed according to the high level description provided by the user, which takes into account things such as neuron models and learning rule types and linked together with the SpiNNaker low-level libraries. The compiled code is then downloaded to the ITCM and any data structures that are used while an application is running are stored in the DTCM. A DMA controller is used to send data to the off-chip memory or copy data into the DTCM from the off-chip memory. It functions in parallel with the processor and is the fastest method to copy synaptic data into the DTCM when spikes are received. Each processor runs at a nominal speed of 200MHz and works in an event-driven fashion, with periodic timer events causing neuron updates on the chip and spike events to process synaptic inputs.

In terms of SNN simulation performance, it is estimated that 5M connections/s per core (where connection is one synaptic signal) can be processed in real time [7] if there is no plasticity and 1M connections/s per core if there is plasticity. However, these numbers are only rough estimates and the real performance is always dependent on what kind of activities the model has to perform on each spike input (for example, how much computation is required for plasticity on each spike arrival) and what is the rate of inputs. Finally, it is worth reiterating that ARM968 is an integer processor and does not contain floating-point hardware support or dividers or similar complicated arithmetic

circuits. Due to this, all of the neuron and synapse model numerical algorithms are run in fixed-point arithmetic using the integer arithmetic hardware components.

2.3.3.2 Large systems

SpiNNaker (now referring to the machine) is a large collection of low power computational nodes that are able to send and receive signals (similar to neuronal spikes in the brain) to/from any other node in the network, and can be programmed to run code upon receiving each spike to model the approximate behaviour of biological neurons and the synapses that connect them. Two main parts are relevant at this level: the network-on-chip (NoC) and the router. Each core assembles SpiNNaker packets of which there are multiple types. It is out of scope to discuss the SpiNNaker packet types, but the most common one is a multicast packet. The main information in such a packet is a 32-bit key which is first used in the router of the chip in which the sender resides, and then passed down to the network either inside the chip (to one or many of the 18 cores which might be receivers) and to one or many of the 6 chip links to neighbouring chips. Any chips where the packet arrives must have a routing table stored inside the router and if a packet arrives the steps of routing are repeated in those chips by using the key again [51].

Using this mechanism, a large system of 1 million cores was recently built and all the cores successfully turned on. The machine can now be accessed free, through various methods provided by the Human Brain Project where people can submit spiking neural network scripts written in PyNN (see below) for simulation on the machine.

2.3.3.3 SpiNNaker software

The usual routine of simulating something on SpiNNaker is as follows. The highest level description of a spiking neural network is made using a Python-based language PyNN [52]. The neural network description is then interpreted by Python and PyNN and passed into the SpiNNaker toolchain for distributing it across some number of ARM cores. Multiple precompiled binaries for combinations of different neuron, synapse and plasticity models should be in place when this step is happening, since the toolchain interprets the network description and selects the appropriate binaries to copy to each core.

Neural simulations on SpiNNaker undertake two main activities: *neural processing* and *synapse processing*. Neural processing involves calculating voltage potentials

that determine when neurons fire, given any inputs that have accumulated in the input buffers. Synapse processing refers to simulating each signal travelling through synapses after neurons fire. Naturally, as there are around 100 billion neurons in the brain and approximately $10\,000\times$ more synapses, synapse processing occupies the largest amount of SpiNNaker processing time and therefore it is the part of simulation which should be optimised the most.

The software can be most conveniently described in this order: PyNN language, neuron models, synapses, plasticity, synapse processing routine and finally neural processing routine.

PyNN neural network description language PyNN is a simulator-independent SNN description language [52]. The main purpose is to be able to have a single notation for constructing SNNs that can be run on multiple different simulators (traditional software on High-Performance Computing (HPC) platforms or neuromorphic simulators). SpiNNaker supports an interface between SpiNNaker and PyNN that enables users to run PyNN scripts without requiring any change when moving from another simulator.

Neuron models Two neuron models are supported on SpiNNaker: leaky integrate-and-fire neuron (LIF) and Izhikevich neuron (variations of these are also supported, for example current or conductance based versions). Izhikevich neuron is discussed in Section 3.1. The code for advancing the differential equation of each in discrete time is provided in the SpiNNaker toolchain [53] and the time step value can be configured by the simulation settings.

Synapses Another element of the SNN simulation is synaptic data, which includes the parameters of each synapse: *weight*, *delay* and *type*. It also includes the post-synaptic neuron ID of the connected synapse. Synaptic data occupies a significant amount of space, therefore is stored in the off-chip memory and copied in chunks into local memory only when the post-synaptic neuron core receives a spike.

Plasticity Synaptic plasticity describes the change of synapses according to various factors, such as spike timing. Hence, when simulating networks which include plasticity, synapses change their weights and therefore require synaptic data to be updated in the off-chip memory. Plasticity is a critical part of the simulation and is usually a major limiting factor when enabled. This is because for each spike that arrives, we not

Algorithm 1 Pseudo algorithm of STDP (reproduced from Knight et al. [9])

```
function PROCESSROW(t)
    for all j in postSynapticNeurons do
        history ← getHistoryEntries(j, t_old, t)

        for all (t_j, s_j) in history do
            w_ij ← applyPostSpike(w_ij, t_j, t_old, s_i)

        (t_j, y_j) ← getLastHistoryEntry(t)
        w_ij ← applyPreSpike(w_ij, t, t_j, s_j)
        addWeightToRingBuffer(w_ij, j)

    s_i ← addPreSpike(s_i, t, t_old)
    t_old ← t
```

only have to go through the synaptic data and add the weights into the neuron buffers, but also go though the spiking history of each post-synaptic neuron, apply correlation of each pre- and post-synaptic spike and evaluate the new weight — a three way loop going through each input spike, each post-synaptic neuron and each spike time of the neuron.

Various attempts were made to improve the performance of the plasticity algorithm on SpiNNaker. Diehl and Cook [54] presented an extensive analysis of the problems and an alternative STDP simulation software framework for SpiNNaker which improved performance. Furthermore, more recently Knight and Furber [55] presented a complete re-configuration of how a PyNN neural description is mapped onto SpiNNaker with separate cores allocated for synapses and neurons which allowed large-scale simulations inspired by the structure and plasticity in the neocortex to be demonstrated by Knight [56]. However, all software optimisations meet hardware limits (mainly of an ARM968) quite quickly, so optimising hardware specifically for these learning algorithms in SpiNNaker2 will allow to improve current models further.

STDP is simulated on SpiNNaker using a *trace-based* approach [57, 58] where each synapse records pre- and post-synaptic neural activity into local trace variables (s_i and s_j respectively) with the following dynamics:

$$\frac{ds}{dt} = -\frac{s}{\tau} + \sum_{t^f} \delta(t - t^f), \tag{2.7}$$

where spikes at times t^f, described by Dirac delta function $\delta(t - t^f)$, increase the value

of the trace which decays exponentially with a time constant τ. The time constants of these traces are typically set to match the shape of the desired STDP function.

The processing of a single synaptic row is implemented using the PROCESSROW() function shown in Algorithm 1 which is called when a pre-synaptic spike arrives and the corresponding synaptic matrix row has been transferred into local memory. This algorithm applies all of the STDP updates that have occurred since the last pre-synaptic spike was received. For more complex rules this double loop can also be extended, for example, to go through some other spiking history apart from post-synaptic neuron, such as dopaminergic signal arrival times (this is discussed in Section 3.3).

Synapse processing When a core receives a multicast packet it sets up a DMA to copy some synaptic data into the local memory. When the DMA copying completes, a loop starts going through each plastic synapse first, calling the plasticity algorithm discussed above. When that is finished, non-plastic synapse update happens, going through each synapse in the synaptic row, indexing the correct ring buffer[2] entry using the synapse *delay*, synapse *type* and neuron ID, and adding the weight into it.

Neural processing routine Neural processing is done on periodic timer interrupts and involves taking the excitatory and inhibitory input from synapses, decaying and adding it, in the form of a current, to the corresponding neurons. It is then determined whether the neuron should spike or not. If it spikes, it will generate a spike packet which will propagate across the system nodes the control of the routers and eventually raise an interrupt on all of the destination cores, which will trigger synapse processing on those cores.

These are main parts of SpiNNaker (and potentially the upcoming SpiNNaker2) software. For more details refer to [5, 51, 53].

2.3.4 SpiNNaker2

A second generation chip named SpiNNaker2 is currently in development phase. The first prototype chips, codenamed *Santos*, have already been manufactured and used for various experiments [59, 60, 61]. A second prototype chip, codenamed *JIB1*, is being

[2]The ring buffer is the data structure that is used to model spike arrival with delays on the post-synaptic neuron processor. Each ARM968 on SpiNNaker has a collection of ring buffers which store the synaptic contributions for specific times in the future. Ring buffers are read periodically, each time taking an element indexed by the current time value of the running simulation. This is required because biological delays are longer than signal travel time through SpiNNaker's interconnect fabric.

tested at the time of writing and the design of the third prototype *JIB2* is finished. After this, the final SpiNNaker2 chip will be finished.

The main part of the SpiNNaker2 chip is more than a hundred ARM Cortex-M4F PEs. The ARM M4F is an upgrade from the older ARM968 model as it contains a 2-12 cycle integer divider, a binary32 FPU, and a more advanced than ARM968 Digital Signal Processing (DSP) instruction set [62]. Each PE has some local memory which will be split between code and data as in the current version of SpiNNaker. An off-chip memory is shared among all the cores as well. Each PE is equipped with a DMA controller to copy data to and from the off-chip memory to the PE and each chip has a router used to send spike packets from any core to any other core in the system. SpiNNaker2 will also contain functionality for dynamic power management as described by Höppner et al. [61]. Power management is done by software control to switch between power levels that are determined by voltage and clock frequency, and the change between two power level domains only takes approximately 100 ns. Processors can therefore be programmed to do application-specific power management, for example based on the spiking activity, without much overhead.

The SpiNNaker2 chip will also have memory sharing between subsets of PEs which will give it more customization possibilities for cases where a large fast memory space is required [63]. Lastly, various accelerators are being added to each PE: matrix multiplication, exponential and logarithm functions (this thesis), rounding support with customizable bit position (this thesis) and random and pseudo-random number generators; the accelerators will help increase the speed of certain algorithms and decrease energy consumption compared to simulations purely done in ARM M4F software. Initial results on the *Santos* prototype chip reported by Yan et al. [21] in implementing a reward-modulated plasticity rule show a more than 2× decrease in clock cycles required for each plasticity update, more than 2× increase in the number of synapses that can be simulated per core, and 41 % energy reduction due to use of accelerators in that particular plasticity rule compared to the same simulation done without the accelerators. Similar speed and energy improvements were reported by Liu et al. [64] when running *deep learning* algorithms on *Santos*.

2.4 Software neural simulators

The most accessible options are various SNN simulators available that can run on standard processors, GPUs or FPGAs.

NEST [65] is a software package for simulating large-scale SNNs on a conventional cluster of processors. It supports various neuron models, including such complex neuron models as Hodgkin-Huxley [66]. Furthermore, two-factor STDP is supported as described by Morrison et al. [67] and there were some experiments shown with three-factor, neuromodulated STDP by Potjans et al. [68].

GeNN [69] is a software framework which allows users to generate spiking neural network simulation code optimized for running on GPUs. Using this framework, Yavuz et al. [69] reported a 200-fold speed increase compared to a conventional CPU running a network with 1M Hodgkin-Huxley neurons. Furthermore, speed-up and energy reduction in simulating a cortical microcircuit model with approximately 80k neurons was explored by Knight and Nowotny [70] and it was demonstrated that in this particular network, when simulated on a single NVIDIA Tesla V100, it outperformed current NEST and neuromorphic simulators both in speed and energy, despite simulating it in floating-point arithmetic. Further improvements could be gained, as noted by the authors, by switching to GPUs with low-precision fixed-point arithmetic and using mixed-precision to alleviate some rounding errors.

Some experiments simulating spiking neural networks on FPGAs have also been performed. Pani et al. [71] demonstrated a simulation of 1440 Izhikevich neuron models on a Xilinx Virtex 6 FPGA device. This has an advantage against other solutions in that neuron models and possibly plasticity rules can be configured by the users in hardware which allows both better speed than software approaches, and better flexibility than analogue approaches.

2.5 Conclusion

Floating-point has dominated computer arithmetic for multiple decades and the properties and pitfalls of this data type are now very well known. Now reduced precision fixed-point arithmetic is making a come back, with the need to make machine learning hardware more efficient and the observation that most of the algorithms there do well with small numbers of bits interpreted as fixed-point numbers. Latest high performance computers are even adding 8 and 16-bit hardware units for performing integer arithmetic (one example is *Fukagu* supercomputer). Various novel numerical formats are also explored, such as posit [72, 73] or bfloat16 with hardware support planned by Intel [36]. In the rest of this thesis both fixed- and floating-point formats for various software and hardware parts of SpiNNaker and SpiNNaker2 will be considered.

Various approaches are available for simulating spiking neural networks: analogue, digital, software (CPUs or GPUs) and FPGAs. Depending on what is required from programmability, complexity of neuron and plasticity models, size of networks, energy and real-time run time constraints, a choice has to be made as to what platform modellers should use for their research. This chapter has shown a landscape of different neural simulation platforms, with a special emphasis given to SpiNNaker and SpiNNaker2 that are subject of this thesis.

The next chapter deals with some numerical accuracy issues on current version of SpiNNaker where various methods are demonstrated for improvements at arithmetic level.

Chapter 3

Neuron and Plasticity Models in Fixed-Point Arithmetic on SpiNNaker

Improvements at software level on the current version of the SpiNNaker chip are explored in this chapter. First, various numerical issues, previously reported in the implementation of the Izhikevich neuron model in fixed-point arithmetic, are replicated using the default SpiNNaker software. Then, three different ways to improve the numerical accuracy are demonstrated: rounding of constants, mixed-format multipliers, and rounding of multiplication results (the first two methods do not affect the speed of the neuron model). In each step, a particular neuron simulation experiment used for generating the original results is rerun and any improvements to the accuracy are analysed. Furthermore, in this section accuracy improvements to the exponential decay on the current generation SpiNNaker chip are shown, which can also be applied to the hardware accelerator of the first SpiNNaker2 prototype, because the method requires only software level modifications. Finally, this chapter includes a new model of three-factor STDP with performance and accuracy discussion.

Some sections in this chapter are reproduced from the material that was published in the Frontiers in Neuroscience journal [11] and the proceedings of the 27th IEEE symposium on computer arithmetic [74].

3.1 Numerical accuracy of the Izhikevich neuron model

One of the neuron models widely used on SpiNNaker is the Izhikevich neuron model [75]. This neuron model can be configured to exhibit a diverse family of spiking patterns. It is widely considered a good trade-off between biological plausibility and

computational efficiency.

The neuron model is described by the following two-dimensional system of ODEs:

$$\begin{aligned} \frac{dV}{dt} &= 0.04V^2 + 5V + 140 - U + I(t), \\ \frac{dU}{dt} &= a(bV - U), \end{aligned} \tag{3.1}$$

where V (initially set $V = V_{init}$) represents a neuron's membrane potential, U (initially set $U = U_{init}$) represents a membrane recovery variable providing negative feedback to V [75] and I includes synaptic or injected current. If $V \geq 30\,\text{mV}$, it emits a spike at which point $V = c$ and $U = U + d$. Parameters a, b, c, and d can be configured to achieve specific spiking behaviours under different stimulation conditions [75].

The ODE system in (3.1) does not have a known analytical solution and therefore requires a numerical integration algorithm. On SpiNNaker, this model is solved using the second order Runge-Kutta (RK) Midpoint ODE solver and, due to the lack of the FPU, this is computed in fixed-point arithmetic [53], using only s16.15 fixed-point data type for all the variables, constants and intermediate results.

Various sources of error are introduced in this solution, one of which is rounding error that is of interest to explore. The first impression is that rounding error must be much lower than the error of an ODE solver, however, the results in this section show that this is not the case here. First of all, there are issues in the GCC implementation of fixed-point arithmetic [74] such as lack of rounding of decimal constants. Also, various other techniques such as mixed-precision arithmetic for intermediate results are also shown to reduce the overall rounding error.

3.1.1 Previous work

There are three main published studies exploring the accuracy of the Izhikevich neuron model with fixed-point arithmetic on SpiNNaker. Jin et al. [76] explored this using a basic Euler method for solving the Izhikevich ODE and evaluated the accuracy using spike counts without investigating errors in spike timing. Later, a set of ODE solvers in the context of SpiNNaker were investigated by Hopkins and Furber [14] and, by evaluating speed and spike timing errors, RK2 Midpoint was chosen as a good tradeoff between accuracy and computational cost for a default SpiNNaker software implementation [53].

In the solution by Hopkins and Furber [14], the ODE solver which updates the

neuron model for a given time step $\Delta t = h$ and input current I_{t+h} is as follows:

$$\begin{aligned}
\theta &= 140 + I_{t+h} - U_t, \\
\alpha &= \theta + (5 + 0.04V_t)V_t, \\
\eta &= \frac{h\alpha}{2} + V_t, \\
\beta &= \frac{h\alpha(bV_t - U_t)}{2},
\end{aligned} \tag{3.2}$$

$$\begin{aligned}
V(t+h) &= V_t + h(\theta - \beta + (5 + 0.04\eta)\eta), \\
U(t+h) &= U_t + ah(-U_t - \beta + b\eta).
\end{aligned} \tag{3.3}$$

Most recent work [15] explores the RK2 Midpoint solver used in Hopkins and Furber [14], and the Euler solver with multiple iterations in a single time step update to reduce the spike lags (but without considering the computational overhead). The authors succeeded in reducing the spike lags from previous studies and provided some graphical comparison of the 10 first spikes. It is shown that when the neuron is subject to a constant DC current, the 10th spike generates some spike lag (approximately 4 ms, judging from the plots in [15]) compared to the reference with a very accurate ODE solver and binary64 floating-point arithmetic. Therefore, longer simulations would still produce substantial spike lag where some simulators with binary64 floating-point arithmetic would yield different spike timings from SpiNNaker; this results in two simulations on different platform with the same initial conditions and parameters that do not produce the same results. This kind of inability to reproduce simulation results between computers and environments is generally not desired and should be evaluated, and the level of expected differences clearly stated for each system. Therefore, it is important to address further why this large reported spike timing error occurs in fixed-point arithmetic and how it can be reduced with minimal decrease in efficiency (which is the main goal of using fixed-point arithmetic in the first place).

All of the experiments run in the next subsection are done with a GCC compiler optimization `-O2` or `-Ofast` turned on. The flag `-O2` causes the compiler to undertake the second level optimisation (out of three main levels in an increasing complexity) [77]. The flag `-Ofast` can result in faster code than `-O2`, but violates compliance of standards which can affect things like rounding. Therefore, for measuring the efficiency of code `-Ofast` is used while for checking accuracy and rounding properties `-O2` is used. Note that the SpiNNaker toolchain, at the time of writing, in various different parts

of compilation uses either `-Os` for binary size optimization or `-Ofast` for speed, and these different flags may provide minor differences (compared to the improvements presented here) in spike timings due to reordering of arithmetic operations in the ODE solver.

3.1.2 Accuracy benchmark

For the purposes of evaluating various improvements to the fixed-point arithmetic the constant DC current test from Hopkins and Furber [14] was replicated. The test is done by evaluating a Regular Spiking (RS) Izhikevich neuron (with the parameters: $a = 0.02$, $b = 0.2$, $c = -65\,\text{mV}$, $d = 8$, $V_{init} = -75\,\text{mV}$, and $U_{init} = 0$). This neuron is stimulated at 60 ms with a step current of 4.775 nA which is left until the simulation is stopped. This constant current should cause an approximately 10 Hz spike rate with 100 ms intervals between spikes.

In Hopkins and Furber [14] the reference for comparison is taken to be a Mathematica implementation that provides a very accurate solution (very accurate ODE solver, adaptive time step, arbitrary precision arithmetic). In the tests of this thesis, for simplicity, the reference for comparison was chosen to be an RK2 Midpoint ODE solver, the same that is used in SpiNNaker, in binary64 arithmetic. This is easier to run as it can be done on SpiNNaker using software binary64 arithmetic and it removes the *algorithmic error* from the comparison as the same ODE solver algorithm is used in the reference. Therefore, theoretically, a good implementation of RK2 Midpoint in fixed-point arithmetic could produce the same spike timings as the RK2 Midpoint in binary64 arithmetic, whereas choosing a reference to be a different, better, ODE solver, will never manage to *match exactly* the problem under testing with the reference. This kind of fixed error due to different ODE solvers can be neglected for our purposes of evaluating different types of arithmetic. This is discussed further in Section 4.4.1.

Hopkins and Furber [14] provided graphical plots of the spike lag/lead of various ODE solvers and arithmetics when the neuron is stimulated with the above conditions until it has fired 19 spikes. Multiple tests with 1 ms and 0.1 ms time steps were performed. It was shown that the current default SpiNNaker ODE solver with fixed-point arithmetic and a 1 ms time step produces approximately 82 ms of spike lag in the 19th spike, compared to the very accurate Mathematica reference. This means that the neuron produced the 19th spike 82 ms later than the reference. Whereas, with a 0.1 ms it produces approximately 57 ms spike lag in the 19th spike. It is evident that time step size is important, but the spike lag at 0.1 ms time step still seems very high.

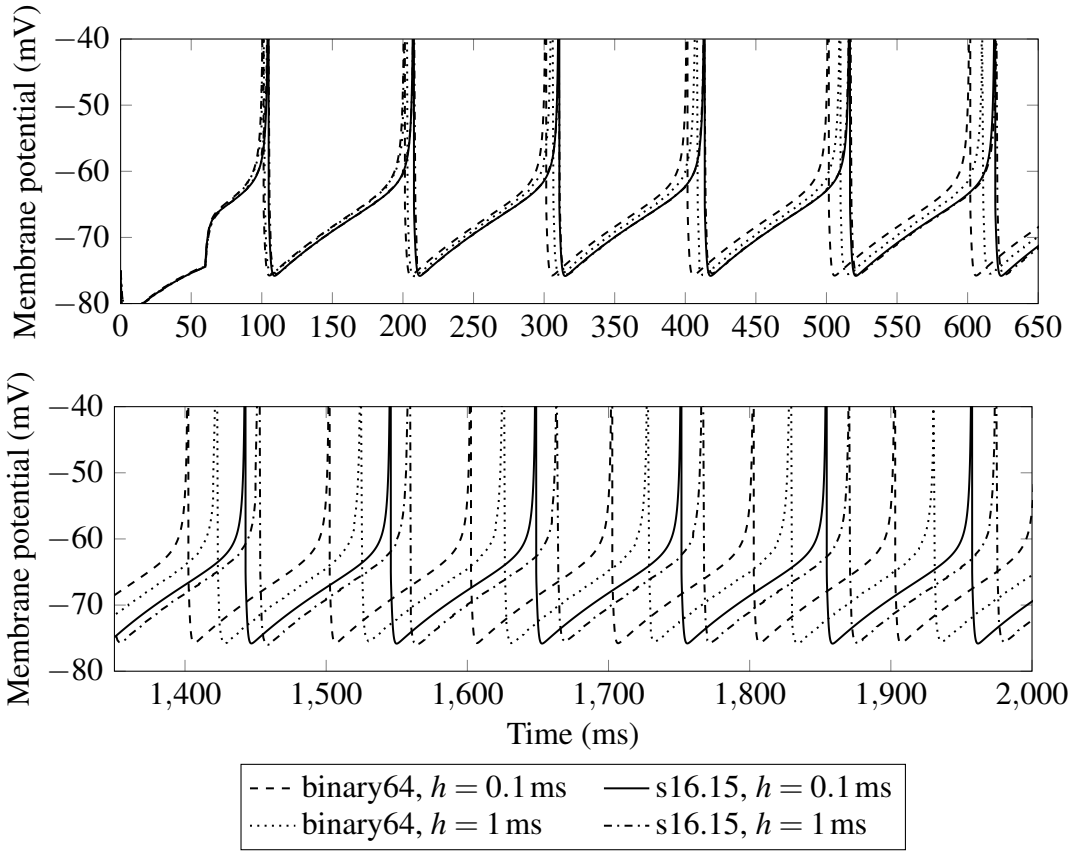

Figure 3.1: Top: 6 first spikes, bottom: 6 last spikes, from the 19 spikes of the neuron being stimulated with a constant DC current input. The RK2 Midpoint ODE solver (the default in SpiNNaker) is used as in Hopkins and Furber [14]. Binary64 traces act as references for each fixed-point test case. The visible drift of each spike from the reference is spike lag/lead which is used as the main measurement of error.

The test with the RK2 Midpoint ODE solver that Hopkins and Furber [14] used is replicated here using the PyNN script included in Appendix A. Figure 3.1 shows the membrane potential traces and Figure 3.2 shows the spike lags of the first 19 spikes, with 0.1 ms and 1 ms time steps and the default fixed-point arithmetic using the s16.15 data type. Single-precision (binary32) floating-point spike lags are also included for comparison, mainly because it is generally considered to be a better arithmetic than fixed-point, due to the wide range of representable values and is available on SpiNNaker2 in hardware.

The reference of each run is a corresponding binary64 set-up, with the same ODE solver and the same fixed time step. The initial conditions of the ODE, such as constants that are used on each step and the initial values of various variables, are assigned

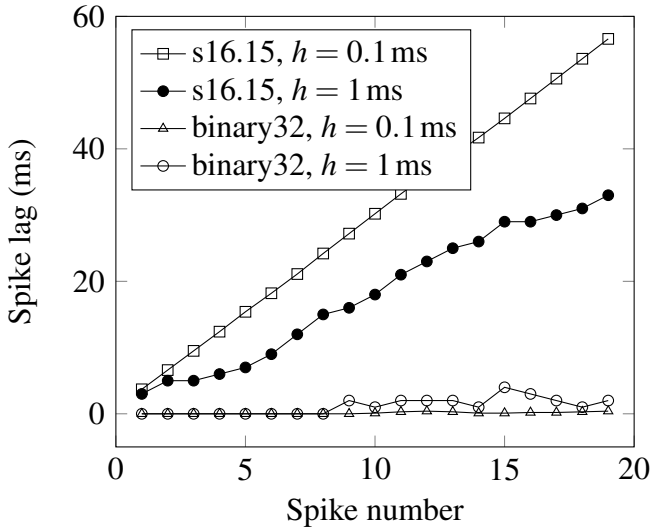

Figure 3.2: Spike lag of the Izhikevich neuron model with different arithmetics and time steps. Using the RK2 midpoint ODE solver, results reproduced as in Hopkins and Furber [14], but with a different (binary64) reference for each time step, instead of Mathematica.

the best possible representation in a given arithmetic (except the DC current value of 4.775 nA which is fixed at the nearest s16.15 value of 4.774993896484375 for all tests), which means that the accuracy of constants is included in the *arithmetic error*. It can sometimes be useful to remove this source of error in the constants as well so that all ODE solvers start at the same state and also on each step use the same values, and this can be done by using the constants of the lowest precision number representation from the set of arithmetics considered for comparison (usually this is possible, for example binary64 arithmetic can represent all of the s16.15 values).

The 19th spike in Figure 3.2 has a spike lag of 56.6 ms and 33 ms with 0.1 ms and 1 ms time steps respectively. The reference has been changed from Mathematica to the same algorithm (RK2 Midpoint) with the same time step (0.1 or 1 ms) run using binary64 floating-point arithmetic. This removes the *algorithmic error* from the comparison (caused by a choice of the ODE algorithm and the time step size), therefore spike lags are lower than reported by Hopkins and Furber [14] who included ODE solver error in the comparison due to using Mathematica. The case with the time step of 1 ms shows less lag because the reference has more lag than with the time step of 0.1 ms. This might lead one to think that accuracy is better at a 1 ms time step, but after studying the traces in Figure 3.1 it becomes clear that it is worse in absolute terms.

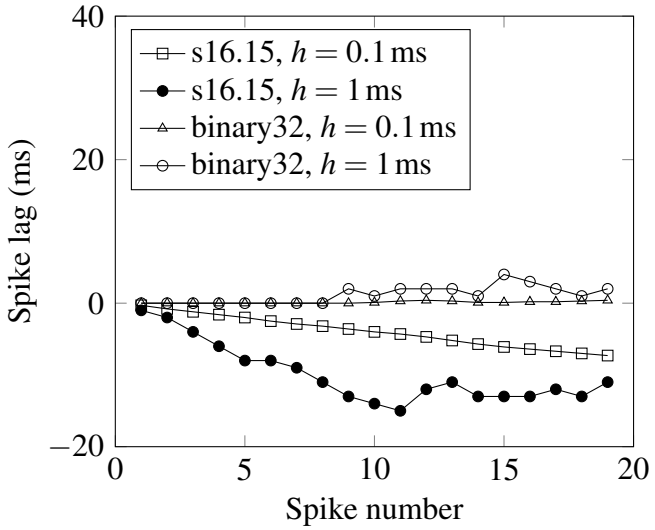

Figure 3.3: Spike lag of the Izhikevich neuron model with different arithmetics and time steps (RK2 midpoint ODE solver [14]) after specifying the constant 0.04 as the closest s16.15 value: 0.040008544921875. Error calculated by comparing to binary64 version of the solver.

3.1.3 Correct rounding of constants

Real numbers, such as the constants used in the ODE of the Izhikevich neuron model explored here, usually cannot be represented exactly in a digital computer. The most obvious path is to write down a decimal value in the program and leave it to the compiler to round it to the nearest value of the numerical format that is used. However, as part of this work it was discovered that the GCC implementation of the fixed-point arithmetic does not support rounding in decimal to fixed-point conversion and therefore the specification of constants suffers from truncation error which can be up to ε, a machine epsilon for a given target fixed-point format; for ODE solvers this error will in most cases accumulate as the constant is used inaccurately on each integration step. Additionally it was found that there is no rounding on most of the common arithmetic operations involving fixed-point numbers and between conversions from one numerical format to the other, less precise fixed-point format. The pragma `FX_FULL_PRECISION` defined in the fixed-point arithmetic standard [34] is not implemented in the GCC compiler version 6.3.1 that was used for this work and therefore there is no way to turn on correct rounding which contributed to the errors shown by Hopkins and Furber [14]. Experiments on GCC compiler version 9.2.1 confirm that this is also the case in that version.

There are multiple constants and parameters that are fixed throughout the simulation in (3.2) and (3.3). In (3.2) there are multiple constants: a, b, h, and 0.04 — where a and b are used to change the spiking behaviour, h is the timestep and 0.04 is a constant specific to this neuron model. a, b and h are parameters that in the SpiNNaker simulation workflow are set-up by the user at PyNN level and are rounded correctly to the s16.15 format (producing a nearest representable number) by Python interfaces on the host and copied into the SpiNNaker machine before running a simulation. However, the constant 0.04 was hard coded in the C source from the original work of Hopkins and Furber [14] using the notation 0.04k (where k tells the compiler that this is an s16.15 value) and therefore not rounded by the GCC compiler to the nearest representable value 0.040008544921875, and instead due to binary truncation (round-down) ending up as 0.03997802734375. Note that $\frac{0.04}{2^{-15}} = 1310.72$, which tells us that 0.04 lies 72 % along the way from 0.03997802734375 to 0.040008544921875 and therefore GCC choosing the former is incorrect.

The first step in this work was to specify constants in decimal exactly, correctly rounded to the nearest s16.15 (for example explicitly stating 0.040008544921875 as the constant instead of 0.04) which reduces the maximum error in the decimal to s16.15 conversion to $\frac{\varepsilon}{2} = \frac{2^{-15}}{2}$ instead of up to full epsilon if the decimal value is stated and left for the compiler to convert. As a result this has reduced the spike lag previously reported by Hopkins and Furber [14] significantly, leading to an understanding that ODE solvers can be extremely sensitive to how the constants are represented. This was also noticed by Trensch et al. [15] and the solution that the authors took was to add more bits in the fraction of the constants and add some scaling factors to return the final result in s16.15.

In Figure 3.3 the exact same experiments that were run in Section 3.1.2 are shown except that the two instances of the constant 0.04 in (3.2) and (3.3) are replaced by 0.040008544921875. The spike time errors of the 19th spike now are -7.3 ms and -11 ms (leading, relative to the reference, instead of lagging as before) for the time steps of 0.1 ms and 1 ms respectively. Compared to the original spike lags of 56.6 ms and 33 ms this is a major improvement without reducing the speed of the ODE solver. It is described that the macro `FX_FULL_PRECISION` in the fixed-point standard [34] is meant for performance and that it should be turned off in the platforms that prefer performance over accuracy. However, rounding of constants is done on compilation and therefore it would not impact run-time performance and should be implemented irrespective of whether the macro is turned on or not.

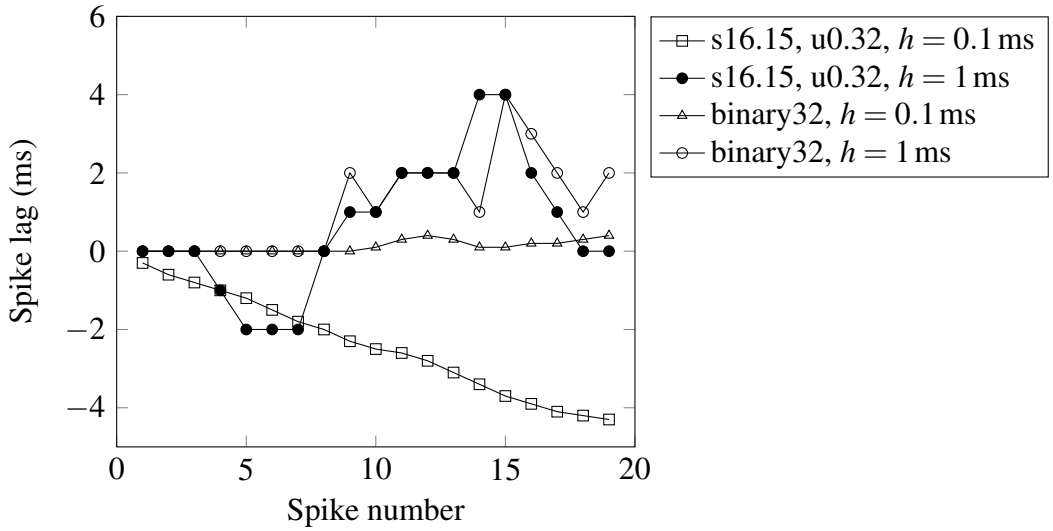

Figure 3.4: Spike lag of the Izhikevich neuron model with different arithmetics and time steps (RK2 midpoint ODE solver [14]) after specifying the constants as u0.32 and adding mixed-format multiplications. Error calculated by comparing to binary64 version of the solver.

3.1.4 Mixed-precision multipliers

Following from the numerical error improvements obtained in the above experiment, another step in this direction was to represent all constants smaller than 1 as u0.32 instead of s16.15 which results in the maximum error of $\frac{2^{-32}}{2}$. If the multiplicand can be represented more precisely, then the result of the multiplication becomes more accurate, even if the same precision number representation is used on the output. These include the parameters a, b as these are typically smaller than 1 [75], parameter h (configurable time step, requires knowing beforehand that it is going to be smaller than 1) and the constant 0.04. Trensch et al. [15] used s8.23 format for the constant 0.04, but there is no downside to going all the way to u0.32 format if the constants are below 1. Because these constants in u0.32 format will be multiplied by s16.15 values, different multiplication routines have to be called. However, as we are still working with the two 32-bit integers, the multipliers are not slower in any way from the s16.15 $\times$ s16.15 multipliers — the only difference is that the shifting right is now by 32 bits instead of 15 bits (which can be performed by ignoring the bottom result register, therefore in fact it does not even need a shift instruction). Furthermore, when a fractional value is multiplied by something, it is known that the result will be smaller than the multiplier and therefore no saturation check is needed. In order to support this, routines for

mixed-format multiplication operations were developed, where s16.15 variables can be multiplied by u0.32 variables returning an s16.15 result — this is used in the next subsection.

At this stage all of the mentioned constants and parameters were rounded to the nearest u0.32 values and declared them as such, letting GCC call its own implementation of mixed-format multiplications. The accuracy result is demonstrated in Figure 3.4. The accuracy of the 19th spike is further improved from the results shown in Sections 3.1.2 and 3.1.3 — spike lags of -4.3 ms and 0 ms were observed for 0.1 ms and 1 ms time steps respectively.

Theoretically this improvement should not add any overhead to the ODE solver as these new mixed-format multipliers are performed in the same number of instructions as the default s16.15 multiplications (or even less if saturation is removed and the shifting right by 32 is not required). However, in practice, GCC seems to not inline the multiplier function calls when u0.32 format is involved (even with the highest optimization level `-Ofast` flag enabled) resulting in a much slower ODE solver due to the requirement of branch and return from each multiplication.[1] To achieve this, custom mixed-format multipliers were used, of which some were already available as part of SpiNNaker software, and confirmed that switching to u0.32 constants did not impact the performance of the solver. Therefore, these new accuracy results in Figure 3.4 can be considered to be obtained without adding any performance overhead to the original RK2 Midpoint solver explored by Hopkins and Furber [14], with a downside that GCC, at least in the current version, cannot optimize mixed-format multipliers even with `-Ofast` enabled and a custom-made multiplication routine has to be used.

3.1.5 Rounding of multiplier results

As well as rounding on conversion between numerical formats, the GCC compiler does not perform rounding on multiplication results. This section reports accuracy improvements when all the multipliers in (3.2) and (3.3) are replaced by custom multipliers for which rounding can be enabled as discussed in Section 2.1.1. Changing from the GCC multiplications to the custom multiplications introduces the following effects, some of

[1] Multiple fixed-point arithmetic routines in GCC have some loss of precision and speed, for example s16.15 multiplication by u0.32 is performed as multiplication of two s32.31 values, or the multiplication of s16.15 by u0.16 is performed as multiplication of two s16.15 values [74]. This causes loss of precision on conversion (in the arguments, even before multiplication is performed) and the main reason is that GCC does not support mixed-format multipliers directly, as indicated by the list of internal compiler functions for performing fixed-point arithmetic operations [78].

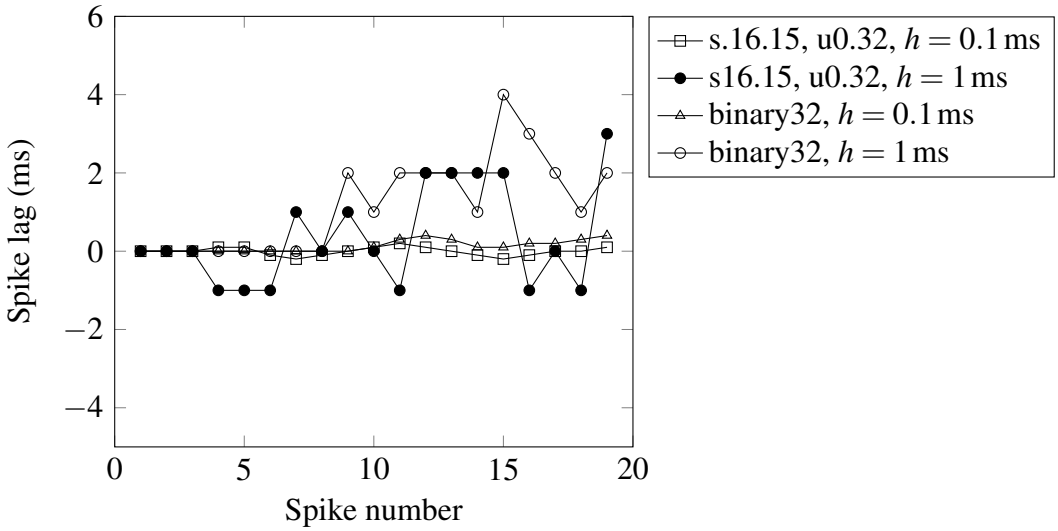

Figure 3.5: Spike lag of the Izhikevich neuron model with different arithmetics and time steps (RK2 midpoint ODE solver [14]) after adding rounding on the multiplications. Error calculated by comparing to binary64 version of the solver

which impact spike timing results slightly even without enabling rounding.

- Saturation is added on s16.15 $\times$ s16.15 multiplications. If the result overflows 16 integer bits, the highest positive or negative value in s16.15 is returned.
- No accuracy is lost in the argument conversion in s16.15 $\times$ u0.32 multiplications, which occurs in a corresponding GCC function as discussed above.
- All calls to multipliers can be inlined when compiled with `-O2`/`-Ofast` for performance.

Figure 3.5 demonstrates the spike lags of the benchmark with rounding added on the fixed-point multipliers, and floating-point versions for comparison. Rounding to nearest is performed by inspecting the MSB of the residual as described in Section 2.1.4 and adding 1ε to the result, if the bit is set. The position of the bits and the magnitude of ε depends on the output format of the multiplication result, which in some cases is s16.15 and in some cases s0.31, where possible.[2] Spike lags were 0.1 ms

[2]Where two u0.32 variables were involved in the multiplication, operations were rearranged to multiply these values first, in order to be able to store the results as precisely as possible for more accurate further operations. Such multiplications output s0.31 values and not u0.32 due to a requirement of negative sign to be added on the result in some places. Then, as soon as these outputs are multiplied by a s16.15 value, the answers start to be in this format too. Note that it is possible to rearrange the negative

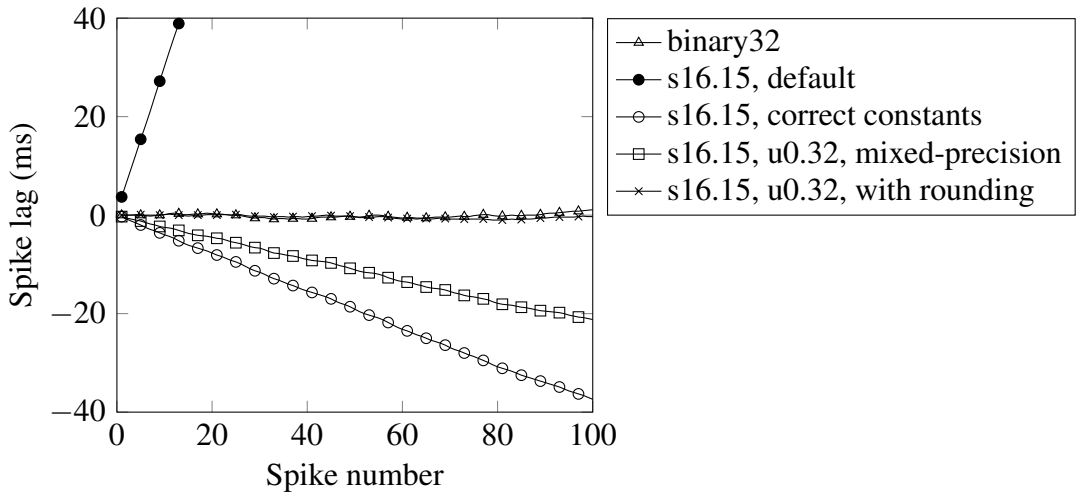

Figure 3.6: Spike lag of the Izhikevich neuron model with different arithmetics and time steps (RK2 midpoint ODE solver [14]) for a duration of 100 spikes with 0.1 ms time step. Error calculated by comparing to the solver in binary64. The default arithmetic, which falls off the diagram, had a difference of 293.8ms in the last spike.

and 3 ms for the time steps of 0.1 ms and 1 ms respectively. While the 1 ms time step case is not impacted by rounding, it can be seen that with a small cost of one extra addition for performing rounding, spike lags are further improved in the 0.1 ms time step test case.

Figures 3.6 and 3.7 collect all the results from the previous sections, while running the simulation for a longer duration until 100 spikes are produced. A clearer tendency can then be seen: fixed-point with correctly rounded constants represented in u0.32 and with rounding on multiplications produces spike lags equivalent to the same set-up run in floating-point arithmetic. Other approaches produce more lag and it usually accumulates: the longer a neuron is stimulated with a constant DC current input, the more lag each spike will have relative to binary64 reference.

At 0.1 ms time step, fixed point with multiplications with rounding is the best performing solution, but the approach with correct constants plus mixed-precision multipliers performs very well too and without any extra cost added from the default fixed-point set-up explored by Hopkins and Furber [14] and Trensch et al. [15]. Lastly, it is worth noting that these results can be drastically different when neuron parameters are changed or a different ODE solver is used — this is explored further in Chapter 4.

signs until the final multiplication by s16.15 is performed and this would allow to output u0.32 $\times$ u0.32 results as u0.32. This was out of scope of this work and for simplicity s0.31 format was chosen with a negative sign attached at the output instantaneously where needed.

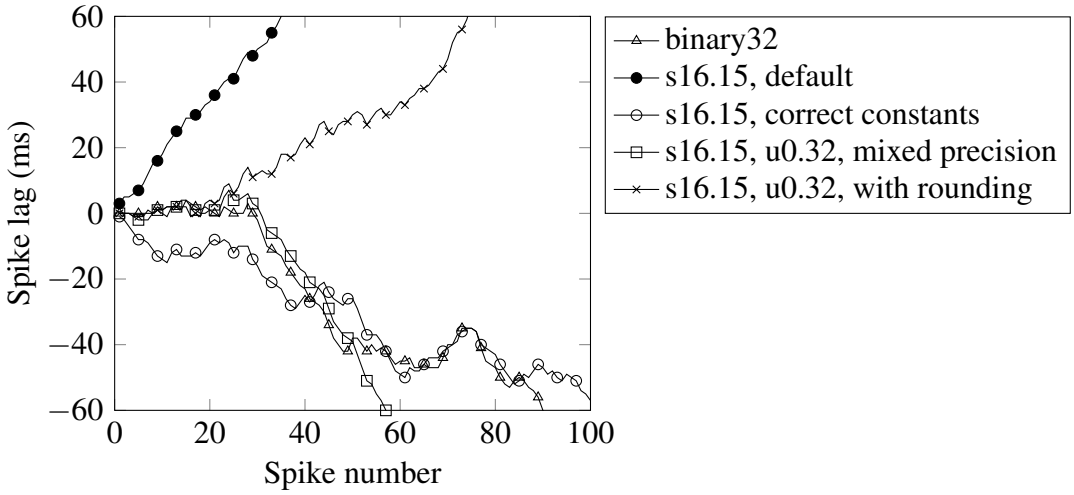

Figure 3.7: Spike lag of the Izhikevich neuron model with different arithmetics and time steps (RK2 midpoint ODE solver [14]) for a duration of 100 spikes with 1 ms time step. Error calculated by comparing to the binary64 version of the solver.

3.1.6 Performance

Table 3.1 shows the time taken for different arithmetics to run a single invocation of (3.2) and (3.3). As expected, binary64 and binary32 floating-point arithmetics are slower than most of the fixed-point arithmetics due to the lack of a hardware support. A first interesting point to notice is that fixed-point GCC arithmetic with mixed-format multipliers takes longer than binary64 arithmetic. As discussed, this is due to the fact that GCC does not support mixed-format multipliers directly and they are performed not inline, which then requires branching, and preserving register contents.

However, custom made multipliers, some of which are available in the SpiNNaker toolchain, are much faster and can be inlined, which is visible in the performance measurements. Non-mixed-format custom multipliers are slightly slower than the GCC equivalents due to the addition of extra checks, such as saturation. It can be seen that when some multipliers are switched from s16.15 $\times$ s16.15 to mixed-format custom multipliers, where saturation is not required because multipliers are smaller than one, computation time decreases. Furthermore, computation time decreases because multiplying a s16.15 by u0.32 and returning s16.15 requires no shifting as the high register from the multiplication instructions contains an answer. Note that additionally, as shown in the previous sections, mixed-format multipliers increase the accuracy of the ODE solver too, so it is a better solution both in terms of accuracy and speed.

Table 3.1: Speed of the integration step of the Izhikevich neuron model using RK2 Midpoint ODE solver for different arithmetics, compiled with the `-Ofast` GCC compiler optimization flag.

Arithmetic	Speed of ODE (μs)
software binary64	9.99203
software binary32	6.68132
fixed-point: default [14] RK2 Midpoint, GCC	0.90881
fixed-point: mixed-precision multipliers, GCC	10.62621
fixed-point: default RK2 Midpoint, custom multipliers	1.86757
fixed-point: mixed-precision, custom multipliers	1.19345
fixed-point: mixed-precision, custom multipliers with RN	1.59792

Some overhead in the ODE solver with custom multipliers in mixed-precision is visible, when compared with a default, because not all of the multipliers in the solver are mixed-format, and those that are not, require saturation checks. Other possible things that can impact performance is whether constants in different fixed-point formats can be represented as immediate values in the arithmetic instructions of ARM or have to be loaded from the memory to a register beforehand. Finally, rounding on the multipliers adds approximately 30 % overhead due to an extra addition instruction on each rounding, as expected.

3.2 Accurate exponential decay

This section describes an approach to improve the exponential decay accuracy in SpiNNaker software [79] and the first SpiNNaker2 prototype chip [59]. Both of these implementations have s16.15 input/output formats. The accelerator was used by Liu et al. [64] to run *softmax function* and by Yan et al. [21] in a *structural plasticity* model on the first prototype SpiNNaker2 chip. Yan et al. [21] ran into limitations of s16.15 exponential and to capture smaller weight changes had to write a 9 cycle software *range reduction* from floating-point to s16.15 and back to obtain wider dynamic range [Private communication]. In this section it is demonstrated that even though the exponential function is made to work in s16.15 the accuracy can still be improved with a negligible performance overhead for negative x arguments, without changing the underlying algorithm. This is done by scaling the input arguments and interpreting the output binary number as being in a different fixed-point format than the nominal s16.15. To evaluate accuracy, software implementation of the exponential function on SpiNNaker

was used, but the demonstrated method is directly applicable to the hardware implementation as well.

3.2.1 Background of the problem

Exponential decay is described as

$$X(t) = X_0 e^{-\frac{t}{\tau_x}}, \tag{3.4}$$

where X_0 is some initial value to be decayed over time t with the decay time constant τ_x. A basic description of this is that because the exponent is negative, as time passes a smaller value is returned by the exponential and therefore the smaller fraction of the current value X_0 is returned. Exponential decay is one of the most common functions in biological models used to model exponentially decaying quantities between neuron spike times.

To model this function on SpiNNaker, Look-Up Tables (LUTs) are usually used where a time constant is fixed throughout the simulation and because time is modelled on a predefined grid such that only certain input values to the exponential can ever be used, the size of these tables is manageable. For more complex simulations, `expk()` with the s16.15 fixed-point format is provided as part of the SpiNNaker software stack [79] (the equivalent of which was designed in hardware for SpiNNaker2 [59]). Given that it takes the input and output arguments as *accum* fixed-point values, the function domain is $x \in [\log_e(2^{-15}) = -10.3972077083..., \log_e(2^{16} - 2^{-15}) = 11.090354888493...]$. To evaluate this function in terms of exponential decay, the absolute error of e^x for $x \in [-10.3972077083991, 0)$ was measured, comparing the `expk()` and C binary64 exponential function. Figure 3.8 demonstrates absolute errors over this range of inputs. It was found that the maximum error in this range is $1.04\varepsilon = 0.0000317382...$, which means that the function, most of the time, returns one of the neighbouring fixed-point values around the C binary64 result and on some occasions a value two steps away.

The method to improve the accuracy of this function is as follows. Noting that the exponential decay in (3.4) generates only negative input values to the exponential function, we know that the output is always in the range $[0, 1)$ (excluding $x = 0$ as a special case). Therefore, 17 bits in the s16.15 number representation are unused (Figure 3.9) because there is no sign and there is no integer part on the output from the exponential decay. A good solution would be to design a specialized implementation

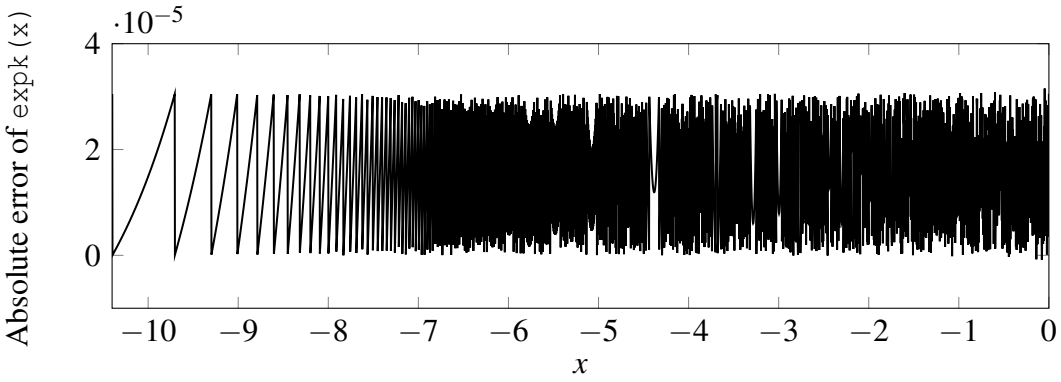

Figure 3.8: Exponential decay function e^{-x} accuracy when calculated with SpiNNaker's `expk()` function which works with s16.15 inputs and outputs.

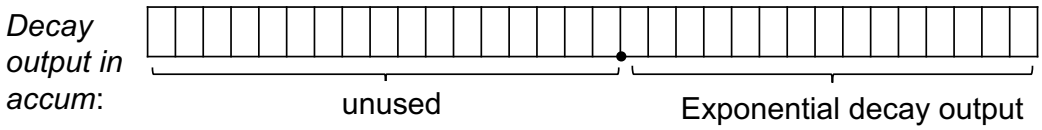

Figure 3.9: Bit-level output description from the standard SpiNNaker exponential decay function.

which takes as an input an s16.15 value and provides a fractional value (u0.32 or s0.31) output to maximize precision of the exponential decay. However, designing a new function incurs an additional effort and in case it is already a hardware routine the most elegant solution would be to manipulate the standard s16.15 exponential function to produce outputs in a different format.

3.2.2 Mixed-precision method for improving the accuracy

To obtain the exponential decay output e^x as s0.31, we need to arrive at $2^{16}e^x$ (a value in s16.15 format shifted 16 places left, which gives the same value when the bits are interpreted as s0.31). However, this cannot be done by simply running the arithmetic algorithm for calculating e^x and then shifting the output as that will place 0's at the bottom part and no accuracy improvement will be achieved. The following exponential function property is of interest: $2^{16}e^x = e^{\log_e(2^{16})}e^x = e^{16\times\log_e(2)+x}$. Now we have $2^{16}e^x$ as a s16.15 or e^x as s0.31 when the binary point location is interpreted to be located at 16 bits to the left. Note that this was achieved not by shifting but by manipulating the exponent x (add the constant $\log_e(2)\times 16$) to get the same effect as first calculating e^x and then shifting 16 places left, without propagating 0's at the bottom end but letting

the exponential algorithm fill in the bottom bits as best as it can.[3] The method is summarized in Figure 3.10. Note that binary point location has to be changed by using the pointer manipulation and not simply casting,[4] for example if we have an *accum* a, we can change the interpretation of the binary point with pointers: `long fract b = *(long fract *)(&a)` (or use `reinterpret_cast<>()`). Then, the multiplier in (3.4) can be modified to shift right by 31 instead of 15 to obtain $X(t)$ as s16.15.

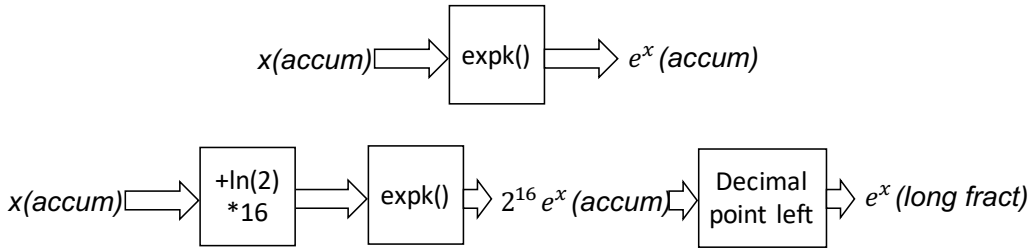

Figure 3.10: Mixing formats for exponential decay. Top — standard *accum* format function. Bottom — a method of scaling the inputs to improve accuracy.

The accuracy of the exponential decay function is now improved significantly as shown in Figure 3.11. The maximum error in the input range $x \in [\log_e(2^{-15}), 0)$ is $0.79\varepsilon_{s16.15} \approx 0.000024108$. However, this maximum error occurs only when x is close to 0, as seen in the diagram. Using this approach, the average error is reduced $\sim 7\times$ from the standard s16.15 implementation of `expk()` function in the SpiNNaker software stack. In more than half of the input domain, the function is up to $40\,000\times$ more accurate, purely due to 16 bits of extra precision on the output.

Additionally, since there are more bits at the output, the input domain of the function is increased to $x \in [\log_e(2^{-31}) = -21.4875625973583..., 0)$, where previously it would saturate to 0 if $x < -10.3972077083991....$ This has an important effect on neuromorphic algorithms because the decay process is now capturing smaller remaining quantities of the decaying variable down to 2^{-31}. One such example is in STDP algorithms, where the weight change Δw_{ij} of a synapse is modelled as follows (more on STDP in Section 3.3):

$$\Delta w_{ij} = \begin{cases} F_+(w_{ij})e^{-\frac{|\Delta t|}{\tau_+}} & \text{if } \Delta t > 0, \\ F_-(w_{ij})e^{-\frac{|\Delta t|}{\tau_-}} & \text{if } \Delta t \leq 0, \end{cases} \tag{3.5}$$

[3] Note that the format u0.32 was intentionally not chosen as $17 \times \log_e(2) + x$ term would cause saturation in the s16.15 exponential function when $x \geq \log_e(0.5)$.

[4] If an s16.15 output from the exponential function would be cast to s0.31, the compiler would simply pick the 15 bottom bits of the value and place them at the top of of the destination s0.31 variable, which is in fact a correct behaviour.

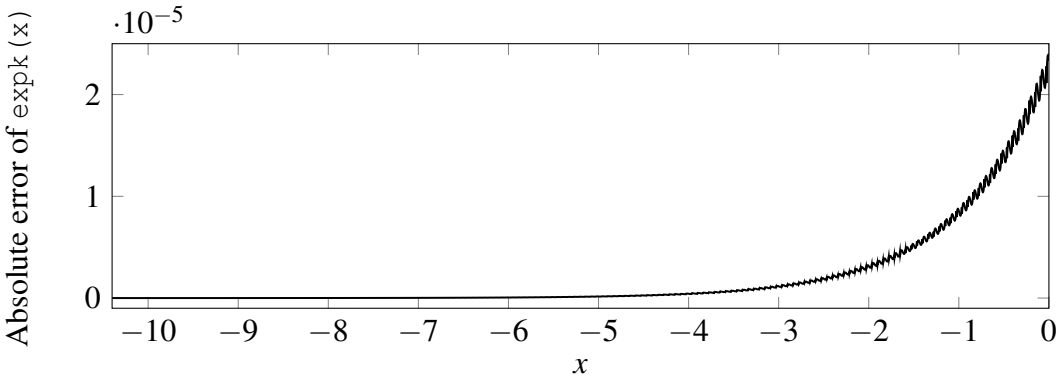

Figure 3.11: Improved exponential decay function e^{-x} accuracy when calculated with SpiNNaker's `expk()` function, with s16.15 input and s16.15 output interpreted as s0.31.

where $\Delta t = t_j - t_i$ is the temporal difference between the post- and presynaptic neuron (connected by a synapse with the weight w_{ij}) spikes, $F_{+/-}(w_{ij})$ describes how the weight change depends on the previous synaptic weight, and $\tau_{+/-}$ are decay constants. It is clear from these equations that in STDP, the only arguments that are important for the exponential function are the negative ones and the output values are between 0 and 1 as it is modelling exponential decay. The input/output data type combination of s16.15/s0.31 is especially suitable for this application instead of the standard s16.15/s16.15 because there will be twice as many more bits at the output from the exponential decay — a desirable property when one wants to model STDP very accurately; and the range of possible input values (temporal distance between presynaptic and postsynaptic spikes divided by the timing constant) before the output saturates to 0 now grows from -10.4 to -21.488.

This is important for short timing constants — for example, with a timing constant $\tau_{+/-} = 1$, the maximum temporal difference, when using the s16.15/s16.15 input/output format, $t_j - t_i$ in 1 ms time steps is 10 ms — more than that and exponential function saturates to 0 which will neglect small weight changes. By mixing input and output data types in this way, we would now capture small weight changes when $\tau_{+/-} = 1$ ms for pre-post temporal difference up to 21 ms. The proposed method does not require any changes to the existing software `expk()` function and can also be used on the hardware accelerator performing s16.15 exponential function which the first SpiNNaker2 prototype chip is equipped with [59].

3.3 Extending the plasticity framework of SpiNNaker

In this section a model of three-factor STDP is developed on SpiNNaker by extending SpiNNaker's plasticity simulation algorithm shown in Section 2.3.3.3. The main focus in this thesis is performance and numerical accuracy issues with the learning rule, especially because it involves multiple calls to the exponential function per spike-pair processing. This learning rule can be useful to model learning based on delayed feedback from the environment, such as path finding and remembering in insects [80]. A network performing classical conditioning was run with this learning rule on SpiNNaker [11].

3.3.1 Synaptic plasticity

Synaptic plasticity describes how the weights of the synapses change over time, and it is believed to be the main mechanism in the brain driving long-term memory and various forms of learning. The theory for this first started from what is now called *Hebbian learning*, which states that the neurons that fire at the same time should have their synaptic connections strengthened so that in the future those neurons have even stronger correlation [81]. As described in this review [8], later, multiple extensions to this idea have been developed, including the weakening of the synapses in certain circumstances. Today both effects are known as Long-Term Potentiation (LTP) and Long-Term Depression (LTD). Correlation between pre- and post-synaptic neurons is the most important aspect, and it was shown experimentally that effects of LTP/LTD on a synapse depend on pre- and post-synaptic neuron spike times in $\sim$100 ms time window; here LTD has a larger time window than LTP in which pre-post spike correlation affects the synapse [8, 82].

STDP is an elegant description of plasticity from the perspective of computation and spiking neural networks, as spike timings are the key information available in these kind of networks. By considering only spike times, we are able to ignore the biophysical causes of plasticity which results in computationally less expensive algorithms that are easier to analyse and set-up for simulation [58]. Therefore this section of the thesis will focus on the computational models of various, most used, STDP functions and not go into detail about more complex biophysical plasticity models. Excellent reviews of timing based plasticity are given by Morrison et al. [58], Frémaux and Gerstner [83] and are usually classified into *short-term plasticity* and *long-term plasticity* and the latter is also subclassified into *two-factor STDP* and *three-factor STDP*.

3.3.2 Two-factor spike-timing-dependent plasticity (STDP)

The most basic description of STDP states that the strengthening of the synapse is induced when a pre-synaptic neuron fires shortly before a post-synaptic neuron and the weakening of the synapse when this is reversed, the post-synaptic neuron fires shortly before the pre-synaptic neuron. A pair-based STDP rule, which approximates the data describing this kind of weight change Δw_{ij} found in pyramidal neurons of rats ([84]) is

$$\Delta w_{ij} = \begin{cases} F_+(w)e^{-\frac{|\Delta t|}{\tau_+}} & if \Delta t > 0, \\ F_-(w)e^{-\frac{|\Delta t|}{\tau_-}} & if \Delta t < 0, \end{cases} \tag{3.6}$$

where $\Delta t = t_j^f - t_i^f$ is the temporal difference between post- and pre-synaptic neuron spike times and $F_\pm(w)$ is a function describing the dependence of the update on the current synaptic weight, and τ_+ and τ_- are timing constants controlling the sizes of the LTP and LTD windows. An approach of choosing which pair of pre- and post-synaptic spikes are considered for a specific weight update must also be specified (the choice of a *spike-pairing scheme*) [58].

As described by Morrison et al. [58] this type of STDP rule can be simulated by keeping track of the spiking activity in the form of an exponentially decaying *history trace*

$$\frac{ds}{dt} = -\frac{s}{\tau} + \sum_{t_i^f} \delta(t - t_i^f), \tag{3.7}$$

which can be thought of as the value which increases by one on each spike at times t_i^f by the Dirac delta function $\delta(t - t_i^f)$, and exponentially decays to zero over time with a time constant τ. When such a trace is kept for each pre- and post-synaptic neuron, on the arrival of the pre-synaptic spike we can look up the current value of the post-synaptic history trace to decide how much depression should be induced, and on the arrival of the post-synaptic spike we look at the pre-synaptic history trace to find the value of the potentiation that should be induced [58]. Therefore the dynamics of the weight w_{ij} between the pre-synaptic neuron i and post-synaptic neuron j with the corresponding history traces s_i and s_j can now be described as

$$\frac{dw_{ij}}{dt} = F_+(w_{ij})s_i(t)\delta(t - t_j^f) - F_-(w_{ij})s_j(t)\delta(t - t_i^f), \tag{3.8}$$

where the Dirac delta function $\delta(t - t^f)$ causes updates to the weight only on spike

times as shown by Morrison et al. [58].

For the weight dependence rule $F_{\pm}(w)$, multiple options are explored in the literature. The main ones are *additive* ($F_{\pm}(w) = A_{\pm}$, a predefined constant controlling the maximum weight change when Δt is close to zero) [57], *multiplicative* ($F_{\pm}(w) \propto w$) [85] and *power law* ($F_{\pm}(w) \propto w^{\mu}$) [67]. The last one is particularly complex due to required general exponentiation function. All of these options for STDP are required by the computational neuroscience community as there is yet no consensus as to the one formulation for STDP that best fits the observed data [58].

3.3.3 Three-factor STDP

Many extensions have been proposed to STDP such as the inclusion of additional spikes [86] (which as pointed out in [58] can replicate experimental data much better than the pair-based STDP) and the post-synaptic voltage [87]. However, while these extensions may improve the ability of STDP to capture the statistical relationship between pre- and post-synaptic activity, Hebbian learning still provides no means of controlling *what* to learn. For example, considering a two layer feed-forward network in which an output neuron is stimulated at the same time by two different input neurons, Hebbian learning will strengthen the synapses between both input neurons and the output. However Hebbian learning rules provide no synapse-level means of associating reward or punishment, surprise or novelty, or any other input that might arrive sometime later, after the output neuron spikes, that could allow the algorithm to learn how to behave next time on the same inputs in order to maximise reward. Dopamine (DA) has been identified as a potential reward signal in the brain [88] and has been experimentally shown to control synaptic plasticity in a large number of ways [89]. On this basis computational neuroscientists developed learning rules based on *neuromodulation* which extend Hebbian learning to include reinforcement from neuromodulators such as dopamine [90, 91].

In most of the models, two extra *history traces* (similar to pre- and post-synaptic neuron spiking traces in Section 3.3.2) are introduced: an *eligibility trace*, which signals that the synapse is a candidate for plasticity due to STDP effects, and a *third-factor trace*, which models the concentration of some neuromodulator *at a synapse*. These two traces are combined to cause plasticity in the synapses, for example dynamics used by Izhikevich [90] (and more recently supported by experimental evidence [89])

to model networks doing something similar to classical conditioning experiments, as

$$\frac{dw_{ij}}{dt} = e_{ij}h_j(M^{\text{3rd}}(t)), \tag{3.9}$$

$$\frac{de_{ij}}{dt} = -\frac{e_{ij}}{\tau_e} + \text{STDP}(\Delta t)\delta(t - t_{i/j}), \tag{3.10}$$

where w_{ij} is the weight of a synapse between a pre-synaptic neuron i and a post-synaptic neuron j, $M^{\text{3rd}}(t)$ is a global third factor and h_j is a function of the third factor specific to a post-synaptic neuron j. A history trace e_{ij} is an *eligibility trace* of the synapse, which step increases on pre- and post-synaptic neuron spikes at times $t_{i/j}$ through the Dirac delta function $\delta(t - t_{i/j})$ by an amount dictated by the $\text{STDP}(\Delta t)$, which in two-factor learning rules is a weight change function in (3.6). It decays exponentially which is controlled by the decay constant τ_e.

3.3.4 Izhikevich learning rule

Izhikevich [90] raises a question: how does an animal know which of the many cues and actions preceding a reward should be credited for the reward? It is shown that a dopamine-modulated STDP model has a built-in *instrumental conditioning* property — the associations between actions and rewards are learned automatically by reinforcing the firing patterns (networks of synapses) responsible, even when the firings of those patterns are followed by a delayed reward or masked by other network activity. To achieve this each synapse has an *eligibility trace C*:

$$\frac{dC}{dt} = -\frac{C}{\tau_c} + \text{STDP}(\Delta t)\delta(t - t_{i/j}), \tag{3.11}$$

where τ_c is the decay time constant of the eligibility trace and $\text{STDP}(\Delta t)$ represents the magnitude of the change to make to the eligibility trace in response to a pair of pre- and post-synaptic spikes with temporal difference $\Delta t = t_j - t_i$. Finally, $\delta(t - t_{i/j})$ is a Dirac delta function used to apply the effect of STDP to the trace at the times of pre- or post-synaptic spikes.

The concentration of dopamine is described by a variable D:

$$\frac{dD}{dt} = -\frac{D}{\tau_d} + D_c \sum_{t_d^f} \delta(t - t_d^f), \tag{3.12}$$

where τ_d is the time constant of dopamine trace, D_c is the increase in dopamine concentration caused by each incoming dopaminergic spike and t_d^f are the times of these spikes. (3.11) and (3.12) are then combined to calculate the change in synaptic strength W with

$$\frac{dW}{dt} = CD. \tag{3.13}$$

As discussed in Section 3.3.2, when a post-synaptic spike arrives very shortly after a pre-synaptic spike, a standard STDP rule would immediately potentiate the synaptic strength. However when using the three-factor STDP rule, this potentiation is instead applied to the eligibility trace. Because changes to the synaptic strength are gated by dopamine concentration D (3.13), changes are only made to the synaptic strength if $D \neq 0$. Furthermore, if the eligibility trace has decayed back to 0 before any dopaminergic spikes arrive, the synaptic strength will not be changed.

Because (3.13) describes a continuous weight change, it cannot be directly evaluated within the event-driven STDP algorithm of SpiNNaker, where pre-synaptic spikes trigger STDP updates. However, it can be transformed into a form suitable for event-driven evaluation. Firstly consider the C and D traces in (3.11) and (3.12). Similarly to the pre- and post-synaptic STDP traces discussed in Section 3.3.2, between the times at which spikes occur, both of these equations are simple first-order linear ODEs. Therefore we can write down the following equations to calculate the change of C and D between spikes:

$$C_{ij} = C_{ij}(t_c^{last}) e^{-\frac{t - t_c^{last}}{\tau_c}}, \tag{3.14}$$

$$D_j = D_j(t_d^{last}) e^{-\frac{t - t_d^{last}}{\tau_d}}, \tag{3.15}$$

where t_c^{last} is the time of the last eligibility trace update (when either a pre- or post-synaptic spike caused an STDP update) and t_d^{last} is the time of the last dopamine trace update (which occurs when a dopamine spike is received). So that each (post-synaptic) neuron can be independently targeted by dopaminergic spikes, the dopamine trace values (D_j) are stored in the post-synaptic history structure along with the post-synaptic traces (s_j) and event times (t_j) in a similar manner to the "target spikes" in the learning rule presented by Nichols et al. [92].[5] However, because the eligibility traces (C) are

[5] An alternative would be that each synapse stores a separate variable containing the level of dopamine, in which case the PyNN network would have to be specified in terms of dopaminergic spikes arriving at sets of synapses. For simplicity it was chosen to model dopamine level per neuron, which is changed by dopaminergic neuron sending spikes into it — this is very suitable implementation in the

individual to each synapse, they must be stored alongside the synaptic weights (w_{ij}) in SDRAM and can thus only be updated within the PROCESSROW (Algorithm 1) function when they have been transferred into local memory. We can now substitute (3.14) and (3.15) into (3.13) to obtain the weight change dynamics:

$$\frac{dw_{ij}}{dt} = C(t_c^{last})D(t_d^{last})e^{-\frac{t-t_c^{last}}{\tau_c}}e^{-\frac{t-t_d^{last}}{\tau_d}}. \tag{3.16}$$

Now, by integrating the preceding equation, the total weight change since the last update at t_{last} can be found:

$$\Delta w_{ij} = C(t_c^{last})D(t_d^{last})\int_{t^{last}}^{t} e^{-\frac{t-t_c^{last}}{\tau_c}}e^{-\frac{t-t_d^{last}}{\tau_d}}dt, \tag{3.17}$$

$$\Delta w_{ij} = \frac{C(t_c^{last})D(t_d^{last})}{-\left(\frac{1}{\tau_c}+\frac{1}{\tau_d}\right)}\left(e^{-\frac{t-t_c^{last}}{\tau_c}}e^{-\frac{t-t_d^{last}}{\tau_d}} - e^{-\frac{t^{last}-t_c^{last}}{\tau_c}}e^{-\frac{t^{last}-t_d^{last}}{\tau_d}}\right). \tag{3.18}$$

The final update Algorithm 1 in Section 2.3.3.3 makes to each synaptic weight is to apply the effect of the pre-synaptic spike at time t. Therefore, if the Algorithm 1 is extended to update the eligibility trace, t_{old} will always represent the last time C was updated: $t^{last} = t_c^{last} = t_{old}$. Furthermore, before the inner loop over the post-synaptic events occurs, the last dopamine trace value is decayed to t_{old} using (3.15). Therefore, $t^{last} = t_d^{last} = t_c^{last} = t_{old}$, meaning that (3.18) can be simplified to

$$\Delta w_{ij} = \frac{C(t_{old})D(t_{old})}{-\left(\frac{1}{\tau_c}+\frac{1}{\tau_d}\right)}\left(e^{-\frac{t-t_{old}}{\tau_c}}e^{-\frac{t-t_{old}}{\tau_d}} - 1\right). \tag{3.19}$$

Algorithm 2 shows the extended, three-factor STDP algorithm. The functions `applyPostSpike` and `applyPreSpike` used to directly update the synaptic weight in Algorithm 1 are now instead used to update the eligibility trace (C_{ij}). When pre- or post-synaptic events are applied, (3.14) is used to decay the eligibility trace (C_{ij}) and (3.19) is used to update the synaptic weight (w_{ij}). Finally, as previously discussed, (3.15) is used to obtain the decayed D_j trace values at t_{old} and t_j.

To allow users to implement dopaminergic synapses sPyNNaker [53] software was modified, the SpiNNaker implementation of PyNN [52]. To implement the dopamine signal, dopaminergic neuron type was introduced that communicates through a special

SpiNNaker environment as this requires only a separate type of synapse which will not cause contributions to the membrane potential but instead to the dopamine level at the post-synaptic neuron.

Algorithm 2 Algorithmic implementation of three-factor STDP

function PROCESSROW(t)
  for all j **in** *postSynapticNeurons* **do**
    $history \leftarrow getHistoryEntries(j, t_{old}, t)$
    $(t_{prev}, s_{prev}, D_{prev}, type_{prev}) \leftarrow getPrecedingHistoryEntry(t)$

    $t_c \leftarrow t_{old}$
    $D_c \leftarrow D_{prev}\text{EXP}(\frac{-(t_c - t_{prev})}{\tau_D})$

    for all $(t_j, s_j, D_j, type_j)$ **in** *history* **do**
      $w_{ij} \leftarrow w_{ij} + \frac{C_{ij}D_c}{-\left(\frac{1}{\tau_C}+\frac{1}{\tau_D}\right)}\left(\text{EXP}(-\frac{(t_j - t_c)}{\tau_C})\text{EXP}(-\frac{(t_j - t_c)}{\tau_D}) - 1\right)$
      $C_{ij} \leftarrow C_{ij}\text{EXP}(-\frac{t_j - t_c}{\tau_C})$
      if $type_j$ **is not** *dopamine* **then**
        $C_{ij} \leftarrow applyPostSpike(C_{ij}, t_j, t_{old}, s_i)$
      $D_c \leftarrow D_j$
      $t_c \leftarrow t_j$

    $(t_j, s_j, D_j, type_j) \leftarrow getLastHistoryEntry(t)$

    $w_{ij} \leftarrow w_{ij} + \frac{C_{ij}D_c}{-\left(\frac{1}{\tau_C}+\frac{1}{\tau_D}\right)}\left(\text{EXP}(-\frac{(t - t_c)}{\tau_C})\text{EXP}(-\frac{(t - t_c)}{\tau_D}) - 1\right)$
    $C_{ij} \leftarrow C_{ij}\text{EXP}(-\frac{t - t_c}{\tau_C})$
    $C_{ij} \leftarrow applyPreSpike(C_{ij}, t, t_j, s_j)$

    $addWeightToRingBuffer(w_{ij}, j)$

  $s_i \leftarrow addPreSpike(s_i, t, t_{old})$
  $t_{old} \leftarrow t$

type of synapse (similar to the "target synapses" employed by Nichols et al. [92]) which do not cause updates to the membrane potential of the post synaptic neuron but simply bring information about dopaminergic spikes into it. This approach has the advantage of allowing any type of PyNN neuron to act as a source of dopaminergic spikes. When a core receives a dopaminergic spike it is not added to the delay ring buffer but is instead added directly to the post synaptic history structure where it can be accessed by Algorithm 2.

3.3.5 Numerical accuracy

Apart from precision loss due to weight scaling [53] (which can be improved with rounding), there are 4 main issues that cause numerical errors of this plasticity model on SpiNNaker.

- The size of the exponential decay LUTs is limited by the constraints of local memory, therefore discretization of the LUT has to be done if long time constants are used.
- When a specific time value Δt lies between two entries available in the LUT, the smaller one is chosen irrespective of how close the time is to the larger entry.
- s4.11 fixed-point format for traces and exponential decay LUTs is used as calculated to be optimal by Knight [56, Sec. 4.3.2]. However, there is no reason for having any integer and sign parts in the exponential decays as the outputs from the exponential decay are in the range $(0,1]$. The entries in this table can be stored as u0.16, then the traces multiplied by this and, if required at the bottom of register, shifted right by 16 places to return decayed traces in s4.11 — this will give 5 extra bits of accuracy on the decay.
- There is no rounding on multipliers when exponential decay is applied.

3.3.5.1 Exponential look-up tables on SpiNNaker

Figure 3.12 shows how exponential decay look-up tables $e^{-\frac{t}{\tau_x}}$ for some time constant τ_x are generated on the host and what values are stored inside SpiNNaker memory for different time shift factors $\tau_{x,shift} \in \mathbb{Z}$ where $\tau_{x,shift} \geq 0$. First of all, at the host side in Python, function $e^{-\frac{t}{\tau_x}}$ is calculated for every value $t \in [0, \text{SIZE} \times 2^{\tau_{x,shift}}]$ in steps

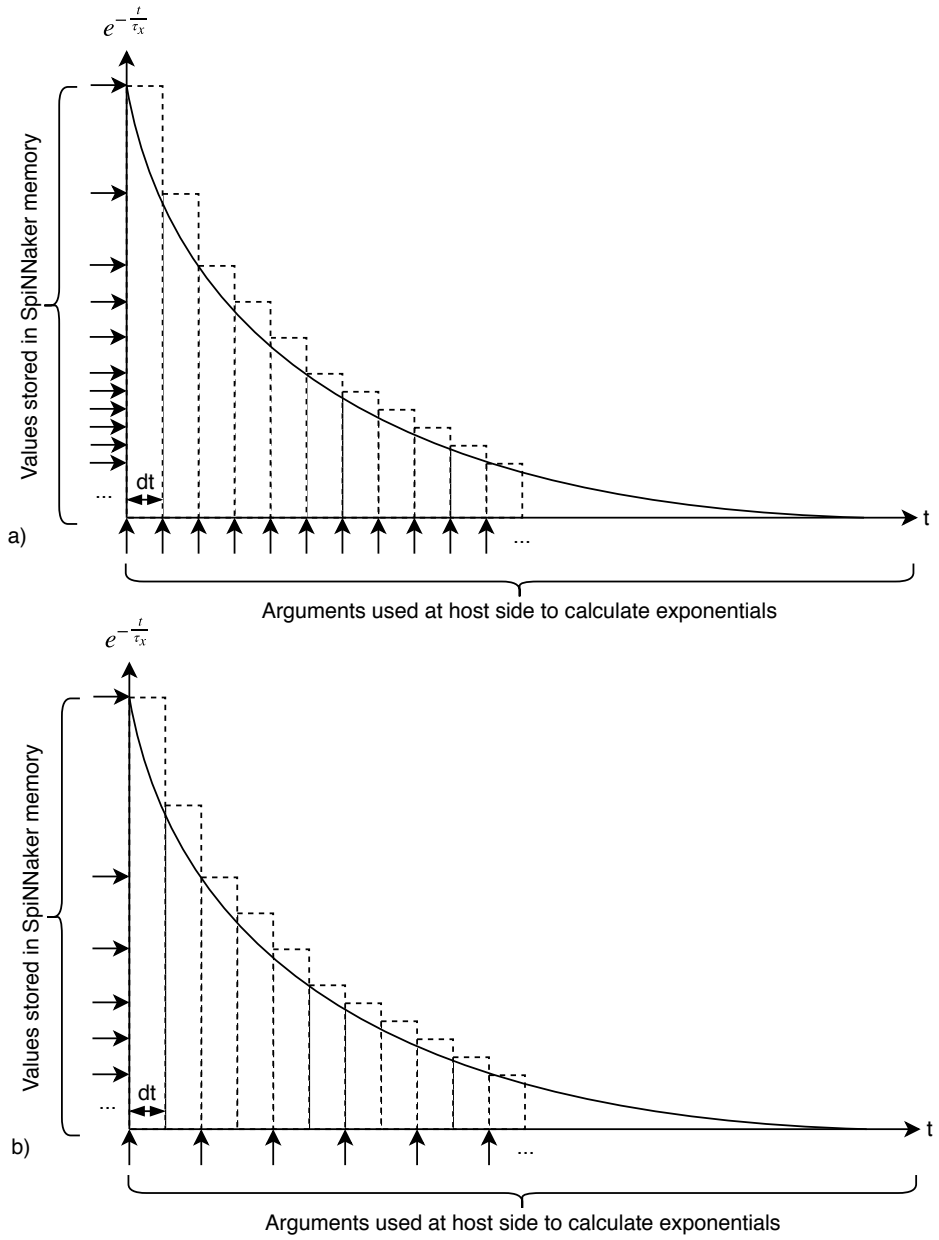

Figure 3.12: Illustration of how exponential look-up-tables are generated on the host machine and used in SpiNNaker. Picture a) shows values stored for each time step dt. Picture b) shows values stored for every second argument — this is controlled by setting a macro $\tau_{x,shift} = 1$ in C source code. The solid curve shows the ideal function and the dashed blocks depict only those values that are needed in the event-driven simulation (multiples of time step dt).

of $dt \ll \tau_{x,shift}$, where dt is a simulation time step and $\ll$ and $\gg$ are logical shift operations. Then, these values are copied into the SDRAM of each SpiNNaker chip.

When the simulation starts, each core responsible for modeling the dynamics involving exponential decay with the timing constant τ_x, copies this look up table into DTCM. Finally, when the simulation is running, the exponential decay for some value t is obtained by indexing the look-up table with an index $t \gg \tau_{x,shift}$. SIZE and $\tau_{x,shift}$ can be tuned beforehand to control how large and what accuracy exponential decay look-up tables will be generated for each time constant. Note that when $\tau_{x,shift} \neq 0$, not all arguments t on the x-axis get their corresponding exponential value allocated to them. If, at the time of simulation, the program is accessing exponential decay of an argument without a corresponding entry in the LUT, the closest available larger value is returned (round-down is used on conversion from time to LUT index).

3.3.5.2 Testing numerical accuracy

A simple test case was developed for neuromodulated STDP to trigger a single weight update. This test case is not necessarily covering all possibilities, but should be general enough to give an impression of what happens numerically in a lot of STDP simulation configurations that a user can come up with and is enough to demonstrate improvements to the arithmetic — although a more extensive study with various test cases and a bigger network would be useful in the future.

A PyNN script is shown in Appendix B for the following test case. There are three neurons: post- and pre-synaptic neurons, with a synapse that is *plastic*, and a dopaminergic neuron connecting to the post-synaptic neuron, with a synapse that is identified as causing Dopamine (DA) level to increase at the post-synaptic neuron's end. Two pre-synaptic spikes are added, at times 1500 and 2400 ms. The post-synaptic neuron is stimulated at 1502 and fires at time 1503 ms. The dendritic delay is 1 ms so the post-synaptic time is at 1504 ms when processed in STDP. The dopamine neuron spikes at 1600+1 ms (also with added dendritic delay). The additive STDP all-to-all pairing rule is used with parameters $\tau_+ = 10$, $\tau_- = 12$, $A_+ = 1$, $A_- = 1$, $\tau_c = 1000$, and $\tau_d = 200$ as in [11].

In this scenario, there is only one plasticity invocation (of Algorithm 2), at time 2400 when a pre-synaptic spike arrives. Here, two spikes are in the post-synaptic history (1504 and 1601 (DA)) but none of them cause a weight update. The first one causes the eligibility trace to increase, but since the dopamine trace is zero at that point, there is no weight increase. Then, the DA spike at 1601 causes the dopamine trace to

increase and the eligibility trace to decrease from the previous value to the value at time 1601, but since we are using the previous value of dopamine trace in (3.19), there is no weight update. Finally, on the pre-synaptic spike at 2400, both eligibility and dopamine traces are non-zero and the weight is updated. At time 1504 the eligibility trace obtains the value from STDP of

$$A_+ \times e^{-\frac{1504-1500}{\tau_+}} = 0.670320046035639.$$

Then at time 1601 it is decayed to

$$0.670320046035639 \times e^{-\frac{1601-1504}{\tau_C}} = 0.608352983811176.$$

Finally, at time 2400, (3.19) is applied:

$$\Delta w = \frac{0.608352983811176 \times 0.1}{-0.006}(e^{-\frac{2400-1601}{1000}} \times e^{-\frac{2400-1601}{200}} - 1) \approx 10.0553.$$

By running the same experiment on SpiNNaker, which uses fixed-point arithmetic with various tradeoffs for memory space and speed as discussed above, the final weight evaluates to 10.0087890625, which means an absolute error of approximately 0.04648. While this error did not damage the qualitative behaviour of the classical conditioning experiment run on SpiNNaker [11], it might cause issues in general and therefore it is worth addressing this numerical inaccuracy and exploring the options for improvement by changing various parts of the SpiNNaker plasticity framework.

3.3.5.3 Improving accuracy

Multiple basic improvements were applied to the arithmetic involved: 1) store exponential decay LUT entries in u0.16 16-bit fixed-point format instead of s4.11; 2) add rounding on all 16-bit multipliers involved in the weight update; 3) take the nearest value in the exponential decay LUT instead of the smaller one (again this is rounding of the index $t \gg \tau_{x,shift}$ before shifting). For using u0.16 exponential decays, the multipliers had to be modified to take one signed and one unsigned value, where the default SpiNNaker framework uses the instruction $smulbb(x, decay)$ which sign-extends each input value before multiplying. A shift of 16 steps right is also required instead of 11. None of this impacts performance, except rounding. After these changes, the final weight from the above experiment was 10.09130859375, with an error of approximately 0.03600859375. This error is slightly lower than before but still significant and

is mainly due to the s4.11 numerical format.

As the 16-bit format accuracy is now exhausted, it can be beneficial to try to hold some of the multiplication results in (3.19) in higher precision. The main source of error is multiplication by $\frac{1}{-(\frac{1}{\tau_c}+\frac{1}{\tau_d})} = -\frac{1}{0.006} = -166.666....$ A large value in this constant multiplier magnifies the error in the multiplicand that is in low precision with the format s4.11. To reduce this, start by rewriting (3.19):

$$\Delta w_{ij} = -\frac{C(t_{old})D(t_{old})}{-\left(\frac{1}{\tau_c}+\frac{1}{\tau_d}\right)}\left(1 - e^{-\frac{t-t_{old}}{\tau_c}} e^{-\frac{t-t_{old}}{\tau_d}}\right). \tag{3.20}$$

This way the multiplication of two exponentials in u0.16 format can be kept in u1.31 format (one integer bit left to cover the case when exponentials return 0). Then this is subtracted from 1 and multiply by the traces. The traces are in s4.11 format, therefore their multiplication $C(t_{old})D(t_{old})$ can be kept in s8.22 format (without loss of precision). Then, the multiplication

$$-C(t_{old})D(t_{old})\left(1 - e^{-\frac{t-t_{old}}{\tau_c}} e^{-\frac{t-t_{old}}{\tau_d}}\right)$$

returns a 64-bit answer and the result is shifted right 31 places to return to s8.22 format. Finally, the multiplication by $-166.666...$ can be performed and because this time the multiplicand has more correct bits at the bottom, the multiplication result error is reduced. After these improvements, the final weight from SpiNNaker was 10.07421875 with an error of 0.01891875.

Further error reduction can be achieved by holding intermediate values in even more precise formats, and storing the exponential decays as u0.32 values, either by using twice as much memory for the LUTs or using software `expk()` or an accelerator with the mixed-format improvements discussed in Section 3.2. This was tested by extending the LUT and the final weight update was 10.06591796875 with the error of 0.01261796875 ($\sim 4\times$ improvement). Without extensive investigation, this is probably as good as it can get with history traces held in the s4.11 format.

3.3.6 Incoming spike processing performance

This section explores performance of the learning rule presented in Section 3.3.4 without any of the aforementioned accuracy improvements, as it was published [11].

SpiNNaker machines have no form of global synchronisation. Therefore each core

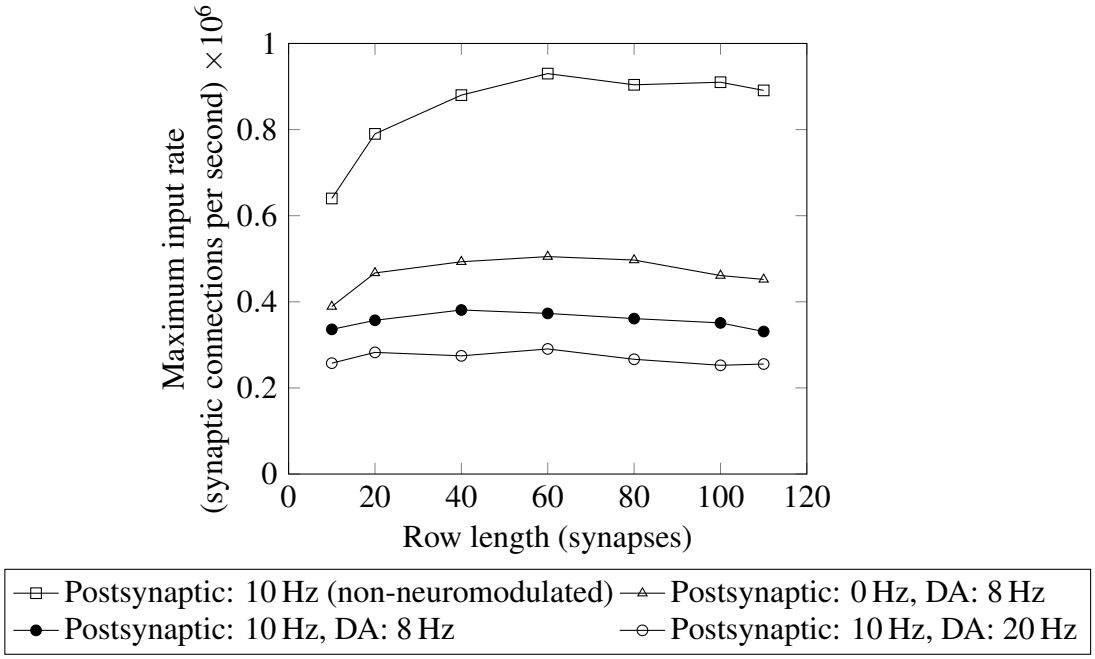

Figure 3.13: Comparison between the incoming spike processing performance of standard STDP (with an additive weight dependence) and three-factor STDP for different post-synaptic and DA spiking rates.

needs to update the state of each neuron it is responsible for simulating and process any incoming spikes it has received within a predetermined simulation time step (typically 1 ms). This means that the number of neurons simulated on each core, the complexity of the neuron or synapse model, the density of connectivity and the rate of incoming spikes all need to be balanced to guarantee real time operation.

In Figure 3.13 a comparison is shown of the incoming spike processing performance of a single SpiNNaker core simulating a population of LIF neurons with standard STDP and three-factor STDP synapses. The extra local memory required to store dopamine trace values in the post-synaptic history structure means that, when using the three-factor STDP algorithm described in the previous section, each core is limited to simulating 126 neurons. As Knight and Furber [55] discussed, the length of synaptic matrix rows has a significant effect on synapse processing performance. This is because, beyond the computational cost of processing each synapse, there is a large fixed cost in processing each row meaning that the best performance is obtained with long row lengths.

Following the procedure outlined by Knight and Furber [55], the population of neurons is stimulated with an increasing number of 10 Hz Poisson spike trains in order

to determine the maximum incoming spike rate that the core could process in real time. Additionally, because the number of events in the post-synaptic history structure affects the performance of Algorithms 1 and 2, the post-synaptic firing rate is varied by injecting a fixed current into the simulated neurons. Because, in the case of three-factor STDP, incoming dopaminergic spikes also get added to the post-synaptic history structure, the performance is also measured with a single dopaminergic neuron, firing at 8 Hz, connected to all neurons in the benchmark population.

This showed that the highest number of inputs into a single core can be achieved with rows approximately 60 synapses long (50 % connection density). Furthermore, as the post synaptic rate is increased, more synaptic history traces have to be processed on each pre-synaptic spike, so overall performance decreases. As predicted, performance with short synaptic rows (10–40 synapses per neuron) suffers from the fixed row-processing overheads mentioned earlier in this section.

It is also worth noting that with very long synaptic rows (80–120 synapses per neuron) performance is also reduced. In [11] it was predicted that this happens because, with very long rows that take a long time to process, even small variations in the number of spikes emitted by the Poisson sources each time step can overrun the time available. The maximum number of synaptic connections incoming into a core simulating a network spiking at 10 Hz and neuromodulated with an 8 Hz dopaminergic signal was 0.38 million which is approximately two times slower than the simplest additive STDP rule.

Note that this learning rule has 4 different decay constants and it has multiple calls to exponentials in Algorithm 2. In the cases where a small simulation time step is used, exponential LUTs will become larger than with the time step of 1 ms used in this study and will not fit into the local memory. In those cases it would likely be needed to call the actual exponential function which would add further overhead to the Algorithm 2. Therefore, these kind of learning rules are the main motivation for adding a hardware accelerator for the exponential function into neuromorphic chips.

3.3.7 Scaling to cortical levels of connectivity

SpiNNaker was designed around the assumption that each ARM core would be responsible for simulating 1000 neurons with 1000 input synapses, each receiving spikes at a mean rate of 10 Hz [76]. However, over recent years, it has been found that, on average, cortical neurons have 8000 input synapses [93, 94, 95] each receiving spikes at the lower average rate of around 3 Hz [96]. While Knight and Furber [55] showed that

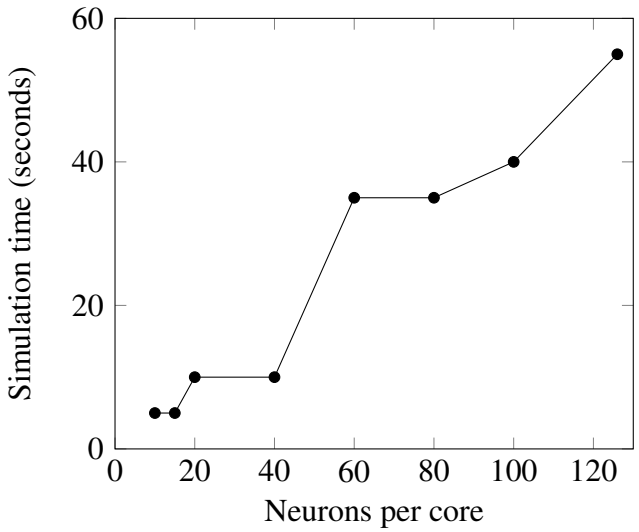

Figure 3.14: Time taken to simulate 5 s of network activity with 8000 inputs to each neuron. The mean firing rate of each input is 3 Hz, the post synaptic firing rate is 3 Hz, and the mean firing rate of the dopaminergic neuron is 4.5 Hz.

SpiNNaker was capable of simulating neurons in real-time with these higher levels of connectivity, even when using standard STDP this required reducing the number of neurons simulated on each core to around only 30. The alternative is to slow down the simulation to some fixed fraction of real-time.

Figure 3.14 demonstrates how the number of neurons per core and the simulation speed can be traded off when simulating neurons with three-factor STDP and cortical levels of connectivity. Each neuron in the network is densely connected to 8000 Poisson sources firing at 3 Hz as well as to a single dopaminergic neuron spiking at an average rate of 4.5 Hz. To simulate this network in real-time, the maximum number of neurons that could be placed on each core was only 15. However, when the number of neurons simulated on each core was increased to the maximum of 126, the simulation had to be slowed down by a factor of $11\times$.

3.4 Conclusion

This chapter presents numerical issues and methods to reduce them, in the two main levels of neuromorphic simulation: neurons and synaptic plasticity.

First, numerical accuracy of a well known Izhikevich neuron model was explored;

this model is famous for its configurability to provide a diverse range of spiking patterns and it has a low computational cost compared to some other more biologically-realistic models. It is characterised by an ODE, which in SpiNNaker is solved for updating on every simulation time step in fixed-point arithmetic with RK2 Midpoint ODE solver. Multiple sources of literature have demonstrated that this neuron has spike lagging due to certain issues with fixed-point arithmetic. It was demonstrated in this chapter that, in fact, an Izhikevich neuron in fixed-point arithmetic can do as well as with floating-point arithmetic if we are prepared to undertake a few simple modifications: specify constants correctly rounded; specify constants and parameters as accurately as possible in a given n-bit fixed-point arithmetic; and round on arithmetic operations where possible. It was shown that with these modifications spike lag is reduced significantly and becomes negligible in a commonly used configuration of the neuron (RS neuron).

The second and third parts were about exponential decay accuracy and new plasticity rules. It was shown that the s16.15 exponential decay function can be improved by manipulating the inputs and outputs without changing the actual software or hardware algorithm of the function. Furthermore, a new plasticity rule was implemented on SpiNNaker and was used to explore numerical accuracy, fixed-point arithmetic properties, and performance. Numerical accuracy was explored in a certain test case with multiple methods for improving it, similar to the methods used in the first part of the chapter for the neuron model.

The techniques shown in this chapter will be useful in the current generation SpiNNaker platform and will inform SpiNNaker2 as well. The Izhikevich ODE solver was shown to be very accurate, however, it was studied here only in one setting and other types of spiking behaviours or ODE solvers are worth looking into. It appears that when neuron parameters are changed, the RK2 Midpoint ODE solver can cause large spiking lags even with binary32 floating-point arithmetic. Therefore the next chapter is dedicated to exploring these issues in more depth.

3.5 Acknowledgements

The author thanks Michael Hopkins who has pointed out an issue of arithmetic versus algorithmic errors in testing the numerical accuracy of the Izhikevich neuron model and Jamie Knight for providing initial code and equations for developing the plasticity rule presented in this chapter.

Chapter 4

Stochastic Rounding

Replacing standard floating-point arithmetic by fixed-point arithmetic typically results in increased numerical errors. On SpiNNaker this mainly happens when simulating complex neuron models or plasticity learning rules in fixed-point arithmetic as shown in the previous chapter. In this chapter the numerical errors in the solution of ODEs are further investigated. The Izhikevich neuron model is used in two configurations and with four ODE solvers to demonstrate that rounding has an important role in producing accurate spike timings and that fixed-point arithmetic with round to nearest used in the previous chapter does not always perform accurately. It is shown that fixed-point arithmetic with stochastic rounding consistently results in smaller errors compared to binary32 floating-point arithmetic and fixed-point arithmetic with round to nearest when tested on four ODE solvers and two neuron configurations. These results demonstrate that on average stochastic rounding can bring the numerical accuracy of this neuron model simulated in 32-bit fixed-point arithmetic close to that of binary64 floating-point arithmetic, by paying some cost for generating random numbers. Furthermore, an accelerator for stochastic rounding is designed and evaluated for SpiNNaker2 which conveniently already has a PRNG in hardware.

A major part of this chapter was reproduced from the material that was published in the Philosophical Transactions of the Royal Society A journal [33].

4.1 Introduction and motivation

Recent interest in deep and convolutional networks has led to the development of highly-effective systems that incur vast numbers of computational operations, but where the requirements for high-precision are limited. As a result, reduced precision formats,

such as fixed-point formats of various word lengths, are becoming increasingly popular in architectures developed specifically for machine learning. Where high throughput of arithmetic operations is required, accuracy is sacrificed by changing the working numerical format from floating- to fixed-point and the word length from 64- or 32-bit to 16- or even 8-bit precision [97]. By reducing the word length, precision is compromised to gain the advantage of a smaller memory footprint and smaller hardware components and buses through which the data travels from memory to the arithmetic units. On the other hand, changing the numerical format from floating- to fixed-point significantly reduces the range of representable values, increasing the potential for the under/overflow of the data types.

There are other approaches apart from floating- and fixed-point arithmetics that are worth mentioning: *posit arithmetic* [72] is a completely new format proposed to replace floating-point format and is based on the principles of interval arithmetic; bfloat16, with hardware support in recent Intel processors [36], is simply a single precision floating-point format (binary32 [22]) with the 16 bottom bits dropped for hardware and memory efficiency; and various approaches for improving floating-point arithmetic using, for example, the *logarithmic number system* in which multiplication becomes addition [98], and dynamically-sized exponent and significand fields which optimize the relative accuracy of the floating-point format in some specific range of real numbers rather than having the same relative accuracy across the whole range [99].

When considering low-precision arithmetics, it is worth comparing fixed-point and floating-point hardware costs. It was reported, to cite a few examples, that a 32-bit integer adder uses $9\times$ less energy [26] than a binary32 floating-point adder. A different study also reported $30\times$ lower area [100] (but note that the energy improvement in this study is unclear and not related to $9\times$ speedup from [26]). As well as reducing the precision and choosing a numerical format with a smaller area and energy footprint, mixed-precision arithmetic [101], stochastic arithmetic [39] and approximate arithmetic [28] have also been explored in the machine learning community with the goal of further reducing the energy and memory requirements of accelerators. Mixed-precision and stochastic techniques help to address precision loss when short word length numerical formats are chosen for the inputs and outputs to a hardware accelerator. Mixed-precision arithmetic maintains intermediate results in formats different from the input and output data, whereas stochastic arithmetic works by using probabilistic rounding to balance out the errors of different signs that accumulate in various

algorithms. Approximate arithmetic (or more generally *approximate computing*) is a recent trend which applies the philosophy of adding some level of (application tolerant) error at the software or inside the arithmetic circuits to reduce energy and area with minimum damage to the application's performance.

Interestingly, similar ideas have been explored in other areas, with one notable example being at the birth of the digital audio revolution where the concept of *dither* became an important contribution for improving the quality of digital audio systems [102, 103]. Similarly, an *approximate dithering adder*, which alternates between directions of error bounds to compensate for individual errors in the accumulation operation, has been demonstrated [104].

The main motivation for using stochastic rounding can be seen when thinking about a basic summation of rounded values, for example continuously adding values in high precision to a value held in lower precision, a situation which can arise in various places where arithmetic operations involved in some algorithms provide results in higher precision than the destination variable and is precisely the situation that arises in the ODE solvers for the Izhikevich neuron on SpiNNaker. For example, let's assume four computations that output a number 0.25 in high precision which has to be rounded before accumulating into the result of lower precision, just an integer. This situation is too simple to arise in any practical scenario but is a useful demonstration. A standard round to nearest scheme would provide a result like this: $\mathit{result} = \mathrm{RN}(0.25) + \mathrm{RN}(0.25) + \mathrm{RN}(0.25) + \mathrm{RN}(0.25) = 0$, which has an error of 1. On the other hand, SR, which rounds up with a probability (in this case 1 out of 4 times) proportional to the *residual* (value of the trailing bits after the rounding bit) might give: $\mathit{result} = \mathrm{SR}(0.25) + \mathrm{SR}(0.25) + \mathrm{SR}(0.25) + \mathrm{SR}(0.25) = 1$, which has an error of 0. It is also possible to get as a result 0, 2, 3 or 4, but the probability of this happening is smaller. An ODE solver is doing something similar at larger scale — multiple high precision values are produced on each step from multiplications, which are either added together or multiplied further before finally being added to the previous value in lower precision and the distribution of residuals from some of the multiplications is biased due to various constants.

The main contributions in this chapter are as follows.

- The accuracies of ODE solvers for the Izhikevich neuron are assessed using different rounding routines applied to operations on fixed-point formats. This builds on earlier work where the accuracy of fixed-point ODE solvers was investigated for this model [14], but where the role of rounding was not addressed.

- Fixed-point ODE solvers with *stochastic rounding* (SR) are shown to be more robust than with other rounding algorithms, and are shown experimentally to be more accurate than binary32 floating-point ODE solvers (Section 4.4).
- 6 bits in the residual and random number are shown to be sufficient for improving the accuracy of ODE solvers with SR (Section 4.4).
- A hardware unit for stochastic rounding and saturation of 64/32-bit values is proposed for SpiNNaker2 and evaluated in synthesis.

4.2 Related work

This section reviews some of the papers that explore fixed-point ODE solvers and stochastic rounding applications in machine learning and neuromorphic hardware.

4.2.1 Fixed-point ordinary differential equation solvers

The most recent work that explores fixed-point ODE solvers on SpiNNaker [15], already discussed in Chapter 3, demonstrates a few important issues with the default GCC s16.15 fixed-point arithmetic when used in the Izhikevich neuron model. The authors tested the current sPyNNaker software framework [53] for simulating the Izhikevich neuron and then demonstrated a method for comparing the network statistics of two forward Euler solvers at smaller time steps using a custom fixed-point format of s8.23 for one of the constants. Some other observations about this study are as follows.

- The iterative use of forward Euler solvers with small time steps is far from optimal in terms of the three key measures of accuracy, stability and computational efficiency. Regarding the latter, it seems that the proposed solver is approximately 10x slower than the RK2 Midpoint solver explored by Hopkins and Furber [14].
- The choice of the s8.23 format to improve the accuracy of the constant 0.04 could be replaced with the u0.32 format and appropriate mixed-format arithmetic operations could be used for any constants smaller than 1; u0.32 $\times$ s16.15 multiplication is supported by GCC as discussed in Chapter 3 but can also be performed very efficiently using a single multiply instruction (reminder: the s16.15 answer is the top register of the two registers that ARM returns, if no rounding is required).

- Rounding is not explored as a possible improvement in the conversion of constants and arithmetic operations and neither is the provision of an explicit constant for the nearest value that can be represented in the underlying arithmetic format.

4.2.2 Stochastic rounding

Randomization of rounding can be traced back as early as the 1950s [105]. Furthermore, a similar idea was also explored in the context of the CESTAC [106, 107] method where a program is run multiple times, making random perturbations of the last bit of the result of each elementary operation and then taking a mean to compute the final results (for more information see [23, p. 486] and references therein).

In [40] the authors investigate the effects of probabilistic rounding in backpropagation algorithms. Three different applications are shown with varying degrees of precision in the internal calculations of the backpropagation algorithms. It is demonstrated that when fewer than 12 bits are used, training of the neural network starts to fail due to weight updates being too small to be captured by limited precision arithmetic, resulting in underflow in most of the updates. To alleviate this, the authors apply probabilistic rounding in some of the arithmetic operations inside the backpropagation algorithm and show that the neural network can then perform well for word widths as small as 4 bits. It is concluded that with probabilistic rounding, the 12-bit version of their algorithm performs as well as a binary32 floating-point version.

In a more recent paper [39] about SR (and similarly in [108]), the authors demonstrate how the error resiliency of neural networks in deep learning can be used to allow reduced precision arithmetic to be employed instead of using the standard 32/64-bit formats and operations. The authors describe a simple matrix multiplier array built of many multiply-accumulate units that multiply two 16-bit arguments and accumulate the results in full-precision (implementing an exact dot product operation). Furthermore, they show that applying SR when converting the result of the dot product into lower 16-bit precision improves the neural network's training and testing accuracy. The accuracy with a standard round to nearest routine is demonstrated to be very poor while stochastic rounding results are shown to be almost equal in accuracy to the same neural network using binary32 floating-point format.

A very important result from this study is that 32-bit fixed-point formats can be reduced to 16 bits and maintain neural network's accuracy, provided that SR is applied. This reduction in precision allows for smaller and more energy efficient arithmetic

circuits and reduced memory requirements in neural network accelerators. However, this study did not report the effects of variation in the neural network results due to stochastic rounding — as applications using SR become stochastic themselves it is important to report both *mean* benchmark results and the *variation* across many trial runs. A recent paper from IBM [109] also explores the use of the 8-bit floating-point format with mixed-precision in various parts of the architecture, and applies stochastic rounding. A similar accuracy to the binary32 format in training neural networks is demonstrated.

Closely related to the work above, the effects of training recurrent neural networks, used in deep learning applications, on analogue resistive processing units (RPUs) have been studied [110]. The aim is to minimize the analogue hardware requirements by looking for a minimum number of bits that the *input arguments* to the analogue parts of the circuit can have. A baseline model with 5-bit input resolution is first demonstrated and it becomes significantly unstable (in terms of neural network training error getting larger) as network size is increased, compared to a simulation model with binary32 floating-point arithmetic.

The authors then increased the input resolution to 7 bits (with rounding to nearest reported on this version, not mentioned on the baseline version) and observed a much more regular development of the training error and with lower magnitude at the last training cycle. Finally, it is shown that stochastically rounding the inputs to 5 bits again makes the training error almost as stable as the 7-bit version without the large error observed when using the baseline version for training large networks. It is important to observe that differently from any other reported results that utilized SR, here SR is applied on the inputs to the application, therefore in a sense randomizing the data to improve the accuracy of the algorithm.

In neuromorphic computing SR is used on the Intel *Loihi* neuromorphic chip [13]. Here SR is not applied to the ODE of the neuron model as it is in this study, but to the biological learning rule — STDP — that defines how synapses change with spiking activity (Section 3.3.2). The implementation of STDP usually requires spike history traces to be maintained for some number of timesteps. In Loihi, history traces (for each synapse) are held as 7-bit values to minimize memory cost and calculated with SR as $x[t] = \alpha \times x[t-1] + \beta \times s[t]$, where α is a decay factor which defines how the trace decays back to 0 between spikes and β is a value contributed by each spike.

Algorithm 3 Stochastic rounding by comparison

```
function SATSR_INT64_INT32(X, n)
    P ← PRNG32()
    MASK ← ((1 ≪ n) − 1)
    P ← P&MASK
    RESIDUAL ← X&MASK
    X ← X ≫ n
    if P < RESIDUAL then
        X ← X + 1
    if X > MAX_INT32 then
        return MAX_INT32
    if X < MIN_INT32 then
        return MIN_INT32
    return X
```

4.3 Implementing stochastic rounding

The initial goal is to implement and test fixed-point SR arithmetic operations with various input and output formats, including nonhomogeneous (mixed-format) configurations.

4.3.1 Implementation

As demonstrated in Section 2.1.4, stochastic rounding can be implemented by inspecting residuals and utilizing a PRNG. For a fully configurable stochastic rounding routine we have to be able to round a specified number of bits n of a 64-bit number. This means that bits $n-1$ to 0 are zeroed and 0x1 is added at the n-th bit location if rounding is performed. Usually there is no need to return the rounded value in the original bit width with bottom bits zeroed, therefore an output number from the rounding routine is usually provided in lower precision, at which point it also has to be saturated to that smaller precision if the value is too large to be represented. For SpiNNaker use cases we are interested in rounding a 32-bit multiplication result (which is 64 bits) to some 32-bit fixed-point format, therefore an algorithm and implementation for this routine was explored.

There are at least two ways to round a value stochastically: by comparing a random number to the residual and rounding up if it is smaller, or by adding a random number to the residual and letting the carry out from that control rounding. Algorithms 3 and 4

Algorithm 4 Stochastic rounding by addition

```
function SATSR_INT64_INT32(X, n)
    P ← PRNG32()
    P ← P&((1 ≪ n) − 1)
    X ← (X + P) ≫ n
    if X > MAX_INT32 then
        return MAX_INT32
    if X < MIN_INT32 then
        return MIN_INT32
    return X
```

Algorithm 5 Rounding to nearest with round up on a tie

```
function SATRN_INT64_INT32(X, n)
    X ← (X + (1 ≪ (n − 1))) ≫ n
    if X > MAX_INT32 then
        return MAX_INT32
    if X < MIN_INT32 then
        return MIN_INT32
    return X
```

demonstrate how to do both. Stochastic rounding by addition looks shorter, but both algorithms require 5 operations in the main rounding parts (saturation is the same in both cases). Saturation logic is self explanatory and it is worth noting that if the input and output numbers are unsigned, we would have to perform only one comparison instead of two. Also it is worth noting that rounding to nearest can be implemented similarly to Algorithm 4 as shown in the Algorithm 5. By comparing Algorithms 4 and 5, notice that SR has an overhead of a PRNG plus one extra operation. Depending on the optimization level and whether rounding is inlined or not, these algorithms might be compiled for a fixed shifting n and therefore optimized on compilation for specific use cases.

For testing the Izhikevich neuron ODE, a configurable multiplication routine was developed which always performs saturation but can be configured globally to either do round down, round to nearest or round stochastically. There are 5 cases of interest (with 5 equivalent cases for 16-bit numerical formats):

- $\text{s16.15} \times \text{s16.15} \rightarrow \text{s16.15}\ (n = 15)$,
- $\text{s16.15} \times \text{s0.31} \rightarrow \text{s16.15}\ (n = 31)$,
- $\text{s16.15} \times \text{u0.32} \rightarrow \text{s16.15}\ (n = 32)$,

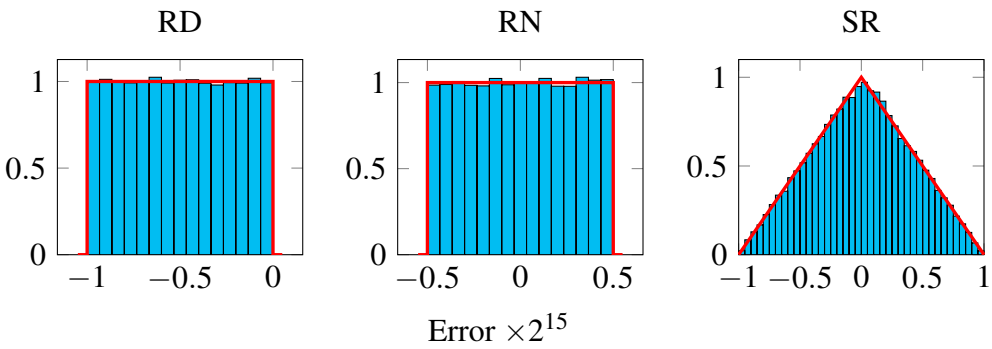

Figure 4.1: Histograms showing the error distribution of 50000 random s16.15 × s16.15 operations with different rounding schemes. RD, RN and SR are round down, round to nearest and stochastic rounding respectively.

- u0.32 × u032 → s0.31 ($n = 33$), and
- u0.32 × s0.31 → s0.31 ($n = 32$).

All these are configured by passing a different constant n into a variant of Algorithm 4 after the multiplication result is obtained. Only the first multiplication requires saturation check. In the case where $n = 33$, the answer is pre-shifted right by 1 step before rounding to avoid difficulties with a 33 bit residual (if we are strict, this requires a 33-bit random number, which is probably unnecessary and requires two registers to be operated on).

4.3.2 Testing multiplication

To establish the correctness of different rounding routines for fixed-point formats it is useful to carry out tests to assess the distribution of errors from a set of multiply operations with a wide range of randomly generated inputs. To test this, 50000 random numbers distributed uniformly between a minimum and maximum value are taken, making sure that the output does not saturate, to evaluate only the rounding error. Each pair of operands is generated as numbers that can be exactly represented in the relevant fixed-point format and then converted to binary64 so there is no error in the inputs. Multiplications in fixed-point and binary64 are then performed. The fixed-point result is converted to binary64 and the difference is multiplied by $\frac{1}{\varepsilon}$ to obtain error in units of least precision.

Figure 4.1 shows the error distributions of the s16.15 × s16.15 multiplier with the three different rounding routines. As expected, multiplication results from a range of

randomly chosen input values have uniformly distributed residuals and therefore RD and RN show this in rounding error. However, SR has a triangular distribution — as the errors become larger, fewer numbers are assigned to having that error, which shows a correct property of stochastic rounding which works on a principle of *rounding occasionally incorrectly* (incorrectly — not to the nearest value). And how occasionally it does that for a given number is proportional to the residual that has to be rounded. For example, results that should have an error of 0.1, when SR is enabled are sometimes rounded incorrectly into the other direction, producing an error of -0.9, hence this triangular distribution.

4.3.3 Testing in summation

The main advantages of SR can be detected in summing algorithms, where rounding errors biased into one direction dominate the final error in the result of the sum. Following the approach taken by Higham and Pranesh [111] harmonic series was run which is a *divergent series* but converges when implemented in limited precision arithmetic [112]. The series is defined as $\sum_{i=1}^{\infty} 1/i = 1 + \frac{1}{2} + \frac{1}{3}...$ — it can be seen that the addends are getting smaller while the total sum keeps increasing and as Higham and Pranesh [111] reported the sum converges in floating-point arithmetic when the addends become small enough that they do not change the total sum anymore (due to very different exponents and round off error on addition). In another paper Blanchard et al. [113] calls this a *stagnation* problem which happens in summing algorithms in floating-point arithmetic.

This experiment was run in 32- and 16-bit fixed-point arithmetic. The sum has a numerical format s16.15 or s8.7 and is initialized to 1. Then the series is started from $i = 2$ and the division is done in either 32-bit or 16-bit fractional format u0.32 or u0.16 and the addend is rounded to the sum's format with various rounding routines.

Table 4.1 demonstrates the results with three floating-point formats and various fixed-point formats. 5 million iterations were chosen to have a manageable run time, but the number of iterations to convergence is also reported. As expected, most of the fixed-point formats converge as soon as the addends in the series become small enough to be evaluated at lower than s16.15 precision, when the values cross 0.5ε. However, it can be seen that fixed-point with SR can accurately replicate the sum of the binary64 format in 5 million iterations without converging. Given that stochastic rounding is probabilistic rounding, it might still produce some effect in later iterations stochastically and therefore it can be said that *it never converges — there is a diminishing, but*

Table 4.1: Iterations until convergence of the harmonic series for different arithmetics. Sums and errors relative to binary64 (double-precision floating-point) at 5 millionth iteration are reported. Floating-point data is from Higham and Pranesh [111]. Averaged sums are from running the experiment 50 times in s16.15 and s8.7 SR arithmetics.

Arithmetic	Sum at $i = 5 \times 10^6$	Error at $i = 5 \times 10^6$	Iterations to converge
binary64	16.002	0	$2.81... \times 10^{14}$
binary32	15.404	0.598	2097152
binary16	7.086	8.916	513
s16.15 RN	11.938	4.064	65537
s16.15 RD	10.553	5.449	32769
s8.7 RN	6.414	9.588	257
s8.7 RD	5.039063	10.963	129
s16.15 SR (50 runs)	Mean = 16.002 std.dev. = 0.012	−0.000135765	$2^{32} + 1$
s8.7 SR (50 runs)	Mean = 11.205 std.dev. = 0.242	4.797	$2^{16} + 1$

non-zero probability of rounding up the addends and affecting the sum. In practice it converges also when the numerical format of the addends runs out of bits and the probability of rounding up becomes 0. This can also happen if there is a limited number of random bits available for performing stochastic rounding.

4.3.4 Pseudo-random number generators

The reference PRNG is a version of Marsaglia's KISS64 algorithm [114]. This has had several versions — SpiNNaker's default implementation uses a version called KISS99 [115]. As discussed more in Hopkins et al. [33] this generator passed some very challenging tests and is considered of high quality. Results with SR presented here have also been checked using faster PRNGs that fail these challenging tests but which are considered to be of a good basic quality for non-critical applications. In reducing order of sophistication these are a 33-bit LFSR algorithm implemented within the SpiNNaker base runtime library *sark*, and a very simple linear congruential algorithm with a setup which is given in [116, p. 284] where it is called *ranqd1*. No significant differences in either mean or standard deviation of the results presented in the following section were found which indicates that SR, at least on this application, is insensitive to the choice of PRNG as long as it meets some quality standard (this is open for further research).

4.4 Ordinary differential equation solvers

The solution of ODEs is an important part of mathematical modelling in computational neuroscience. In this study the primary interest is in solving ODEs for neuron behaviour though they can also be used for synapses and other more complex elements of the wider connected network such as gap junctions or neurotransmitter concentrations. Many of the neuron models that we are interested in are formally *hybrid systems* in that they exhibit both continuous and discrete behaviour. As a reminder of the Izhikevich neuron model in Section 3.1, the continuous behaviour is defined by the time evolution of the internal variables describing the state of the neuron and the discrete behaviour is described by a threshold being crossed, a spike triggered followed by a reset of the internal state, and then sometimes a refractory period set in motion whereby the neuron is temporarily unavailable for further state variable evolution.

4.4.1 Algorithmic error and arithmetic error

In the original study Hopkins and Furber [14] most focused on *algorithmic error*, exploring how closely a chosen ODE solver can match output from the reference implementation. This algorithmic error is created by the inability of a less accurate ODE solver method to match the reference result (assuming infinite precision arithmetic). As the output in this kind of ODE model is best described as the time evolution of the state variable(s) it is by no means easy to formulate a measure that allows direct comparisons to be made. Hopkins and Furber [14] used spike lag/lead relative to a chosen reference as that is the main data of interest to neural modellers and is easy to observe and measure and the same measurement was used in Chapter 3 and in this study as well.

It should be carefully noted that in this chapter, as well as in the previous, the *arithmetic error* is the main point of interest. An ODE solver is chosen and the reference for comparing different arithmetics is the same ODE solver calculated in IEEE binary64 floating-point arithmetic. So the purpose of this study is different from the original investigation by Hopkins and Furber [14] — the goal is not to compare different ODE solvers to a more advanced Mathematica solver, but to compare different arithmetics and rounding modes using the same solvers. The chosen algorithm for solving an ODE is imperfect but it is a realistic use case on which to test the different arithmetics that are of interest.

4.4.2 Results from applying stochastic rounding

This section presents the results from four different ODE solvers with fixed-point arithmetic and stochastic rounding added on each multiplier, at the stage where the multiplier holds the answer in full precision and has to truncate it into the destination format, as described in Section 2.1.4.

Results below are from testing four different fixed-timestep algorithms to demonstrate some generality. In increasing order of complexity and execution time these are two 2^{nd} order and one 3^{rd} order fixed-timestep Runge-Kutta algorithms (see for example [117]) and a variant of the Chan-Tsai algorithm described in [118]. All are implemented by Hopkins and Furber [14] using the ESR (explicit solver reduction) approach described by the authors where the combination of ODE definition and solver mechanism are combined and "unrolled" into an equivalent algebraic solution which can then be manipulated to be optimised for speed and accuracy of implementation using the fixed-point formats available. Hopkins and Furber [14] mainly focused on 1 ms timestep and the Chapter 3 in this work focused on both 1 ms and 0.1 ms time steps. In this chapter only 0.1 ms timestep was used for simplicity and because 0.1 ms is generally becoming more of interest for this neuron model and across different parts of neural simulation.

A combination of more accurate representation of the constants, mixed-format multiplication and implementation of RN mode demonstrated in Chapter 3 has significantly reduced the error in the fixed-point solution of the Izhikevich neuron model from what was previously reported [14]. Therefore, all the results below with RD on the multipliers do not reproduce the results from the previous study [14] because these new results are generated with more precise constants as well as some reordering of arithmetic operations in the originally reported ODE solvers to keep the intermediate values in s0.31 and u0.32 formats as long as possible.

The experiments were run on the Spin3 board, which contains 4 SpiNNaker chips with ARM968 cores which were used to test the ODE solvers.

4.4.2.1 Neuron spike timings

To test ODE solvers and different arithmetics, the same approach is taken here to stimulate the neuron as in Chapter 3 plus a new type of neuron (with a different set of parameters) is added. This is a constant DC input of $\sim$4.775 nA for the RS and Fast Spiking (FS) neurons (as defined by Izhikevich [75]) originally chosen by Hopkins and

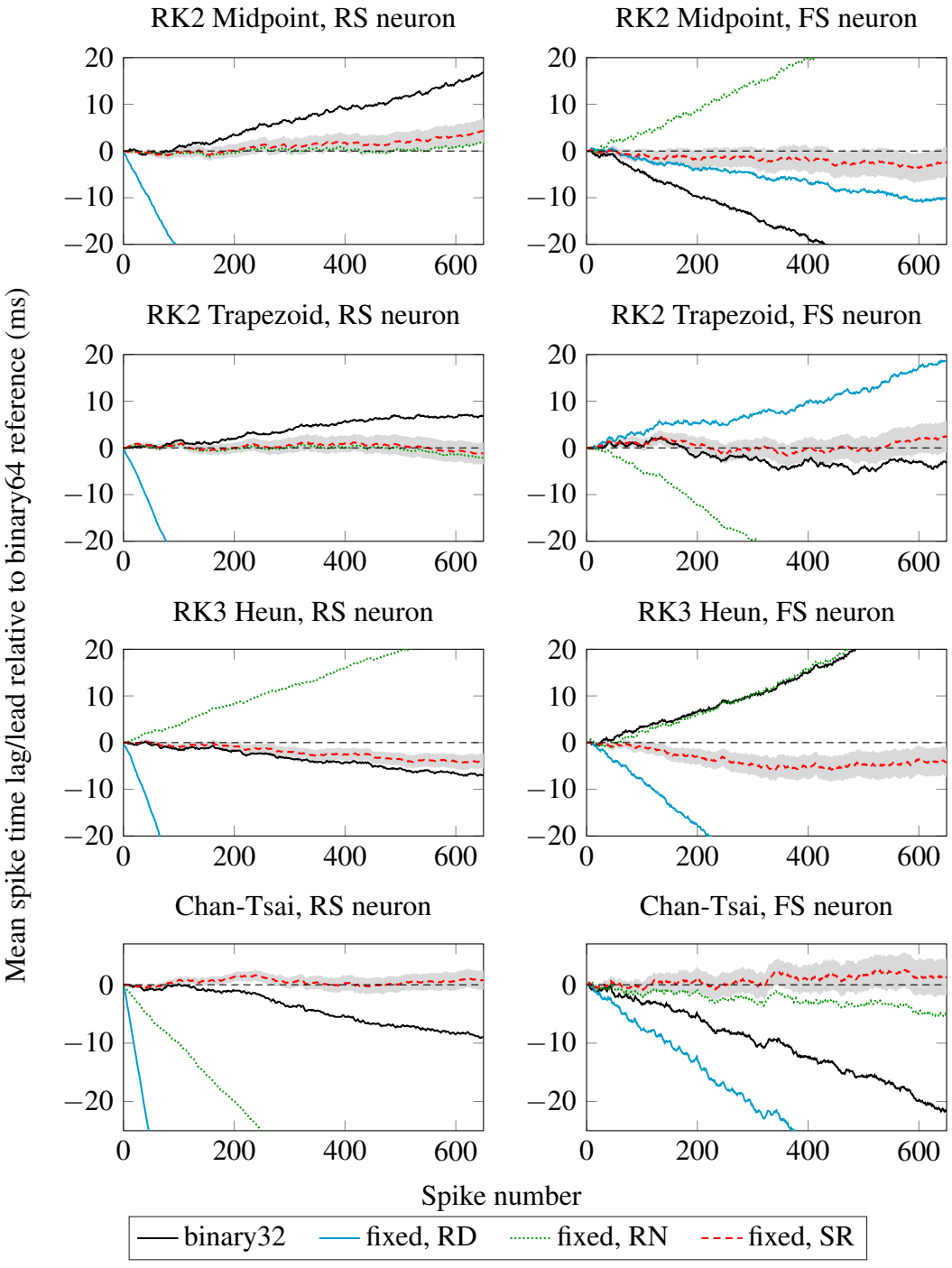

Figure 4.2: Spike lags of the neuron in DC input test at 0.1 ms timestep. Spike lags are computed against the binary64 floating-point reference in each case. SR result is shown as the mean from 100 runs with shaded area showing the standard deviation. A negative value on the Y axis indicates a lead, a positive value indicates a lag.

Table 4.2: Summary of ODE results: spike lags (ms) of the 650th spike in the DC input test. Positive means lag, negative — lead. Fixed, {RD/RN/SR} refers to fixed-point arithmetic with round down, round to nearest and stochastic rounding respectively.

Solver	Neuron type	binary32	fixed, RD	fixed, RN	fixed, SR (std.dev.)
RK2	RS	16.8	-131.4	1.9	4.3 (2.62)
Midpoint	FS	-29.7	-10.0	33.7	-2.3 (3.16)
RK2	RS	6.9	-172.7	-2.1	-1.2 (2.30)
Trapezoid	FS	-3.2	18.7	-40.1	2.3 (3.33)
RK3	RS	-7.1	-206.9	26.0	-4.0 (1.59)
Heun	FS	29.4	-53.6	31.4	-4.4 (3.10)
Chan-	RS	-9.0	-356.3	-67.9	0.8 (1.60)
Tsai	FS	-21.7	-44.6	-5.1	1.4 (3.10)

Furber [14] and subsequently used by Trensch et al. [15]. The results for 4 different solvers are shown in Figure 4.2 with the exact spike lags of the last, 650th spike, shown in Table 4.2.

Because the SR causes spike times to be slightly different on every run of the test, if a different pseudo-random number stream is used each time, it has to be taken into account when generating results. Producing a single set of spike times from one run is not enough anymore as in Chapter 3. This has been done by running the ODE solver 100 times with a different random number stream and recording spike times on each run. From this data the mean and standard deviation of this distribution is calculated. Standard deviation in Figure 4.2 is shown as a shaded area around the SR mean curve.

From the plots and last spike statistics the SR results are very accurate in all cases compared to the alternatives, and in 7 out of the 8 cases are closest to the arithmetic reference after 650 spikes ($\sim$1 min of simulation). Note that the RK2 Midpoint, RS neuron test case, used in Chapter 3, is also visible here for a much longer run time. As was shown in that chapter, fixed-point RN and binary32 were performing almost as well as binary64 floating-point arithmetic. However, from this longer run it becomes clear that surprisingly, while fixed-point RN continues to be quite close to binary64, binary32 starts accumulating error approximately after 100th spike. Also, notice how RK2 Midpoint performance changes when the RS neuron is replaced with the FS neuron — in this case both fixed-point RN and binary32 perform poorly. As a matter of interest, the RK2 Trapezoid algorithm found to produce the most accurate solutions at 1 ms without correct rounding of constants or multiplications demonstrated by Hopkins

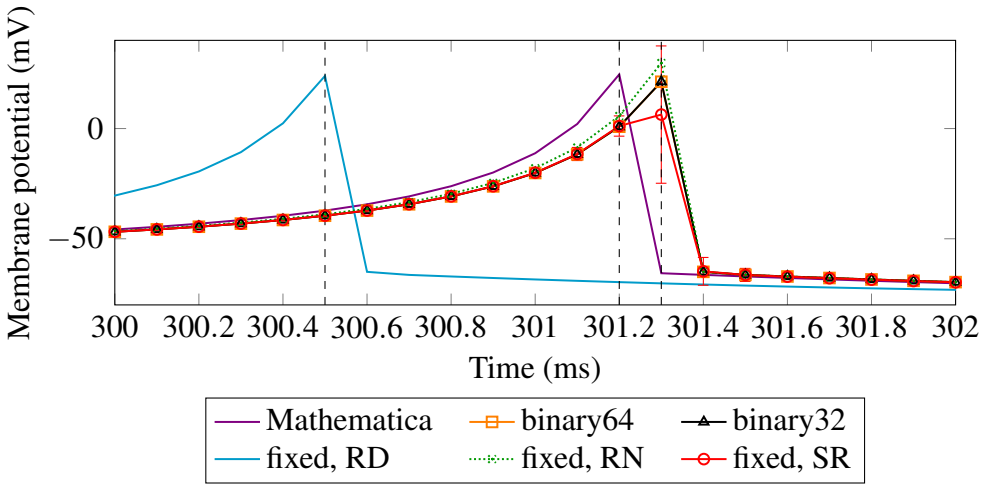

Figure 4.3: The membrane potential of a neuron producing a third spike in the DC current test, using RK2 Midpoint solver with various arithmetics and rounding modes. The temporal difference between the Mathematica and binary64 floating-point arithmetic spike times is *algorithmic error* and the temporal gap between (for example) fixed-point RD and binary64 is *arithmetic error*. Fixed, {RD/RN/SR} refers to fixed-point arithmetic with round down, round to nearest and stochastic rounding respectively.

and Furber [14] continues to provide a good performance here in terms of arithmetic error, producing mean spike time errors of only -1.2 ms and 2.3 ms for the RS and FS neuron models after 69 and 165 seconds of simulation time respectively, and is quite a bit faster than the third-order solvers.

In general, all of these results demonstrate that when SR is used with fixed-point arithmetic, ODE solvers become robust to changing the neuron parameters. Other arithmetics in most cases start producing more lag when the neuron type is switched. The same can be said of changing the ODE algorithm — SR is always close to binary64 floating-point arithmetic irrespective of which solver is used while other arithmetics do well in one solver and do poorly in another solver.

4.4.2.2 Evolution of membrane potential V

As an illustration of the imperfect evolution of the underlying state variables in the Izhikevich ODE, Figure 4.3 shows the progression of the V state variable (the membrane potential of the neuron) after 300 ms for a variety of solver/arithmetic combinations and 0.1ms timestep. One spike and reset event is shown in each case.

The absolute algorithmic reference given by Mathematica is shown in purple. The

significant spike time lead given by RD is shown in the light blue line. The other results show a slight lag relative to the absolute reference, and all are very close to the arithmetic reference in orange. The SR result shows the mean with the standard deviation error bars because it contains 100 random results. The large standard deviation at the spike event and increased standard deviation near it is caused by spikes in some runs out of 100 being reset or having spiked at this timestep.

4.4.2.3 The effects of reduced comparator precision in SR

To build an efficient hardware unit to perform SR it is useful to consider how many bits are required in the SR algorithm. There are two ways to implement SR as shown in Section 4.3: the first is to directly build a comparator between the residual of the number to be rounded $\left(\frac{x-\lfloor x \rfloor}{\varepsilon}\right)$ and the random number (as in (2.6)); another approach that is equivalent, implemented by Gupta et al. [39] in an FPGA to optimize the utilization of the DSP units, was to add the random number to the input number and then truncate the fraction. This reverses (2.6) in such a way that the random numbers that are higher than or equal to $1 - \left(\frac{x-\lfloor x \rfloor}{\varepsilon}\right)$ will produce a carry into the result bits (round up) and the numbers that are lower will not produce a carry (which will result in round down due to binary truncation). The first approach has a comparator of the remainder's/random number's width plus an adder after truncation of the remainder, whereas the second approach has only a single adder of the full word width. Whichever approach is used, the hardware cost of the adder/comparator could be reduced by defining a minimum number of bits required in the residual $\frac{x-\lfloor x \rfloor}{\varepsilon}$ and the random number.

To explore this problem in the ODE test bench used above, a series of ODE solvers solving two types of the Izhikevich neuron were run and the effects on the spike lag different number of bits in SR had were measured. The results are shown in Figure 4.4. These results can be compared to the numbers in Table 4.2 where all the available bits were used in SR. Note that the fixed-point multipliers in the ODE solvers can use 15 or 32 bits in SR, depending on the formats of arguments. It is clear that as we decrease from 12 to 6 bits in the SR, the degradation in quality of lead or lag relative to the arithmetic reference is negligible. However, when the number of bits is decreased from 6 to 4 and fewer, the RS neuron starts to lead with a very high spike timing difference of the 650th spike. Given these two tests on RS and FS neuron, it is concluded that 4-bit SR is acceptable, with some degradation in quality of the neuron model, whereas 6-bit version performs as well as the 12-bit or full remainder length SR version (15 or 32 bits).

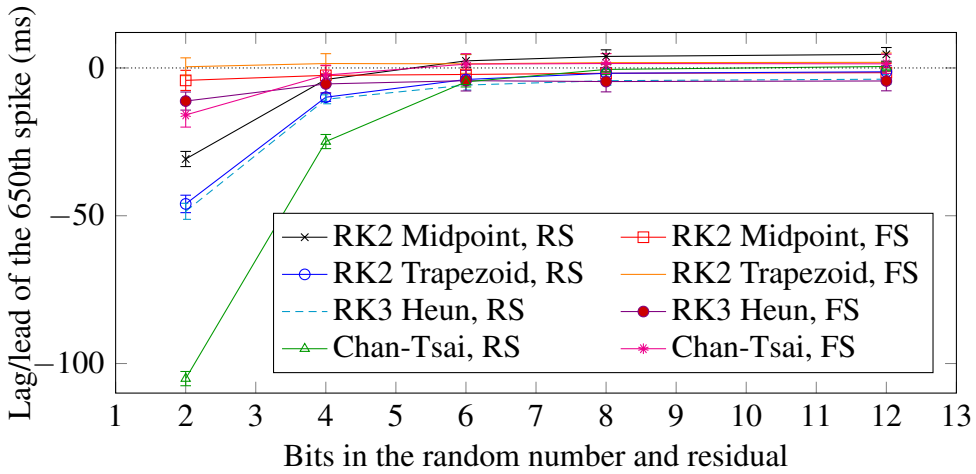

Figure 4.4: Average spike lags of the 650th spike with the standard deviation from 100 runs. Four solvers with two different neuron types each are shown for varying numbers of bits in the stochastic rounding comparison step. All versions are compared to an equivalent reference solver implemented in binary64 floating-point arithmetic (the dotted straight line at $y = 0$ ms).

4.4.2.4 16-bit arithmetic formats

Using the same constant DC current test applied to a neuron as in Section 4.4.2.1, 16-bit numerical formats for this problem were evaluated. The results with 16-bit ISO standard [34] fixed-point formats in the two second-order ODE solvers are shown in Table 4.3. Because most of the variables are now held in s8.7 numerical format, with 7 bits in the fractional part, even SR performs quite badly. However, it is clearly better than RD and RN, with RD causing neuron to lag in spike times the most and RN being subject to underflow in one of the variables that causes updates to the main state variable to become 0, and therefore no spikes are produced. It seems that SR helps the ODE solver to recover from underflow and produce more reasonable answers with a certain amount of lag.

It is likely that custom 16-bit data types used within the solver for key interim variables (for example by scaling the 16-bit variables relative to a maximum known value) would improve this performance further. Most importantly if overflows in the adders can be avoided by rearranging the operations or adding saturation to reduce the error caused by a wrap-around of the 8 integer bits in s8.7. This is a good tradeoff between performance and accuracy where some spike lag is acceptable and smaller memory footprint and datapath width is a priority.

Table 4.3: Summary of ODE results with 16-bit fixed-point arithmetic compared to a binary64 floating-point reference (with the constants represented in 16 bits): spike lags (ms) of the 650th spike in the DC current test. Positive means lag, negative — lead. The test cases marked with a dash did not produce any spikes due to underflow in one of the internal calculations. Fixed, {RD/RN/SR} refer to fixed-point arithmetic with round down, round to nearest and stochastic rounding respectively.

Solver	Neuron type	binary32	fixed, RD	fixed, RN	fixed, SR (std.dev.)
RK2	RS	16.8	-21681.4	-	889.4 (58.82)
Midpoint	FS	-29.7	-2754.5	686.4	676.7 (30.67)
RK2	RS	6.9	-22786.2	-	363.3 (57.65)
Trapezoid	FS	-4.6	-2391.2	892.8	516.7 (27.92)

4.5 Round and saturate accelerator for SpiNNaker2

Here an accelerator for rounding and saturation is described and evaluated. This accelerator is included in the SpiNNaker2 prototype chip *JIB2* which is currently in design. Most of the material is reproduced from a preprint paper that has been published online [119].

4.5.1 Motivation

SpiNNaker2 will be based on an ARM Cortex-M4F processor, which does not have a capability of rounding a fixed-point number to a specified number of bits. There are three instructions with rounding available: `SMMLAR` — multiply two numbers, add a third number to the top 32-bits of the result and return the rounded 32 top bits; `SMMLSR` — the same as `SMMLAR`, but subtracts the third argument; `SMMULR` — multiply and return rounded 32 top bits of the result [62]. Rounding is done by adding 0x80000000 to the product, therefore the tie-breaking rule is round up [120]. While this would work well for s16.15 $\times$ u0.32 multiplications, it is limited in terms of other different mixed-format multiplications discussed in Chapter 3. To implement round to nearest and stochastic rounding would therefore require multiple instructions to perform Algorithms 3 and 5 on two registers. Furthermore, there is no mention as to whether these instructions perform saturation after rounding. While saturation instructions for 32-bit values with configurable saturation bit position and saturated addition are available on the M4F, saturating a 64-bit value from the multiplication would need to be done by comparison and because it is a value across two registers, multiple instructions would

be required to obtain a rounded and saturated value somewhere in the middle of a 64-bit value. Due to this, it is proposed to include a small memory mapped accelerator to perform rounding and saturation of 64- and 32-bit values, which reuses an already existent PRNG hardware unit on SpiNNaker2.

4.5.2 Specification

The rounding and saturation accelerator is a memory mapped unit, connected through an Advanced High-Performance Bus (AHB) bus — a set of memory addresses are allocated for different rounding routines and numerical formats, to which arguments are written and from which the rounded values are read out. For rounding multiplication results, it is useful to have a 64-bit to 32-bit number rounding, with configurable rounding bit position from 0 to 31. Given that the ARM M4F processor has 32-bit wide interfaces, two memory cycles are required for inputting 64-bit arguments through AHB. For other cases, for example rounding weight updates which might be 32 bits or 16 bits, 32-bit to 32-bit, 32-bit to 16-bit, and 16-bit to 16-bit round and saturate is supported. For this, one memory cycle is required for input, therefore, assuming one cycle for the main part of rounding, the accelerator will either have a 4 or 3 cycle delay for a write-round-read operation for 64/32-bit arguments respectively.

Both signed and unsigned numerical formats should be supported, given a wide range of use cases for both shown in this thesis. Furthermore, as the ARM M4F has binary32 floating-point hardware support, it might be beneficial to round binary32 to bfloat16 (16-bit single precision floating-point format with sign, 8 exponent and 7 significant bits). This format can be useful for representing synaptic weights for example, which can be operated on using the binary32 FPU by inputting bfloat16 weights into the top 16 bits of the FPU registers. Finally, given that an adder is required in stochastic rounding, we can also implement round to nearest, which can reuse the adder to add 0x1 shifted to the appropriate bit position.

4.5.3 Design

Figure 4.5 gives an architectural diagram for performing a similar stochastic rounding and saturation algorithm to Algorithm 4. Signals *signed arithmetic* and *round mode* are derived from the address supplied by the AHB bus, depending on which address was written by the processor.

The main mechanism is to pick the top 32 bits of the residual depending on the

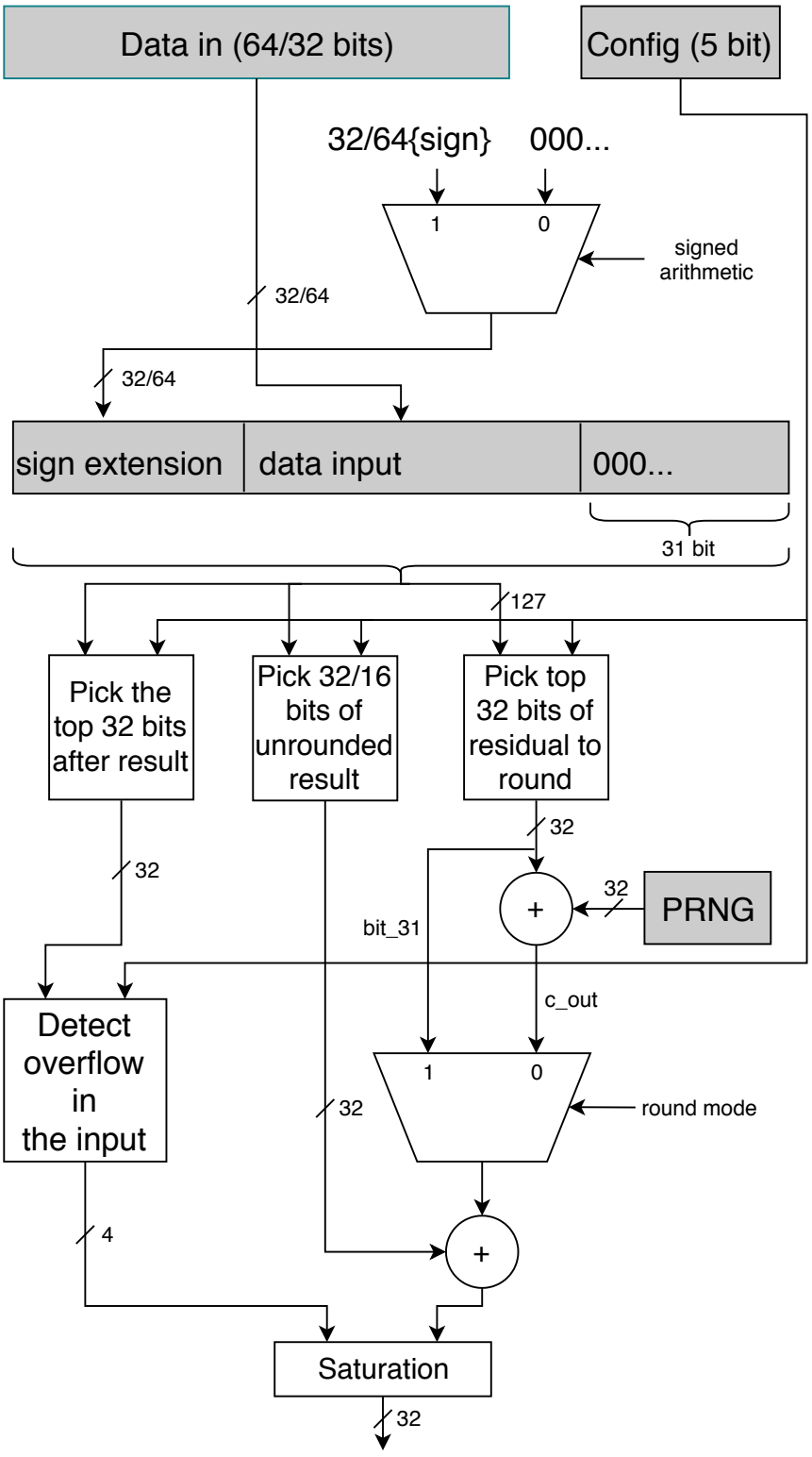

Figure 4.5: Architectural diagram for the rounding and saturation accelerator.

configuration register, which is set up beforehand and contains the number of bits to round — 0 to 31 (0 means round 1 bit, 31 means round 32 bits). Then, the 32 bits after these residual bits are also isolated which is an unrounded result at this point. The minimum number of bits to round is 1, therefore the data input is extended to the right by 31 bits to support Verilog's *base minus 32* bit slicing functionality. For the same reason the input data is extended by 32 bits to the left, to check for overflow by bit wise logic on the top 32 bits outside the main result that will be rounded and extracted.

The register named PRNG is filled in by reading the next 32-bit random number from the existent SpiNNaker2 PRNG. The pseudo-random number is added to the residual and the carry bit *c_out* is captured from that. Then, depending on the *round mode*, either the top bit (in case of round to nearest) or the *c_out* (in case of stochastic rounding) is added to the unrounded result which performs round up if it is 1 and round down if it is 0. Finally the rounded result and the overflow bits are used to saturate the result if required.

4.5.4 Evaluation

The main logical path of the accelerator contains two adders — one 32-bit wide for rounding and one 8-, 16- or 32-bit wide for the stochastic rounding part when a random number is added to the residual. The architectural diagram in Figure 4.5 demonstrates a 32-bit version, but it is worth evaluating the three versions as there is some evidence that not all of the 32 bits are needed in SR, as shown in Section 4.4.2.3. All of the logic, except some saturation checks are performed in a single cycle. Saturation logic contains basic checks of the overflow flags depending on the address input from the AHB bus and in the current implementation it is done on the AHB output cycle.

A synthesis study of the presented design was executed using the makeChip hosted design service platform [121] for the GLOBALFOUNDRIES 22FDX technology [122] for which the SpiNNaker2 chip is being developed. An ultra-low voltage *8t-CNRX* standard-cell library with multiple voltage threshold options is used for implementation. The standard cells use the adaptive body biasing (ABB) technique for post-silicon adaptation of transistor threshold voltage [123, 124]. Namely, two main categories of cells are used: Low-Voltage Threshold (LVT) and Super-Low-Voltage Threshold (SLVT) cells — the former with the larger propagation delay but significantly less leakage than the latter, much faster, cells. A nominal supply voltage of 0.50V is considered for low power operation. Due to manufacturing variations, synthesis is performed in a worst case speed condition at 0.45V and $-40\,^{\circ}\mathrm{C}$. Three versions of the accelerator are

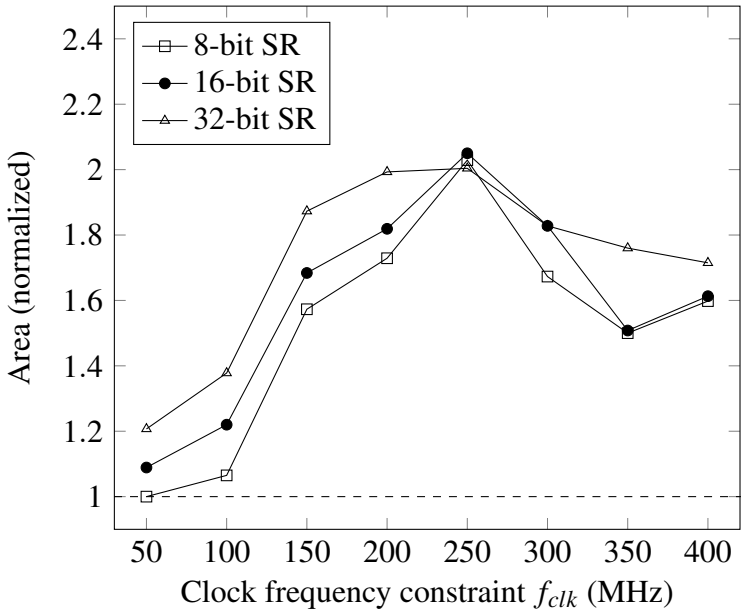

Figure 4.6: Circuit area of the accelerator when synthesised with different clock constraints. Three accelerator versions are shown with 8-, 16- and 32-bit stochastic rounding.

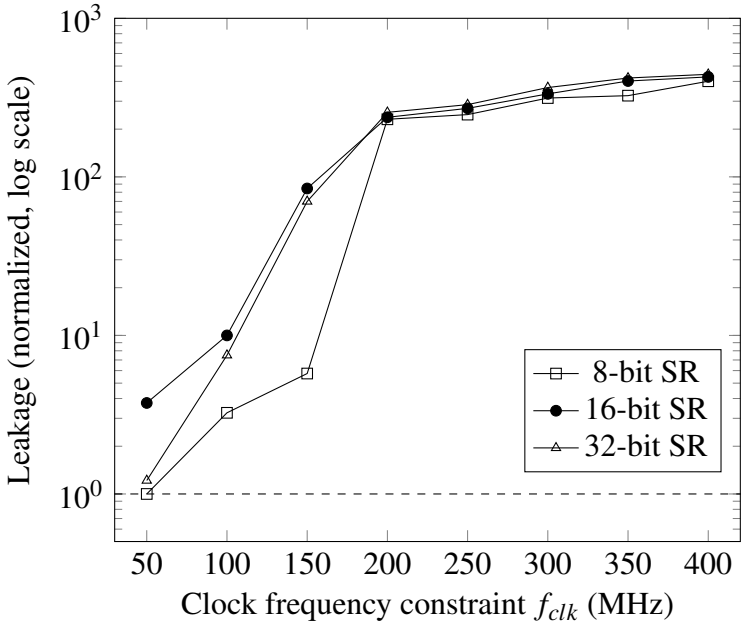

Figure 4.7: Leakage power of the accelerator when synthesised with different clock constraints. Three accelerator versions are shown with 8-, 16- and 32-bit stochastic rounding.

synthesised varying the clock frequency constraint

$$f_{clk} = \{50, 100, 150, 200, 250, 300, 350, 400\}\,\text{MHz},$$

and leakage power as well as area are measured.

Figure 4.6 shows comparison of the three accelerators for different clock constraints and Figure 4.7 shows leakage power. From this data it can be seen that at low clock frequencies, the adder's width can save some area and leakage, but at higher frequencies other costs dominate and the savings are not that evident anymore. Especially for leakage; the leakage of the circuit apart from the adder dominates the total and changing to a smaller adder does not produce significant changes.

Figure 4.8 shows an accelerator highlighted in a layout of a single PE. The area of the accelerator is estimated at $1004\,\mu\text{m}^2$.

4.6 Discussion and further work

This chapter addressed the numerical accuracy issues of ODE solvers in fixed-point arithmetic, solving a well known neuron model in fixed- and floating-point arithmetics. Before, in Chapter 3, it was identified that the constants in the Izhikevich neuron model should be specified explicitly by using the nearest representable number as the GCC fixed-point implementation does round down in decimal to fixed-point conversion by default (this was also independently noticed by another study [15] but authors there chose to increase precision of the numerical format of the constants instead of rounding the constants to the nearest representable value as in this work). Next, all of the constants smaller than 1 were stored in the u0.32 format instead of keeping everything in s16.15, to maximize the accuracy. This required the routines for mixed-precision arithmetic operations to be developed. This has not been done in any of the previous studies exploring numerical accuracy of this neuron model on SpiNNaker [14, 15]. Jin et al. [76] went in this direction using 16-bit formats and two different scaling factors with mixed-precision arithmetic, but no comprehensive analysis of the 32-bit numerical formats and rounding was performed. Jin et al. [76] focused on 16-bit for efficiency, considering the Euler method for ODE solver at instruction level. Therefore, this work should complement work by Jin et al. [76] more in terms of numerical accuracy.

Chapter 3 demonstrated very accurate results on the default SpiNNaker ODE solver and one configuration of the neuron, however, as identified in this chapter, changing

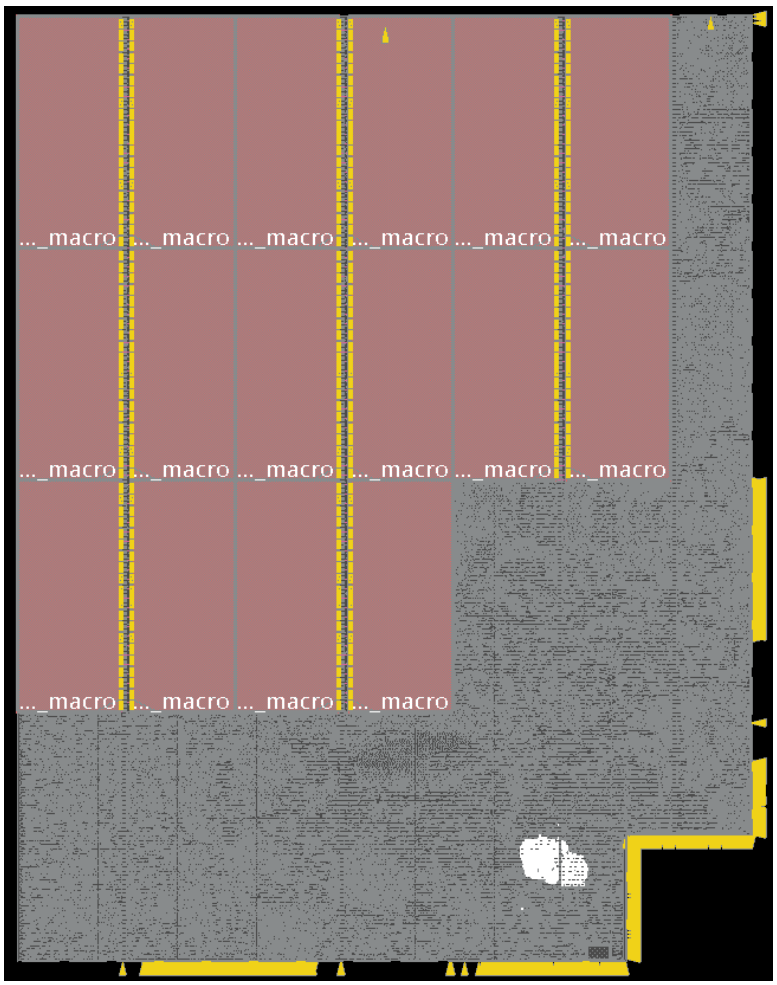

Figure 4.8: Layout of a PE after place and route. Cells marked ...*macro* bundled at north-west corner are local memory. The rest of the cells at the south-east corner belong to an ARM M4F based PE. Out of that, cells highlighted in white belong to the rounding accelerator (without the PRNG). Picture provided by Stefan Scholze.

the neuron parameters or the ODE solver algorithm can result in very different numerical errors. It was then identified that fixed-point multiplications are the main remaining source of arithmetic error and explored different rounding schemes. Stochastic rounding on multiplication results was subsequently shown to produce substantial accuracy improvements across four ODE solvers and two neuron types. Fixed-point with stochastic rounding was shown to perform better than fixed-point RN and binary32 floating-point format, and the mean behaviour is very close to binary64 floating-point ODE solvers in terms of spike times. It was also shown that simple PRNGs will often be good enough for SR to perform well. The minimum number of bits required in the random number in SR was found to be 6 across four different ODE solvers and two neuron types tested. In these cases, using more bits is unnecessary, and using fewer will cause the neuron timing to lead compared to the reference.

In Section 4.4.2.4 16-bit arithmetic results were shown to have advantages with SR. Although the absolute performance is quite poor (most likely due to overflows and underflows), 16-bit results with SR perform better than 16-bit RD (which largely results in spike time lead) and RN (which produces no spikes). Further work using scaled interim variables to ameliorate these issues is likely to provide further gains.

Furthermore, in the paper that was published with this work [33] a concept of *dither* was applied where the input current of the neuron has added noise. It was shown that this has also reduced the spike lags in the majority of cases explored in this chapter; in the case where it works as well as SR this can be a computationally cheaper alternative to stochastic rounding (no random number generation on each multiplier in the ODE solver).

While an extensive investigation of the speed of ODE solvers was out of the scope of this work, measurements in the test bench show that SR performance is mainly dictated by the PRNG performance. Preliminary numbers from RK2 Midpoint ODE solver performance benchmarks show an overhead of approximately 30 % when RN is replaced with SR. Furthermore, SR is $\sim 2.6\times$ faster than software-emulated binary32 floating-point and $\sim 4.2\times$ faster than binary64 arithmetic in running a single ODE integration step. In terms of SpiNNaker2, while it will have a binary32 FPU, the results in this chapter demonstrate that fixed-point with SR can be more accurate, and in most of the cases as accurate as binary64 floating-point arithmetic on average, and should be considered instead of simply choosing binary32 in a given application (especially noticing how much error binary32 arithmetic can accumulate in some tests visible in Figure 4.2).

For future work, it would be useful to perform mathematical analysis to understand better why SR is causing fewer rounding errors in ODEs. For performance, explore how many random numbers are needed in each ODE integration step — do we require all multipliers to use a separate random number, or is it enough to use one per integration step? Also, for SpiNNaker neuromorphic applications, it would be useful to build fast arithmetic libraries with SR and measure their overhead compared to default fixed-point arithmetic and various classical ways of rounding. Also, investigating the application of SR in neuron models other than Izhikevich's, and ways to solve them, the first example being LIF with current synapses which has a closed-form solution. Another direction is to investigate fixed-point arithmetic with SR in solving partial differential equations (PDEs) and other iterative algorithms (for example in linear algebra). Finally, it would be beneficial to investigate SR in reduced precision floating-point: binary16 and bfloat16 numerical formats which are becoming increasingly common in the machine learning and numerical algorithms communities, for large-scale projects such as climate simulations using PDEs [27, 125].

Given all of these results, there is some indication (although more research is required) that any reduced precision computing platform solving ODEs and running other similar algorithms involving long summation will benefit from using stochastic rounding. This has been showed here experimentally in harmonic sum and ODE solvers. It is also useful to point out that different arithmetics have different places where SR could be applied. For example, unlike fixed-point adders, floating-point adders and subtracters need to round when exponents do not match, and SR could be applied there after the addition has taken place (which would require preserving the bottom bits after matching the exponents). Similarly in neural learning, where the computed changes to a weight are often smaller than the lowest representable value of that weight. The application of stochastic rounding in solving ODEs has not yet been investigated on any digital arithmetic, and these are the first results demonstrating substantial numerical error reduction in fixed-point arithmetic.

4.7 Conclusion

Improvements to the numerical accuracy with stochastic rounding, of certain applications demonstrated by machine learning community, indicated that it could be applied more widely, in this case on SpiNNaker applications. The most well known example of SR is given by Gupta et al. [39], demonstrating a neural network training and

performing a digit recognition task in fixed-point 16-bit formats, performing well and competing with binary32 floating-point equivalent application, only when stochastic rounding is applied on the outputs from the multiply-accumulate units. The goal of this chapter was to identify areas where SR could be useful on SpiNNaker, itself working mainly in fixed-point arithmetic, and to compare the numerical errors with the state of the art SpiNNaker software [14, 53]. SR libraries were developed, shown to work well in harmonic series run in fixed-point and were shown to reduce error in the Izhikevich neuron model — one of the most used neuron models that can be configured to change the spiking patterns.

While results were already improved by correct rounding of constants and mixed-precision multiplications addressed in the previous chapter, SR provides robustness across different neuron spiking types and different ODE solvers, which is not the case with RN, even with the improvements from the previous chapter. The results were shown to be important for numerical accuracy — ODE solvers in 32-bit fixed-point arithmetic with SR have less neuron spiking lag than the equivalent floating-point solvers in most of the tested cases. Therefore, the main goal of the chapter was achieved with very positive results, even showing promise beyond SpiNNaker use cases to affect any low power computer solving ODEs in fixed-point arithmetic. Given these results, it was then straightforward to make a decision that SpiNNaker2 should include a small rounding unit to perform rounding faster than possible in software. An accelerator was designed and evaluated and will be included in the SpiNNaker2 chip, giving, for a small cost of chip area and leakage power, a very substantial improvement in certain numerical algorithms.

The next chapter of the thesis moves on from the neuron models and rounding, and addresses an accelerator for computing exponential function which is used in a wide array of neuron and plasticity models.

4.8 Acknowledgements

Most of the material on testing the ODE solvers in this chapter was developed in collaboration with Michael Hopkins [33]. The author thanks Michael for many interesting discussions about numerical errors and ODE solvers as well as for providing the initial code of the ODE solvers.

Chapter 5

Hardware e^x and $\log_e(x)$ Function Accelerator for SpiNNaker2

Exponential function is required in most parts of neural and plasticity models. On the current generation SpiNNaker machine, this function has been evaluated either using look-up tables stored in memory or by calling a software implementation which takes around hundred cycles to run. In this chapter a hardware exponential function for SpiNNaker2 in fixed- and floating-point arithmetic formats is developed and evaluated. A particular chosen algorithm permits the addition of a logarithm function with most of the logic shared between the two. Having both e^x and $\log_e$ can be beneficial for implementing the general exponentiation function, required for example in certain plasticity models. The accelerator is developed in a 22nm SpiNNaker2 environment and the fixed-point version was included in the recently manufactured SpiNNaker2 prototype chip (codenamed JIB1). The final version documented here, with improved accuracy and a floating-point interface, is currently planned to be included in the next SpiNNaker2 prototype chip (JIB2). This chapter shows the algorithms, numerical accuracy, chip area and energy utilization of the accelerator, as well as ways to use the accelerator in mixed-format precision with different controllable accuracies.

Some sections in this chapter are reproduced from the material that was published as part of the proceedings of the 25th IEEE symposium on computer arithmetic [126].

5.1 Introduction

One of the most common functions in SNNs is e^x, used to model exponentially decaying quantities. Most neuron models [14] and biological learning, STDP rules

[9, 127, 128], developed on SpiNNaker all use exponential decay. In ARM software it is hard to realise a very fast implementation of this function — it requires large LUTs (with values in higher precision than input/output) and the usual latency is 60–200 clock cycles, depending on the numerical format and accuracy required (see [129] for the performance analysis on x86 architectures and see [130] for ARM example). Due to this, it is important to consider dedicating silicon for the exponential function in a digital neuromorphic chip.

In the current SpiNNaker chip there is no hardware support for transcendental functions, including exponentials, so the models developed use pre-computed LUTs; this is described in detail in Section 3.3.5.1 (also see [131]). However, this approach has two limitations: a limited number of time constants and a limited input domain are available due to the size of on-chip memory, and in the case of a model that requires time constants to be dependent on some dynamic quantity, such as the voltage-dependent time constants in the intrinsic currents of the well known *Hodgkin-Huxley* neuron model and its variants [132], the number of look-up tables required for each possible value that the time constant can take would be too large to store in the local SpiNNaker memory. The memory requirements are further increased if the simulation time step is 0.1 ms, which is rarely used on SpiNNaker at the moment, but in all likelihood will be used on SpiNNaker2 as it will give more accuracy in all parts of simulation. In this scenario, the size of LUTs for the same amount of time of decay look-up will grow 10 times, if every possible entry is stored. For example, modelling a 16-bit exponential decay $e^{-\frac{\Delta t}{\tau_x}}$ for 1 second and all the values that Δt can take at 0.1 ms simulation time step will require 20 kB of memory space. Although note that these tables can always be made smaller with certain side effects, such as reduced numerical accuracy depending on how coarse/fine the table is or a requirement to add interpolation to approximate a function between the entries in the table — all of this can be done to optimize the table for a specific problem.

A software exponential is also available in the SpiNNaker software library, but with a latency of approximately 95 clock cycles, it would be a major limit to real-time synaptic plasticity processing, where a single pair of spikes takes approximately 30 cycles as reported by Knight and Furber [55]. When processing most learning rules, we usually need more than one exponential per spike-pair processing. Learning rules requiring 3 or more decay time constants have already started appearing in computational neuroscience literature and some were already tried on SpiNNaker: see for example voltage-dependent STDP [87] implemented on SpiNNaker [128], the BCPNN learning

rule [9, 131] and the neuromodulated STDP [11] (demonstrated in Chapter 3) learning rule.

Fixed-point data types s16.15, s0.31 and u0.32 are the main ones used in the SpiNNaker software stack for all neural and synapse models, and it is likely that in SpiNNaker2, for most of the performance sensitive code it will be preferred instead of the FPU available in ARM M4F, especially when noting that special DSP instructions [62] can be used on the fixed-point and not on the floating-point numbers. Although it is possible to find optimal fixed-point formats for accuracy in any given application, the choice was made to follow the fixed-point standard [34] and design the arithmetic unit with the two main signed fixed-point formats: s16.15 and s0.31. Furthermore, to maintain consistency of arithmetic capabilities available in ARM M4F, and in case a need for floating-point exponential and logarithm functions presents itself in the future, the floating-point (binary32) interface was also designed (for example, to model AdEx neuron model [42, 43], which has not yet been done on SpiNNaker).

The most recent SpiNNaker2 chip prototype has a fully pipelined exponential unit built in [59]. However, the implementation is limited to the standard fixed-point format s16.15 which in most cases, where an exponential function is used to model exponential decay, has 16 unused integer bits in the outputs, as discussed in Section 3.2. Other formats such as s0.31 could be useful to improve accuracy of models. For example Yan et al. [21] did have to extend the exponential function in s16.15 with software floating-point wrapper. Additionally, the design in [59] uses the identity $e^{a+b+c} = e^a \times e^b \times e^c$ to parallelize computation and therefore can be quite large in circuit area due to the required multipliers.

Here a different implementation based on iterative shift-add algorithms is demonstrated; these type of algorithms are usually considered to be slower due to serial dependencies of the underlying equations but do not require multiplication, which can reduce area of the circuit. The internal iterative part of the algorithm is performed in fixed-point carry-save redundant number representation to reduce the critical path delay (although a signed-digit [133] representation could also be used as shown by Muller [16]). The accelerator was designed so that the two different fixed-point formats can be mixed to gain more accuracy on some arguments. Furthermore, a useful intrinsic property of the iterative algorithms is that after just a few iterations they already contain the approximate output with correct bits at the top and some error at the bottom. It was chosen to use this property to provide programmable accuracy control, following the principles of *approximate computing* [28, 134, 135] (in this case approximation

comes not from the errors in the circuit as is most common, but from not running the maximum number of iterations of the algorithm) in order to add options for modellers to sacrifice some accuracy in exponential decay and to tradeoff accuracy for speed and energy where required. This property will provide a platform for experimenting with concepts arising from the ongoing discussion about the number of bits required for representing weights in STDP [136].

Most of the algorithms for performing elementary functions are categorised into two types: *polynomial approximations* or *convergence algorithms* [16, 31]. For example, the COordinate Rotation DIgital Computer (CORDIC) algorithm is a well known shift-and-add (convergence) algorithm that can be used to evaluate a wide array of functions [137]. However, as Bajard et al. [138] point out, CORDIC cannot easily be implemented in a redundant number representation to avoid propagation of carries in the iterative part of the design. For this work, a well-known convergence algorithm presented by Muller [16, Ch. 8] was chosen; this algorithm provides exponential and natural logarithm functions with overlapping hardware components (note that having both of these functions we can also derive the equation for evaluating the power function as discussed in Section 5.4.2). Also see [139] for a review of the challenges that this sequential algorithm presents — unfortunately this work did not present physical implementation results. The implementation documented in this thesis is done in radix-2, which means that one iteration of the algorithm calculates one bit of the function's output.

The unit is included in the prototype neuromorphic chip as an AHB slave that can be driven from the ARM core by writing and reading the set of specific memory locations, similarly to the implementation by Partzsch et al. [59]. Design synthesis studies are executed on the makeChip hosted design service platform [121] for the GLOBALFOUNDRIES 22FDX technology [122].

The rest of the chapter is structured as follows.

- The main algorithm for exponential (e^x, further called exp) and natural logarithm ($\log_e(x)$, further called log or $\log(x)$ without the base) is analysed in detail in Section 5.2.
- The architecture of the unit and implementation is documented in Section 5.3.
- Testing and accuracy analysis is discussed in Section 5.4.1.
- A synthesis study is discussed in Section 5.4.3, where the details of leakage

power, and area for different versions of the accelerator at worst case timing conditions are provided.

- Lastly, a place and route study is documented in Section 5.4.5 where more accurate power measurements are reported when running software test cases at worst case power conditions, and the proposed design is compared to similar designs in the literature.

5.2 Algorithms

The iterative *shift-and-add* algorithm presented in [16, Ch. 8] is used for this accelerator. The following is a brief description of the algorithms, first in general, and then versions for the carry-save representation. In Sections 5.2.1 – 5.2.3 the algorithms are reproduced from [16] and extended beyond the material in the book in Section 5.2.3.

5.2.1 Main iterative algorithm

The main algorithm consists of inputs t and N and the sequences t_n and d_n run for N steps and defined as

$$\begin{aligned} t_0 &= 0, \\ t_{n+1} &= t_n + d_n \log(1+2^{-n}), \end{aligned} \tag{5.1}$$

with

$$d_n = \begin{cases} 1 & \text{if } t_n + \log(1+2^{-n}) \leq t, \\ 0 & \text{otherwise,} \end{cases} \tag{5.2}$$

and that satisfy

$$\lim_{n\to\infty} t_n = t = \sum_{n=0}^{\infty} d_n \log(1+2^{-n}). \tag{5.3}$$

The sequence $\log(1+2^{-n})$ is a decreasing sequence (0.693..., 0.405..., 0.223..., ...) therefore on each step a new smaller value is added to t_n (if $t_n + \log(1+2^{-n})$ is still below t) or not (if the sum would become larger than t).

Now a sequence E_n is defined such that at any step n of the algorithm

$$E_n = \exp(t_n). \tag{5.4}$$

Since $t_0 = 0$, E_0 is initialised as 1. When $d_n = 1$, $\log(1+2^{-n})$ is added to t_n. As a result,

to keep E_n consistent with the definition, $E_{n+1} = \exp(t_{n+1}) = \exp(t_n + \log(1+2^{-n})) = \exp(t_n)\exp(\log(1+2^{-n})) = E_n\exp(\log(1+2^{-n})) = E_n(1+2^{-n}) = E_n + E_n 2^{-n}$. Then, to generalise for $d_n = \{0,1\}$ choice on each step we can write

$$E_{n+1} = E_n + d_n E_n 2^{-n}. \tag{5.5}$$

Note that this calculation requires only an adder and a shifter, since the multiplication is by a power of 2. Then,

$$\begin{aligned} \lim_{n\to\infty} t_n &= t, \\ \lim_{n\to\infty} E_n &= \exp(t). \end{aligned} \tag{5.6}$$

The algorithm convergence domain is

$$t \in [0, \sum_{n=0}^{\infty} \log(1+2^{-n}) \approx 1.56202...], \tag{5.7}$$

and the relative error of approximating e^t by stopping iterating at step n is shown by Muller [16] to be $|\frac{e^t - E_n}{e^t}| \leq 2^{-n+1}$ (which in terms of bits, means that there will be $n-1$ or more correct significant bits — 2ulps of maximum error).

Another additional feature of this algorithm is that it can easily be transformed to compute the natural logarithm function. Given an input x we want to compute $\log(x) = t$. What we are interested in now is that the sequence t_n in (5.1) is converging to the answer. However, in this case t is not known, as that is what we are looking for, and thus cannot choose d_n in (5.2). But what we do know is $x = \exp(t)$ (it is an input argument) and since E_n in (5.5) was built in such a way that at any step n, $E_n = \exp(t_n)$, (5.2) is equivalent to

$$d_n = \begin{cases} 1 & \text{if } E_n(1+2^{-n}) \leq x, \\ 0 & \text{otherwise.} \end{cases} \tag{5.8}$$

Now by using (5.8) we will have the same choice of d_n on each step as in the algorithm for exponential, but since we do not need to know t in the choice of d_n, we can in fact compute it as t_n converges to it. Additionally, as Equations 5.1 and 5.5 do not change when converting exponential algorithm to the natural logarithm algorithm, *we can re-use the same hardware resources when implementing both algorithms*. The algorithm

convergence domain when it is used for natural logarithm as shown by Muller [16] is

$$x \in [1, \prod_{n=0}^{\infty}(1+2^{-n}) \approx 4.768]. \tag{5.9}$$

This part of the algorithm was presented because it is easy to understand how it works in this form. The next two sections will show the faster versions of the algorithms that are not straightforward without understanding the basic principles shown in this section.

It is worth noting that these algorithms are not implemented straight into hardware as written in this section as it would result in a slow critical path, especially on the d_n choice step. Noting that these algorithms are multiple decades old, it is possible that modern logic synthesizers will spot that the algorithm can be done in carry-save number representation, but it will almost certainly not find the improvements deeply hidden in the mathematics of the algorithm and the number system that Muller [16] demonstrates for the choice of d_n and which will be explained next.

5.2.2 Algorithm for exponential in carry-save representation

To speed-up the algorithm in Section 5.2.1, the two additions in Equations 5.1 and 5.5 are done using carry-save adders (Chapter 2). Then, when the last iteration is computed and we need to obtain the final answer, the value E_n or t_n is converted, depending on whether it is the exponential or logarithm function, from carry-save to a standard non-redundant representation using a single carry-propagate adder. However, using carry-save adders would not bring much advantage if, in the comparison operation in Equations 5.2 and 5.8, we have to check the full word length value — it would result in large comparator circuits and become a major bottleneck on every iteration. Because of this, Muller [16] presents a restructured algorithm which requires a check of only 4 bits to do the comparison to find out d_n on each step.

The basic iteration in (5.1) is changed to the following, which is simply reversing convergence of t_n so that L_n converges from x to 0 and moving d_n inside the logarithm:

$$L_{n+1} = L_n - \log(1 + d_n 2^{-n}), \tag{5.10}$$

with $L_0 = x$, and

$$d_n = \begin{cases} 1 & \text{if } L_n \geq \log(1+2^{-n}), \\ 0 & \text{otherwise.} \end{cases} \tag{5.11}$$

Series E_n remains the same:

$$E_{n+1} = E_n + d_n E_n 2^{-n}. \tag{5.12}$$

Then, if $L_0 = x$ is in the convergence domain $t = x \in [0, 1.56202...]$, this gives

$$\begin{aligned} \lim_{n\to\infty} L_n &= 0, \\ \lim_{n\to\infty} E_n &= E_0 e^{L_0}. \end{aligned} \tag{5.13}$$

At this step, Muller [16] adds another new change in the algorithm that brings redundancy but allows the comparison step to be performed by examining fewer digits (less than a full length of L_n). Instead of only allowing values of $d_n \in \{0,1\}$, it is also allowed to have $d_n = -1$. Note that $\log(1-2^{-0})$ is not defined, so the evaluation starts at iteration $n = 1$. Now the algorithm converges if

$$\begin{aligned} L_1 \in [&\sum_{n=1}^{\infty} \log(1-2^{-n}) \approx -1.242..., \\ &\sum_{n=1}^{\infty} \log(1+2^{-n}) \approx 0.869...]. \end{aligned} \tag{5.14}$$

Define

$$L_n^* = 2^n L_n, \textit{ truncated after the first fractional digit}.$$

Here L_n is in carry-save representation with digits $0\,(00_2), 1\,(10_2 \text{ or } 01_2)$ and $2\,(11_2)$. Now the choice of d_n in (5.2) is transformed to

$$d_n = \begin{cases} -1 \text{ if } L_n^* \le -3/2, \\ 0 \text{ if } -1 \le L_n^* \le -1/2, \\ 1 \text{ if } L_n^* \ge 0. \end{cases} \tag{5.15}$$

Muller [16] showed that this choice of d_n can be implemented by a look-up table indexed by 1 fractional and 3 integer bits of L_n^* (when converted to non-redundant representation) and proved that it assures convergence of the algorithm, therefore speeding up the comparison step and allowing the gains of using carry-save adders to be realised.

The algorithms presented so far are reproduced from Muller [16, Ch. 8] to give a full account before the contribution of the next section. The next section starts with the algorithm for logarithm and extends it beyond what was shown by Muller [16].

5.2.3 Extending the algorithm for logarithm in carry-save representation

A similar modification of the algorithm for $\log(x)$ in carry-save number representation also has to be found in order for the comparison step, where the choice of d_n is made, to be faster. Such a modification is presented for signed-digit number representation by Muller [16] but not for carry-save implementation. The steps to find it are very similar to those used in the signed-digit representation of the algorithm and allowed to find a working choice of d_n for logarithm function.

We use the same iteration for E_n and L_n as before and initialise $E_1 = x$, $L_1 = 0$. As pointed out by Muller [16] we notice that $E_n \times e^{L_n}$ is constant and if we find such a sequence of d_n values that would make E_n converge to 1, after n steps we would get that $E_n \times e^{L_n} = e^{L_n}$. Then we will have $L_n \to \log(x)$. We need to find a sequence of terms d that will give us this convergence in carry-save number representation, and once again, without using large comparator circuits. Define

$$\lambda_n = E_n - 1, \tag{5.16}$$

and

$$\lambda_n^* = 2^n \lambda_n, \textit{ truncated after the first fractional digit}. \tag{5.17}$$

From this point on, the results are novel, since Muller [16] did not present a fast choice of d_n in carry-save. Using the property $0 \leq 2^n\lambda_n - \lambda_n^* \leq 1$ and the properties obtained from the Robertson diagram shown in [16, Sec. 8.3.2], it was found that for $n \geq 2$, the choice of d_n is

$$d_n = \begin{cases} -1 \text{ if } \lambda_n^* \geq 1/2, \\ 0 \text{ if } -1/2 \leq \lambda_n^* \leq 0, \\ 1 \text{ if } \lambda_n^* \leq -1. \end{cases} \tag{5.18}$$

Similarly to signed-digit number representation which is analysed in the book, this choice of d_n does not work at the first step $n = 1$ of the algorithm. However, noticing that at *the first step E_1 is in non-redundant representation, and since $\lambda_1 = E_1 - 1$ only changes bits in the integer part of the fixed-point value of E_1, we can consider the fractional part of λ_1 to be in non-redundant representation*, even though the integer part might have both intermediate sums and carries set. This allows us to use the

property $0 \leq 2\lambda_1 - \lambda_1^* \leq \frac{1}{2}$, which then allows to find that the choice of d_1 is

$$d_1 = \begin{cases} -1 \text{ if } \lambda_1^* \geq 1, \\ 0 \text{ if } -1/2 \leq \lambda_1^* \leq 1/2, \\ 1 \text{ if } \lambda_1^* \leq -1, \end{cases} \tag{5.19}$$

which is only different from (5.18) in that when $\lambda_n^* = 1/2$, we choose $d_n = 0$ instead of $d_n = -1$, and therefore results in a simple special case for step one of the algorithm. The algorithm converges in the domain, as shown by Muller [16],

$$E_1 \in [0.4194..., 3.4627...]. \tag{5.20}$$

Finally, with the basic iteration Equations 5.10 and 5.12, and the rules to choose d_n on each step of the algorithm in carry save representation using (5.15), (5.18), and (5.19), we have a unified iterative algorithm for exponential and natural logarithm in carry-save number representation. It was verified that this algorithm works with a model implemented in C, which is discussed in the next section.

5.2.4 Simulating the algorithm in C

The algorithm presented above can easily be built in C by emulating the carry-save number representation and all the required operations on such numbers. It is useful to obtain a bit-level equivalent model to the hardware unit that will be built later, therefore this is the main focus of this step. The resultant code for the model of this algorithm is provided in Appendix C.

Some useful checks can be run to confirm correctness of the implementation of this algorithm. Table 5.1 shows how the algorithm progresses on each step for the exponential function with $x = 0.7$, using a fixed-point representation with 35 fractional bits and iterating for 32 steps. It can be seen that L_n is approaching 0 while E_n is converging to the answer, at $n = 32$ returning an error of 0.000000000513... compared with `math.h` exponential function. Table 5.2 shows a similar test for the logarithm with $x = 0.65$. This time, E_n is converging to 1 while L_n is converging to the answer, returning at $n = 32$ an error of 0.0000000000102... compared with the `math.h` logarithm function in double precision.

Another useful test to do using the model is to check the convergence domains of the functions shown previously in (5.14) and (5.20) for exp and log respectively. Such

Table 5.1: Progress of the iterative shift-add algorithm when $\exp(0.7)$ is called in fixed-point format with 35 fractional bits.

n	E_n	L_n	d_n
1	1.00000000000000000000	0.70000000001164153218	1
2	1.50000000000000000000	0.29453489190200343728	1
3	1.87500000000000000000	0.07139134057797491550	1
4	2.10937500000000000000	-0.04639169509755447507	0
5	2.10937500000000000000	-0.04639169509755447507	-1
6	2.04345703125000000000	-0.01464299680083058774	-1
7	2.01152801516582258046	0.00110536016291007400	1
8	2.02724307775497436523	-0.00667678029276430607	-1
9	2.01932415951159782708	-0.00276288099121302366	-1
10	2.01538016705308109522	-0.00080784616875462234	0
11	2.01538016705308109522	-0.00080784616875462234	-1
12	2.01439609474618919194	-0.00031944568036124110	-1
13	2.01390429885941557586	-0.00007527525303885341	0
14	2.01390429885941557586	-0.00007527525303885341	-1
15	2.01378137993742711842	-0.00001423823414370418	0
16	2.01378137993742711842	-0.00001423823414370418	0
17	2.01378137993742711842	-0.00001423823414370418	-1
18	2.01376601602532900870	-0.00000660881050862372	-1
19	2.01375833415659144521	-0.00000279411324299872	-1
20	2.01375449323677457869	-0.00000088676461018622	0
21	2.01375449323677457869	-0.00000088676461018622	-1
22	2.01375353304320015013	-0.00000040992745198309	-1
23	2.01375305294641293585	-0.00000017150887288153	-1
24	2.01375281292712315917	-0.00000005229958333075	0
25	2.01375281292712315917	-0.00000005229958333075	-1
26	2.01375275294412858784	-0.00000002249726094306	-1
27	2.01375272296718321741	-0.00000000759609974921	-1
28	2.01375270800781436265	-0.00000000014551915228	0
29	2.01375270800781436265	-0.00000000014551915228	0
30	2.01375270800781436265	-0.00000000014551915228	0
31	2.01375270800781436265	-0.00000000014551915228	0
32	2.01375270800781436265	-0.00000000014551915228	0

Table 5.2: Progress of the iterative shift-add algorithm when $\log(0.65)$ is calculated in fixed-point format with 35 fractional bits.

n	E_n	L_n	d_n
1	0.64999999999417923391	0.00000000000000000000	1
2	0.97499999997671693563	-0.40546510810963809490	0
3	0.97499999997671693563	-0.40546510810963809490	0
4	0.97499999997671693563	-0.40546510810963809490	0
5	0.97499999997671693563	-0.40546510810963809490	1
6	1.00546874996507540345	-0.43623676677816547453	0
7	1.00546874996507540345	-0.43623676677816547453	-1
8	0.99761352536734193563	-0.42839358933269977570	1
9	1.00151045317761600018	-0.43229222975787706673	-1
10	0.99955437809694558382	-0.43033719493541866541	1
11	1.00053050537826493382	-0.43131328091840259731	-1
12	1.00004196513327769935	-0.43082488043000921607	0
13	1.00004196513327769935	-0.43082488043000921607	0
14	1.00004196513327769935	-0.43082488043000921607	-1
15	0.99998092744499444962	-0.43076384341111406684	1
16	1.00001144441193901002	-0.43079436055268160999	-1
17	0.99999618547735735774	-0.43077910164720378816	1
18	1.00000381481368094683	-0.43078673104173503816	-1
19	1.00000000014551915228	-0.43078291634446941316	0
20	1.00000000014551915228	-0.43078291634446941316	0
21	1.00000000014551915228	-0.43078291634446941316	0
22	1.00000000014551915228	-0.43078291634446941316	0
23	1.00000000014551915228	-0.43078291634446941316	0
24	1.00000000014551915228	-0.43078291634446941316	0
25	1.00000000014551915228	-0.43078291634446941316	0
26	1.00000000014551915228	-0.43078291634446941316	0
27	1.00000000014551915228	-0.43078291634446941316	0
28	1.00000000014551915228	-0.43078291634446941316	0
29	1.00000000014551915228	-0.43078291634446941316	0
30	1.00000000014551915228	-0.43078291634446941316	0
31	1.00000000014551915228	-0.43078291634446941316	0
32	1.00000000014551915228	-0.43078291634446941316	-1

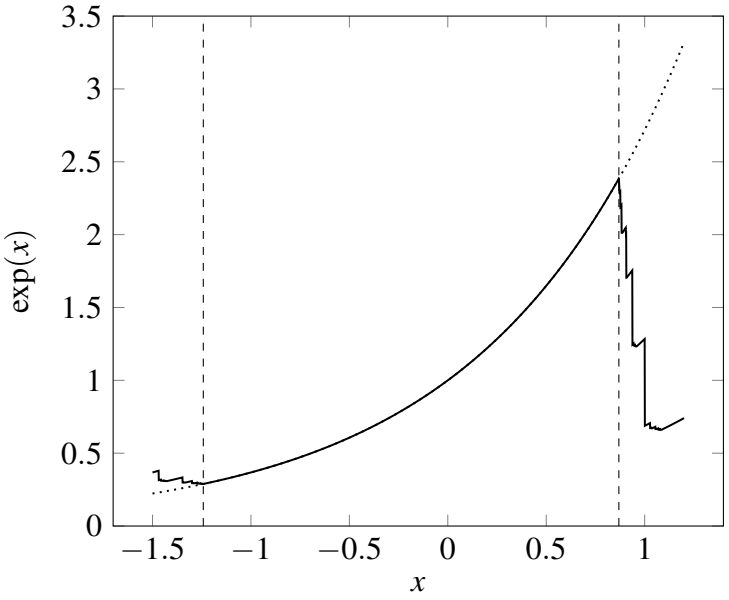

Figure 5.1: Sweep over the exponential function for $x \in [-1.5, 1.2]$ using the C model of the iterative algorithm presented above (solid line) and the `math.h` binary64 exponential function (dotted). Vertical dashed lines mark the convergence domain of the iterative exponential function algorithm shown in (5.14).

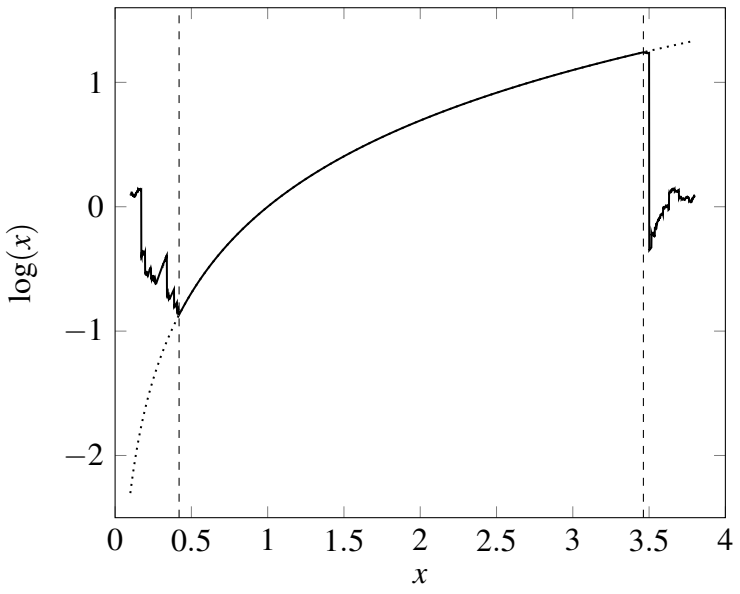

Figure 5.2: Sweep over the logarithm function for $x \in [0.1, 3.8]$ using the C model of the iterative algorithm presented above (solid line) and `math.h` binary64 logarithm function (dotted). Vertical dashed lines mark the convergence domain of the iterative logarithm function algorithm shown in (5.20).

tests are demonstrated for exponential and logarithm in Figures 5.1 and 5.2 which allowed to confirm that the convergence domains dictated by the algorithms are in fact correct in the model — in the figures the model is visibly computing wrong answers outside the domains compared to the reference function and correct answers inside the domains.

5.2.5 Range reduction and reconstruction

The algorithm presented in Section 5.2.1 converges only when x is in a limited range, which is inadequate for most of the use cases of exponential and logarithm functions. To provide full-range functions for formats s16.15, s0.31, and binary32 we must first perform *range reduction*, where x is transformed to some value inside the convergence domain of the algorithm, then the function evaluated, and using the answer and the information from the range reduction stage, *range reconstruction* is performed to get the the final answer.

5.2.5.1 Exponential

If x is in the range shown in Table 5.3, find x' and k such that x' is in the convergence range shown in (5.14) and expressed as

$$x' = x - k \times \log(2). \tag{5.21}$$

For the fixed-point exponential function, choosing the following works:

$$k = \lfloor x \times \frac{23}{16} \rfloor. \tag{5.22}$$

Note that $\frac{23}{16}$ is around 0.4% smaller than $\frac{1}{\log(2)}$. This gives us, when considering the range of possible arguments in s16.15 format (including ranges of mixed formats), $x' \in [-0.0751..., 0.73069...]$. For a wider range of inputs of exponential function in binary32 floating-point, this choice of k does not work as it goes out of range due to inaccuracy of $23/16$. Therefore, for floating-point exponential, a more accurate approximation of $k = \lfloor x \times \frac{369}{256} \rfloor$ was found to produce $x' \in [-0.0917..., 0.771...]$. The multiplication $k \times \log(2)$ has to be done in higher precision than the internal precision of the iterative algorithm (which is s3.35 and is further discussed in Section 5.3.1). Given that the maximum k for s16.15 input range is 15, 40 bits were chosen for this multiplication, with 5 guard bits to assure that the top 35 fractional bits of the result

Table 5.3: Approximate minimum and maximum ranges of values of exp with different 32 bit 2's complement formats. $*$ — saturates to 0x0 below this input value; † — saturates to 0x7FFFFFFF above this input value. The binary32 function either saturates to 0 on underflow or infinity on overflow.

I/O formats	exp input range	exp output range
s16.15/s16.15	$-10.397...^{*}$ to $11.09...^{\dagger}$	0.00003... to 65534.5...
s0.31/s0.31	-1 to $(-2^{-31})^{\dagger}$	0.367... to 0.99...
s16.15/s0.31	$-21.487...^{*}$ to $(-2^{-15})^{\dagger}$	$\sim 2^{-31}$ to 0.99...
s0.31/s16.15	-1 to $1-2^{-31}$	0.367... to $\sim e$
binary32/binary32	$-103.278...$ to 88.722...	$\sim 2^{-149}(sub.)$ to $\sim 3.403\times 10^{38}$

are error free. The constant $\log(2)$ is held as a negative value, therefore (5.21) is an addition and since x' is a range reduced result going into the carry-save representation used in the iterative algorithm, x can be assigned to the intermediate sums and $-k\times\log(2)$ to the intermediate carries, in effect obtaining x' without actually performing the addition.[1]

The range reconstruction is done as follows:

$$\exp(x)=\exp(x'+k\times\log(2))=2^k\times\exp(x'). \tag{5.23}$$

Because x' is in the range of the convergence domain shown in (5.14), $\exp(x')$ can be calculated using the iterative algorithm and then the final result in the full range just by shifting $\exp(x')$ by k places. In total, range reduction for exponential requires only multipliers by constant ((5.21) and (5.22)). Note that for evaluating (5.23) for fixed-point outputs, a shifter with $k\in[-31,15]$ will be needed (whereas floating-point requires the exponent to be set to k plus or minus any shifting steps required to normalize $\exp(x')$, known by utilizing a Count Leading Zeros (CLZ) module).

5.2.5.2 Logarithm

If x is in the range demonstrated in Table 5.4, find $x'\in[\frac{1}{2},1]$ such that

$$x'=\frac{x}{2^k}. \tag{5.24}$$

[1]Note that the internal representation of s3.35 cannot store x and $-k\times\log(2)$ fully as it can be up to ~ 11 for fixed-point and ~ 88 for floating-point, but as we know that this is a subtraction of two very close values that will produce a small value x' that will fit into this representation, the top bits can be safely ignored.

Table 5.4: Approximate minimum and maximum ranges of values of $\log_e$ operation with different 32 bit 2's complement formats. ‡ — saturates to 0x80000000 below this input value; † — saturates to 0x7FFFFFFF above this input value. Full positive range of floating-point representable values is legal.

I/O formats	log input range	log output range
s16.15/s16.15	2^{-15} to $2^{16}-2^{-15}$	$-10.397...$ to $11.09...$
s0.31/s0.31	$0.367...^{\ddagger}$ to $0.99...$	$-0.99...$ to $\sim -2^{-31}$
s16.15/s0.31	$0.367...^{\ddagger}$ to $\sim e^{\dagger}$	-1 to $0.99...$
s0.31/s16.15	2^{-31} to $0.99...$	$-21.487...$ to 0
binary32/binary32	$\sim 2^{-149}(sub.)$ to $\sim 3.403\times 10^{38}$	$-103.278...$ to $88.722...$

If x is in fixed-point representation, we can find k by counting the leading zeros of x. For the normalized floating-point values, we can instead find $x' \in [1, 2)$ by just taking the significand and attaching "1" at MSB and making $k = exponent$. If the input is denormal, the significand has to be normalized to obtain x' and then k has to be appropriately decreased by the number of leading zeros.

When we have x' which is reduced to the convergence domain shown in (5.20), we can calculate the natural logarithm $\log(x')$ using the iterative algorithm presented previously. Then the result in the full range of x can be reconstructed as

$$\log(x) = \log(x') + \log(2^k) = \log(x') + k \times \log(2). \tag{5.25}$$

A well known issue in floating-point arithmetic with this range reduction approach is when x is very close to 1, as pointed out, for example, by Wong and Goto [140], Detrey et al. [141], and Langhammer and Pasca [142]. For some small value ε, if $x = 1-\varepsilon$, we will have an exponent of -1 and a significand of $2-2\varepsilon$ and the logarithm will be computed using (5.25) as $\log(2-2\varepsilon)-\log(2)$. Because the two numbers subtracted are so close to each other, we will get a very small value out of this operation. Then, when we want to construct a significand for a floating-point answer, this small value stored in some internal fixed-point representation will have to be shifted left to normalize, propagating zeros at the bottom of the significand. This will cause what is called *catastrophic cancellation*, causing large errors for x very close and below 1.

Following the approach outlined by Detrey et al. [141], the transformation of x' is done: if $x' \in [1.5, 2)$ (which is chosen for efficiency reasons, requiring only the check of one bit in the significand), then $x' = \frac{x'}{2}$ and $k = exponent + 1$. This error-free transformation causes $x' \in [0.75, 1.5)$ which avoids the problematic case. Note

that other authors use $\sqrt{2}$ instead of 1.5 to make this decision, which gives a range of possible outputs from the first term of (5.25) more centered around 0. In this work a choice was made to use a less expensive check of a single bit of the significand.

To summarize, in total range reduction and reconstruction for a natural logarithm requires a CLZ module and a shifter (5.24) as well as a multiplier by constant and an adder (5.25). For floating-point also needed are a CLZ module, a shifter at the end to normalize the answer and a 2's complement module to convert negative fixed-point answers to positive when constructing the significand from the internal fixed-point representation.

5.3 Implementation

This section shows how to implement the algorithm presented in Section 5.2.2 in hardware. First it is demonstrated how to implement a single iteration, which can then be instantiated an arbitrary number of times in a parameterized Verilog module. Then a top level architecture to drive the main block of iterations is shown.

5.3.1 Initial implementation considerations

The main considerations at the initial stage of implementation are as follows.

1. Maximum number of iterations supported.
2. Internal fixed-point precision and bits in the integer parts, in which the iterative algorithm works.
3. Related to both points above, precision and dimension of the $\log(1 + d_n \times 2^n)$ tables that are stored.

Since the fixed-point format used on SpiNNaker is s16.15, mainly due to balanced range and precision, as well as being supported by GCC and standardized, this is the main format that will be focused on making accurate enough in the exp-log accelerator. According to this format, the decisions of the above three points are made to achieve 1 LSB/ulp (absolute error smaller or equal to 2^{-15}) and then the error with other numerical formats is accepted even if higher than that, in order not to complicate the design further.

For the first item, the maximum number of iterations of the algorithm supported, a simple approach is to choose 32, given that we are working with 32-bit numerical

formats and knowing that the algorithm at each iteration finds roughly one bit of the answer. This also directly impacts how many clock cycles it will take to compute the function, therefore anything more than 32 is probably too slow and anything less will reduce the maximum supported accuracy. This also means that the two tables required for $\log(1+d_n \times 2^n)$ have to be stored for up to $n = 32$.

When that is fixed at 32, an internal representation has to be chosen. This is a fixed-point format which will store E_n and L_n while the algorithm is iterating and modifying those variables. This will also decide the precision of the constants $\log(1+d_n \times 2^n)$ in the look-up tables. Once again the C model of the algorithm can be used to experimentally find the minimum numbers of bits required. By aiming for 1ulp accuracy in the range $x \in [-10.397..., 11.09...]$ for the s16.15 numerical format exponential function, it was found that the minimum size of the fractional part has to be 35 bits. It was also found that to gain more accuracy, the look-up table entries for $\log(1+d_n \times 2^n)$ should not be rounded but truncated to the format with 35 bits, as then it produces a lower maximum error over the full range.

Finally, the internal representation should have a sign bit, for working with negative values as well as 3 integer bits, which is enough when looking at the range reduced inputs x' and the maximum values that the iterative function can generate from those inputs and is a minimum requisite for the choice of d_n in the iterative algorithm as discussed by Muller [16]. For logarithm this will also be suitable as the main source of error in (5.25) comes from the iterative part.

To summarize, an internal representation with a sign bit, 3 bits for integer and 35 bits for fraction — s3.35 — is chosen to obtain close to 1ulp accuracy for exponential and logarithm in the s16.15 format.

5.3.2 Single iteration unit

In Figure 5.3 a hardware unit for a single iteration step of the iterative algorithm, computing exponential or logarithm, is shown. The inputs are E_n and L_n, and the output is an update of these variables as per Equations 5.10 and 5.12. All three possibilities for progressing E_{n+1} and L_{n+1} are calculated in parallel:

- $E_{n+1} = E_n$ and $L_{n+1} = L_n$ when $d = 0$,
- $E_{n+1} = E_n + E_n 2^{-n}$ and $L_{n+1} = L_n - \log(1+2^{-n})$ when $d_n = 1$, and
- $E_{n+1} = E_n - E_n 2^{-n}$ and $L_{n+1} = L_n - \log(1-2^{-n})$ when $d_n = -1$.

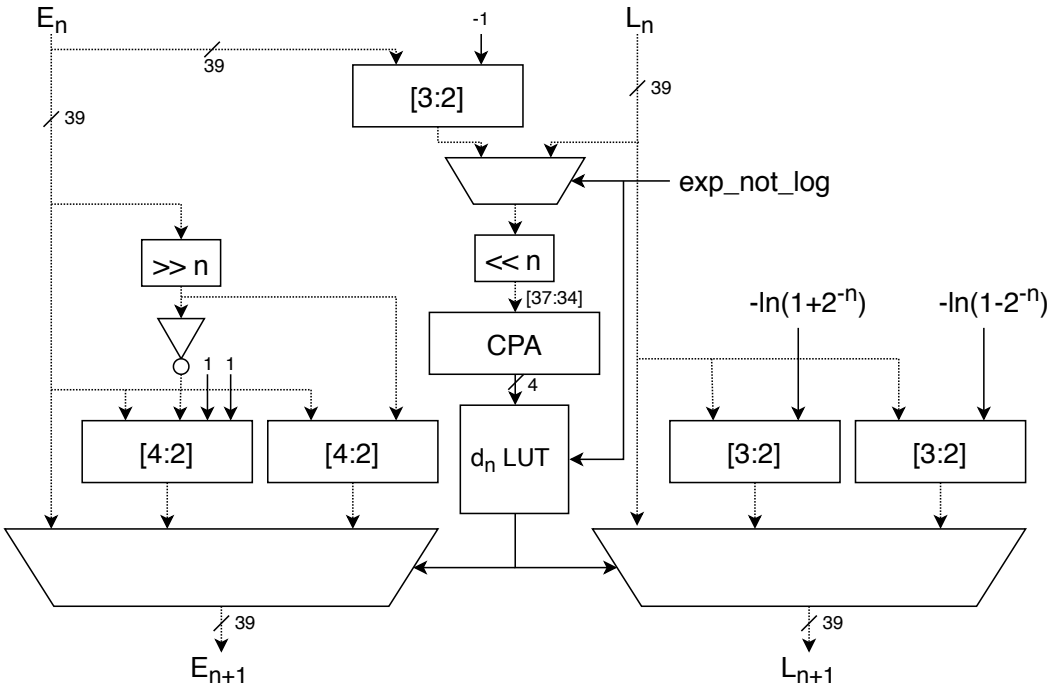

Figure 5.3: Architecture of a single iteration of the algorithm presented in Section 5.2.2. The internal representation is a 39-bit 2's complement carry-save. Dotted lines show carry-save busses that are made from two 39-bit busses for intermediate carry and intermediate sum. Solid lines are busses for binary numbers in non-redundant representation. Units labelled [3:2] are carry-save adders (Section 2.1.5) that add a carry-save number with a number in a standard non-redundant representation. Units labelled [4:2] function as two [3:2] carry-save adders (note that these units have two carry-in inputs which are used to sign-invert a carry-save number on the left side of the image). However, rather than chaining two [3:2] adders to make a [4:2], a fast algorithm presented in [31, p. 123] was used. Signal *exp_not_log* is used to choose whether it should calculate truncated L_n^* or λ_n^* for the choice of d_n.

Here d_n is calculated in the middle path of the circuit, as per (5.15), (5.18), and (5.19). At the bottom are two 3:1 multiplexers that correspond to different operations on E_n and L_n depending on the value $d_n \in \{0, 1, -1\}$, that choose the correct updates for the subsequent iteration.

It can be seen from the circuit that it has 7 different paths running almost independently in parallel. It was found that the middle part, where we calculate truncated L_n^* or λ_n^* (exponential or logarithm accordingly, as in Table 5.5) and choose the value d_n, is a critical path of this unit. To improve this path, value d_n was implemented as a one-bit-hot pattern $[001_2, 010_2, 100_2]$ instead of the actual 2-bit values $[0, 1, -1]$, which allows the bottom multiplexers to be faster.

Another slow component on this path is the 4-bit carry-propagate adder in the middle which is used to calculate the non-redundant representation of the truncated L_n^* or λ_n^*. This adder was split into two 2-bit adders run in parallel, one returning a 2-bit result and another adding lower bits returning a 2-bit result with carry-out. Then, this 5-bit value was used in the d_n look-up table. Because two 2-bit adders can run in parallel, this also gave a small improvement on the critical path. To save space, a 4-bit look-up table for d_n in Table 5.5 is provided, but it is straightforward to generate one for 5-bit values.

Table 5.5 shows how the choice of d_n can be made by considering only 4 digits of L_n or $E_n - 1 = \lambda_n$. Based on the limits of L_n^* and λ_n^* provided by Muller [16], the 4 digits of the truncated values were tied to the choice of d_n outlined in (5.15), (5.18), and (5.19). Only one truncated value for logarithm cannot be tied to one single possibility for the value that the full width λ_n^* would have. However, based on the limits provided by Muller [16] and an exhaustive sweep of the convergence domain using the C model of the algorithm, it was found that this combination never appears, therefore it can be ignored.

The table also shows some choices in bold font, which mark the choices that appear in the convergence domains of our specific case of range reduction, which are narrower than the full convergence domain of the algorithms (in our case $x' \in [-0.0917..., 0.771...]$ for exponential, and $x' \in [0.5, 1.5]$ for logarithm). This observation could provide some improvement to the critical path as only 3 digits would have to be checked for making the choice of d_n on each step. However, all further evaluation of this hardware unit is done with the full table implementation for generality, as it would be required, for example, when implementing binary64 functions, which would most likely result in wider ranges of x'.

Table 5.5: Look-up table for the choice of d_n on each step of the iterative algorithm for exponential and logarithm, addressed by $\hat{L_n^*}$ or $\hat{\lambda_n^*}$ which is L_n^* or λ_n^* truncated to 3 integer and 1 fractional bits in carry-save representation, and then added with a 4-bit adder to convert to a non-redundant representation. Since one bit at the top is lost for optimization of this LUT, the values are tied to the values of L_n^* and λ_n^* (which then allows us to choose d_n that will assure convergence) using the bounds shown by Muller [16]. Note that when $\hat{\lambda_n^*} = 1010_2$ we cannot know whether $\lambda_n = -3$ or $\lambda_n = 5$ without the extra bit, however, this does not matter as it was found, using the C model, that this combination never appears in the convergence domain of the log function. Choice of d_n for exp is as derived by Muller [16] and derived here for logarithm using the limits of λ_n shown by Muller [16] and the choices for d_n found in Section 5.2.3. Values marked with a dash are impossible based on the bounds and choices of d_n, and the ones in bold are the options that appear in the reduced range of x' discussed in Section 5.2.5.

$\hat{L_n^*}/\hat{\lambda_n^*}$ in binary	L_n^* in decimal	λ_n^* in decimal	d_n if exp	d_n if log
0000_2	0.0	0.0	**1**	**0**
0001_2	0.5	0.5	**1**	**-1** (0 if $n = 1$)
0010_2	1.0	1.0	**1**	**-1**
0011_2	1.5	1.5	**1**	**-1**
0100_2	2.0	2.0	-	-1
0101_2	2.5	2.5	-	-1
0110_2	3.0	3.0	-	-1
0111_2	3.5	3.5	-	-1
1000_2	-4.0	4.0	-	-1
1001_2	-3.5	4.5	-	-1
1010_2	-3.0	-3.0 or 5	-1	-
1011_2	-2.5	-2.5	-1	1
1100_2	-2.0	-2.0	**-1**	1
1101_2	-1.5	-1.5	**-1**	**1**
1110_2	-1.0	-1.0	**0**	**1**
1111_2	-0.5	-0.5	**0**	**0**

5.3.3 Main architecture

The previous subsection showed how to build a hardware module of a single iteration of the algorithm discussed in Section 5.2.2. Now, define integer $I \neq 0$ as the number of such iterations instantiated and connected in series and N_{cycles} as the number of clock cycles to iterate through those I iterations. Note that if $I > 1$ then $N_{cycles} \neq N$, where $N \leq 32$ is the algorithmic iterations defined in Section 5.2.1. Figure 5.4 shows the top level architecture that has $I = 2$ iterations instantiated and runs them in a loop as controlled by a software-programmable parameter $0 < N_{cycles} \leq \frac{32}{I} = 16$ giving $N = I \cdot N_{cycles} = 2N_{cycles}$ algorithm iterations.

First of all, 32-bit data is taken from the bus when available and sent through data preprocessing, which includes such things as converting floating-point input to fixed-point and separating various different parts of the floating-point input. Then range reduction is performed (Section 5.2.5) in the second clock cycle, at which point E_0 and L_0 are calculated in carry-save representation and will be used in the next cycle. To obtain full 32-bit accuracy, range reduction is performed in higher precision than the internal representation of s3.35. Then, the range reduced value is passed into the next clock cycle to start the main calculation using the iterative algorithm, which runs for N_{cycles} cycles, a parameter that is preprogrammed. By default $N_{cycles} = 16$, which gives $N = 32$ algorithm iterations in total. Each iteration sub-unit keeps track of the iteration number that it is running which is used to index the log tables for $\log(1 + d_n 2^{-n})$ when calculating (5.10). For example, iteration block 1 is running iterations with $n_1 \in \{1,3,5,7,9,11,13,15,17,\dots\}$ and iteration block 2 is running iterations with $n_2 \in \{2,4,6,8,10,12,14,16,18,\dots\}$.

Finally, on the last clock cycle, range reconstruction is performed as shown in the previous sections. In the end, the result is read out to the bus in s16.15, s0.31 fixed-point, or binary32 floating-point format (when this is binary32, an extra cycle is required to convert fixed-point answer to floating point). The format of the arguments, number of iterations N_{cycles}, and whether the operation is exponential or logarithm are defined by the different memory addresses allocated.

5.4 Results

This section will present various results obtained from the simulation and synthesis of the presented accelerator: first, the numerical accuracy is evaluated using the Verilog code in simulation, then synthesis is performed, and finally power is evaluated.

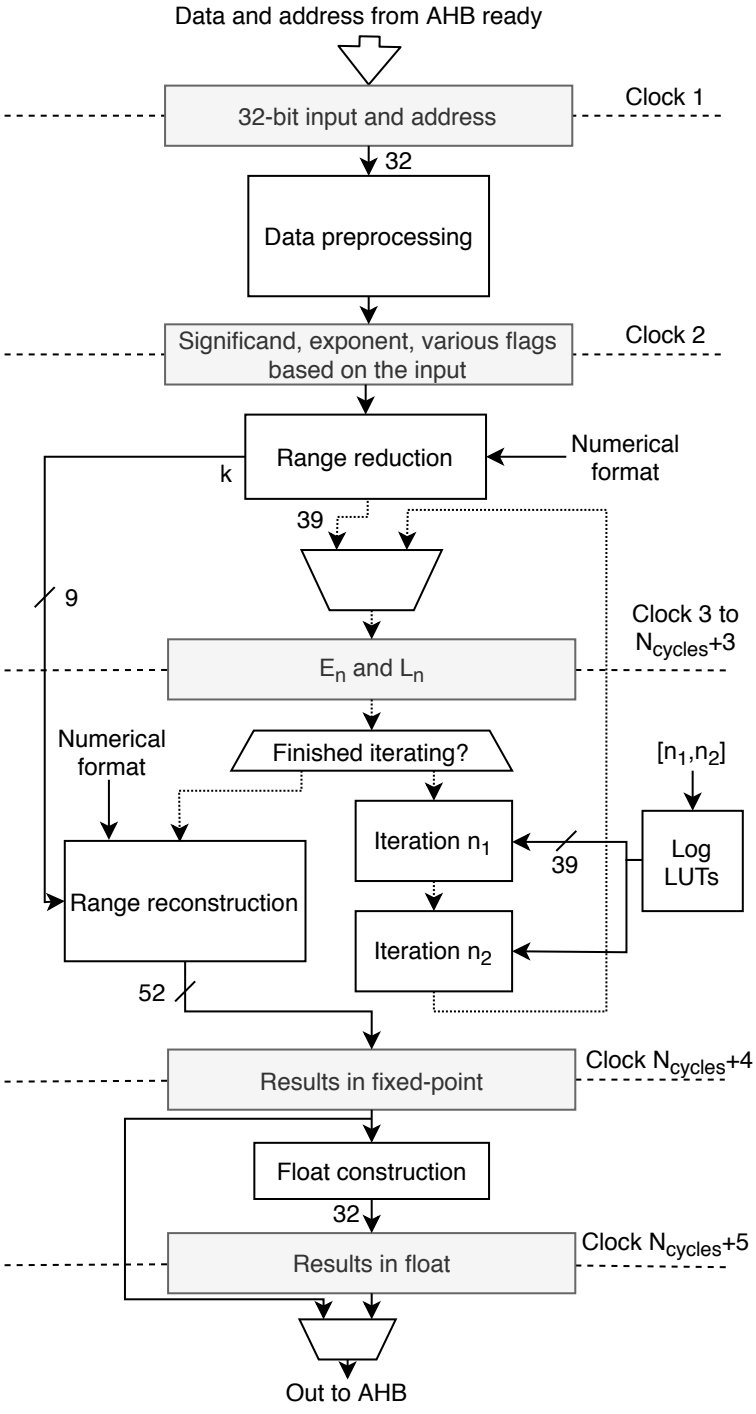

Figure 5.4: The architecture of the iterative exponential and natural logarithm unit. The number of cycles to iterate N_{cycles} is preprogrammed. Block "Log LUTs" represents a function which outputs the corresponding $\log(1-2^{-n_k})$ and $\log(1+2^{-n_k})$ values depending on the iteration number n_k that each iteration module is running.

5.4.1 Accuracy and monotonicity

Accuracy and monotonicity of the exponential and logarithm functions are evaluated in this section. Since it is useful to mix some fixed-point input and output formats, both mixed and non-mixed formats are evaluated. Furthermore, the accelerator can be configured beforehand to do a certain number of iterations depending on the accuracy that is required, therefore accuracy and monotonicity is evaluated for multiple such configurations.

In terms of the accuracy requirements for elementary functions, there are three main standards that could help in making a decision of what should be our aim: the ISO standard that includes fixed-point arithmetic [34], the IEEE 754-2019 floating-point standard [22], and the OpenCL standard [143] that was used by Langhammer and Pasca [142] for what is probably state-of-the art analysis of exponential and logarithm designs on FPGAs. The first one specified 1 or 2ulp accuracy, depending on whether the user requests accuracy or speed of operations. The second standard requires all of the operations to be done as though they were performed in an infinite precision arithmetic without any bounds on the range and then *correctly rounded*, which means 0.5ulp of maximum error — or always return the closest possible answer to the true mathematical value, in a given numerical format. The last one, the OpenCL standard, requires either 3ulp or 4ulp of accuracy for exponential and logarithm. In this work the approach was taken, as mentioned before, of achieving 1ulp accuracy on the s16.15 exponential and not to increase the hardware resources if other numerical formats do not meet this accuracy, hereby minimizing chip area and leakage costs.

There is also an interesting debate that can be had; neuromorphic applications are unique and possibly these standards are too strict, especially the requirement of correctly rounded results which are required only because the standard must cover the whole space of possible scientific computing applications that might be run on any given system, and that usually have different sensitivities to the errors in arithmetic operations. Correctly rounded arithmetics also help to do error analysis since only one rounding error bound is required everywhere. Thus the standard has no alternative but to enforce, on the designers of mathematical libraries and hardware units, to do the best that is possible in a given numerical format. As there are no similar standards yet developed for spiking neural network simulators, it is useful to have accuracy control and be able to speed up simulations if some models appear to not require high accuracy on these elementary functions at least — this is in part the motivation for having accuracy control in this accelerator.

Regarding *monotonicity*, Muller [16] provides a definition and some useful formulation, but a basic understanding comes down to the following: if a function is increasing or decreasing over some interval, then the approximation in quantized arithmetic should also be doing that. Assuming a monotonically increasing function $f(x)$, if we call it with a larger argument $f(x+\varepsilon)$, it must hold for all possible values x in some given range that $f(x+\varepsilon) \geq f(x)$. This can be important in neuromorphic applications to avoid bringing noise into the simulations — for example in modelling exponential decay we do not desire a certain amount of time to decay a value less than anything that is smaller than that amount of time. This property requires a more in-depth look in the context of spiking neural networks, but this was out of scope of this study.

Rounding is not always helpful in all cases — depending on the internal representation and the function being implemented, rounding an approximated value can get the answer even further from the exact answer. This can happen if the bits that are being rounded are, in fact, not correct, because for example they are accumulated errors while iterating that were shifted left on range reconstruction, especially if a small number of iterations is chosen. Experimentally it was found that rounding was making errors larger in s16.15 exponential, while in other formats and functions it was helping to reduce the maximum errors. However, due to saving chip area and very minor improvements to the errors that rounding adds in some cases, it was done only in floating-point formats.

5.4.1.1 Accuracy of fixed-point exp and log

To measure the accuracy of the fixed-point formats of the unit and verify it over a wide range of available arguments, a similar approach to that of Partzsch et al. [59] was taken where the accelerator's accuracy was compared to the double-precision (binary64) floating-point exponential function of the C standard library `math.h`. This should be a suitable reference since GCC estimates are that 1ulp accuracy is achieved for exponential and logarithm implementation on most of the supported platforms [144], although a warning is given that due to a large search space the maximum errors reported might not be from the exhaustive tests.

Given an argument x, calculate absolute error (with or without taking the absolute value as required) with

$$\Delta\exp = |\exp'(x) - \exp(x)|, \tag{5.26}$$

where $\exp'$ is hardware accelerator function and exp is `math.h` function `exp()`. By

Table 5.6: Accuracy and monotonicity of exponential function for different numbers of configured iterative clock cycles N_{cycles}, in the accelerator with 2 algorithm iterations per clock cycle, and s16.15 input and output format. All results for exponential are from exhaustive tests in the function domain whereas for log they are limited due to the time it takes to exhaustively test this in Verilog simulation — only $N_{cycles} = 16$ is exhaustive and others are from $\sim$ 8M values uniformly distributed across the function's input domain. ulp $= 2^{-15} = 0.000030517578125$. Maximum error reported is rounded up to the larger integer.

	exp			log		
N_{cycles}	Max error	Average	Monotonic	Max error	Average	Monotonic
16	1 ulp	0.477 ulp	Yes	2ulp	0.5ulp	Yes
14	8 ulp	0.564 ulp	Yes	2ulp	0.5ulp	Yes
12	125 ulp	3.172 ulp	Yes	2ulp	0.5ulp	Yes
8	30044 ulp	693.33 ulp	Yes	2ulp	0.56ulp	No

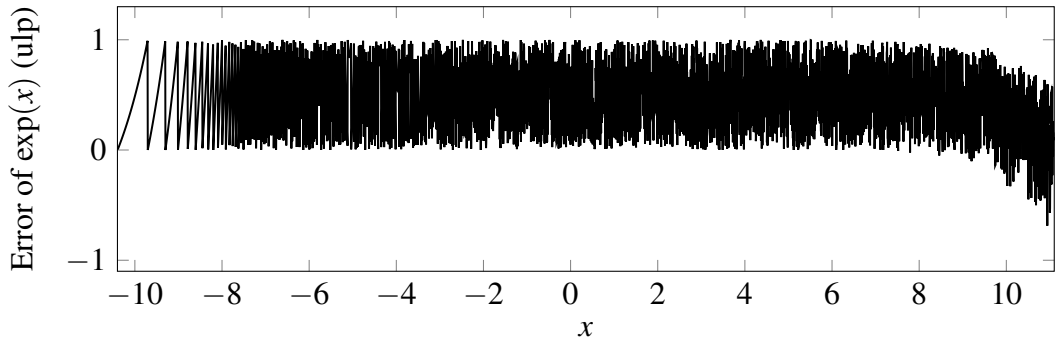

Figure 5.5: Accuracy of the s16.15 exponential accelerator with $N_{cycles} = 16$ cycles set-up to iterate (every 160th value of the input domain is plotted).

sweeping through all possible s16.15 arguments for exponential, it was found that, when the unit is configured to do 32 algorithmic iterations with $N_{cycles} = 16$ (number of cycles to loop in the iterative part of the accelerator), all of the outputs had an absolute error below 1ulp ($2^{-15} = 0.000030517578125$, the smallest value representable by the LSB in s16.15 format), meaning that the result from the exponential hardware accelerator is one of two neighbouring values of the binary64 floating-point reference. By running a sweep over all values, it was also verified that the exponential function at this accelerator configuration is monotonic.

For the natural logarithm function with $N_{cycles} = 16$ an exhaustive test of approximately 2 billion samples across the range of possible inputs $x \in [2^{-15}, 65536)$ was

Table 5.7: Accuracy and monotonicity of exponential and logarithm functions for different numbers of iterations N_{cycles}, in the accelerator with 2 algorithm iterations per clock cycle, and s0.31 input and output format. Experiments with $N_{cycles} = 16$ are run exhaustively over the function input domain. Other results are shown by using approximately 5M input values uniformly distributed in the input domains of the functions. ulp $= 2^{-31} \approx 0.000000000465$. Maximum error reported is rounded up to the larger integer.

	exp			log		
N_{cycles}	Max error	Average	Monotonic	Max error	Average	Monotonic
16	2ulp	0.29ulp	Yes	2ulp	0.54ulp	Yes
14	9ulp	2.22ulp	Yes	7ulp	2.32ulp	Yes
12	127ulp	39.13ulp	Yes	87ulp	34.3ulp	Yes
8	32416ulp	9537ulp	No	22644ulp	8737ulp	No

run. It was found that 99.999 % of samples had an absolute error below 1ulp and a small number of tests (3326) had a maximum error of 1.01ulp. By sweeping over all possible arguments, it was confirmed that the logarithm function at this accelerator configuration is also monotonic.

Table 5.6 lists maximum absolute error and whether the function is monotonic or not for some configurations $N_{cycles} \in \{8, 12, 14, 16\}$. Figure 5.5 shows the absolute errors for a subset of test samples from the exponential function in the full input range. From this figure it can be seen that all the outputs from the exponential function have absolute error below 1ulp. The results shown in Figure 5.5 correspond to the first row of Table 5.6.

Next, the accuracy of the exponential and logarithm with inputs and outputs in s0.31 format was measured (Table 5.7) . When working in s0.31 input/output format, for exponential we have a domain of $x \in [-1, 0)$ and for logarithm $x \in [\sim 0.368, 1)$ — these ranges can be covered exhaustively but it takes around two days to run in Verilog simulation. Due to this, as before only the most accurate configuration was run exhaustively and other configurations checked for some subset of all possible inputs.

When the accelerator was configured to do $N_{cycles} = 16$ cycles, it was found that most of the possible exponential outputs from the inputs in the domain $x \in [-1, 0)$ had accuracy below 1ulp ($2^{-31} = 0.000000000465...$) with some arguments (9741) having a larger error below 2ulps (average error of 0.29ulp) — the first row in Table 5.7. This means that most of the results of the accelerator are one of the two values on both sides of the binary64 sample, and in rare cases it is one of the four neighbouring values.

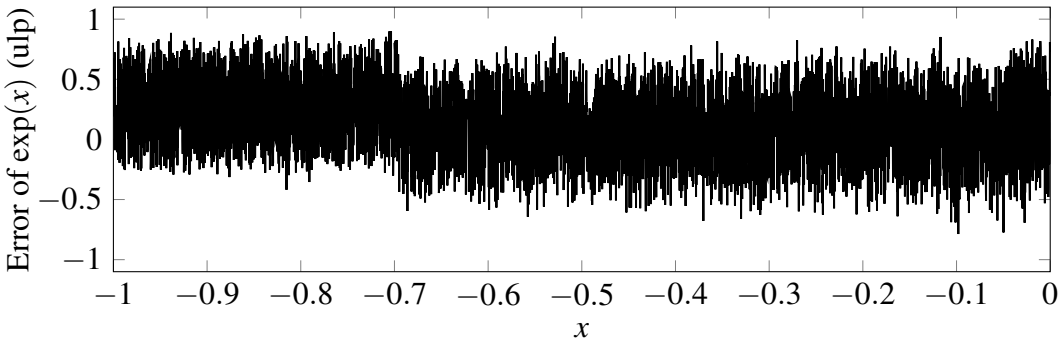

Figure 5.6: Accuracy of s0.31 exponential accelerator with $N_{cycles} = 16$ cycles set-up to iterate (every 400 000th value of the input domain is plotted).

Note that $\text{ulp}_{\text{s16.15}} = 65536\text{ulp}_{\text{s0.31}}$ and therefore even the $N_{cycles} = 8$ configuration is more numerically accurate than the configuration with $N_{cycles} = 16$ in s16.15 format, and should be used instead of that if the wide input domain is not required in a given application. Figure 5.6 shows the absolute errors from some inputs in the full input range of the s0.31 exponential function.

For the logarithm function in s0.31 format with the accelerator set to do $N_{cycles} = 16$ cycles, in the range $x \in [\sim 0.368, 1)$, the maximum error is 2ulp with the average error of 0.54ulp. Note that at this input format, the logarithm function has a very limited domain, with ~ 0.368 being the smallest value before the result saturates. Therefore, it is more likely that most applications would use a mixed input/output format s0.31/s16.15 instead (Refer to Table 5.4).

5.4.1.2 Accuracy of mixed-format fixed-point exp

This section shows results from the tests of the accelerator when different input/output fixed-point formats are mixed. This can be either s16.15 as input and s0.31 as output or s0.31 as input and s16.15 as output. Some such combinations for logarithm provide a very limited function domain, for example s16.15/s0.31 in Table 5.4 and is unlikely to be used in any application. Here only the exponential function with the input format of s16.15 and output format of s0.31 is analysed here, which, as shown, has a convenient input and output range to maximize the accuracy of exponential decay function (Section 3.2). The results are shown in Table 5.8.

For exponential in input/output format of s16.15/s0.31, the input range is $x \in [-21.488, 0)$. By sweeping through the full domain, when the unit is configured to do $N_{cycles} = 16$ cycles, it was found that the maximum error was below 1ulp ($2^{-31} =$

Table 5.8: Accuracy and monotonicity of the exponential function for different numbers of iterations N_{cycles}, in the accelerator with 2 algorithm iterations per clock cycle and mixed fixed-point input and output format (s16.15/s0.31). ulp $= 2^{-31} \approx$ 0.000000000465. Maximum error reported is rounded up to the larger integer.

N_{cycles}	Max error	Average	Monotonic
16	1ulp	0.477ulp	Yes
14	8ulp	0.567ulp	Yes
12	125ulp	3.17ulp	Yes
8	30180ulp	675ulp	Yes

0.000000000465...). When N_{cycles} is reduced, more error is introduced at the bottom, with $N_{cycles} = 14$ having a maximum of 8ulp of error, and so on. When $N_{cycles} = 8$, a maximum error of 30180ulp with an average error of 675ulp was measured. While it is a very large relative error, the absolute value is smaller than the machine epsilon of the s16.15 format and therefore a much faster and more accurate *exponential decay* function can be implemented by using this accelerator configuration instead of the standard s16.15 format for both input and output. And as shown in Chapter 3, mixed-format multiplication can easily be implemented to give more accuracy in exponential decay when computing $X_0 e^{-\frac{\Delta t}{\tau}}$ (also, making sure that $-\frac{\Delta t}{\tau}$ is calculated as accurately as possible).

5.4.1.3 Accuracy of binary32 exp and log

To measure accuracy of the floating-point exponential and logarithm accelerator functions it is important to note that the LSB (machine epsilon) has different values for different numbers depending on the exponent and therefore errors have to be measured relative to that (as discussed in Section 2.1.3). Ulp error measurement is very useful in this case as it is, by definition, measuring errors relative to the exponent. As Muller [37] points out, it is important to use the error for measurement that is not derived from the approximation, but from the reference values (given a value x and its approximation X, we should find the size of ulp to measure error using ulp(x) and never ulp(X)), and the error measurement test bench developed here takes that into account.

First, given some argument in the binary32 floating-point format, a reference value is calculated using C binary64 libraries for exponential and logarithm. Then this reference value in binary64 is converted to the nearest binary32 floating-point number, and its ulp is calculated. Finally, it was checked that the approximation of exponential or

Table 5.9: Accuracy and monotonicity of binary32 exponential and logarithm functions for different numbers of iterations N_{cycles}, in the accelerator with 2 algorithm iterations per clock cycle. The first test with $N_{cycles} = 16$ was done using all representable values in the full function domain while the other tests for speed were performed using around 4 million arguments uniformly spread in the function input domain. Maximum error reported is rounded up to the larger integer.

	exp			log		
N_{cycles}	Max error	Average	Monotonic	Max error	Average	Monotonic
16	1ulp	0.066ulp	Yes	192ulp	0.25ulp	Yes
12	1ulp	0.086ulp	Yes	2043ulp	0.27ulp	Yes
8	252ulp	15ulp	No	$\sim$ 10Mulp	8.28ulp	No

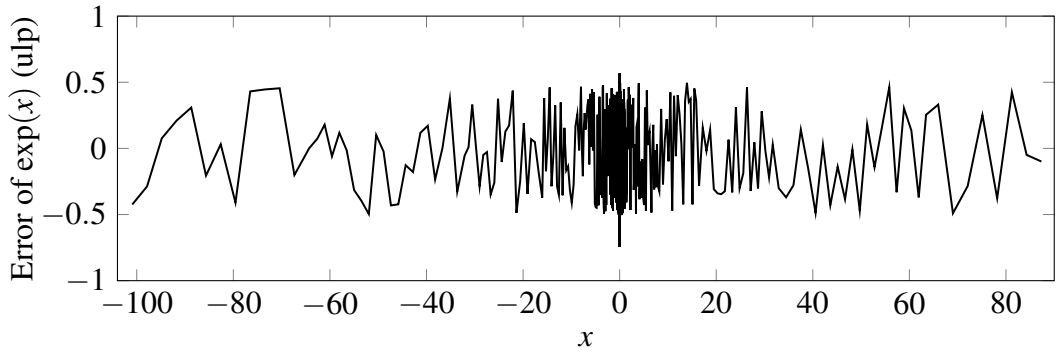

Figure 5.7: Accuracy of binary32 exponential in the full accuracy configuration of $N_{cycles} = 16$ cycles (every 400 000th floating-point value in the exponential function domain shown).

logarithm coming from the accelerator, called with the same floating-point argument, is within $\pm k \times \text{ulp}(x)$ from the reference value and report the maximum k over the full input domain of the function. The fact that ulp has different values to the left and right of some values in floating-point arithmetic is well known, specifically when the exponent changes between the two neighbouring values, and this fact is also taken into account by checking ulp on both sides of the reference.

Table 5.9 contains the accuracy measurements of exponential and logarithm functions in 3 different accuracy configurations of the accelerator. From the exhaustive test of all possible input values when $N_{cycles} = 16$, it was found that exponential had 1ulp accuracy and logarithm had 192ulp accuracy. Another observation from this table is that for some lower accuracy settings, the average error of the exponential function is increasing slightly, but the maximum error stays at 1ulp and the function is monotonic.

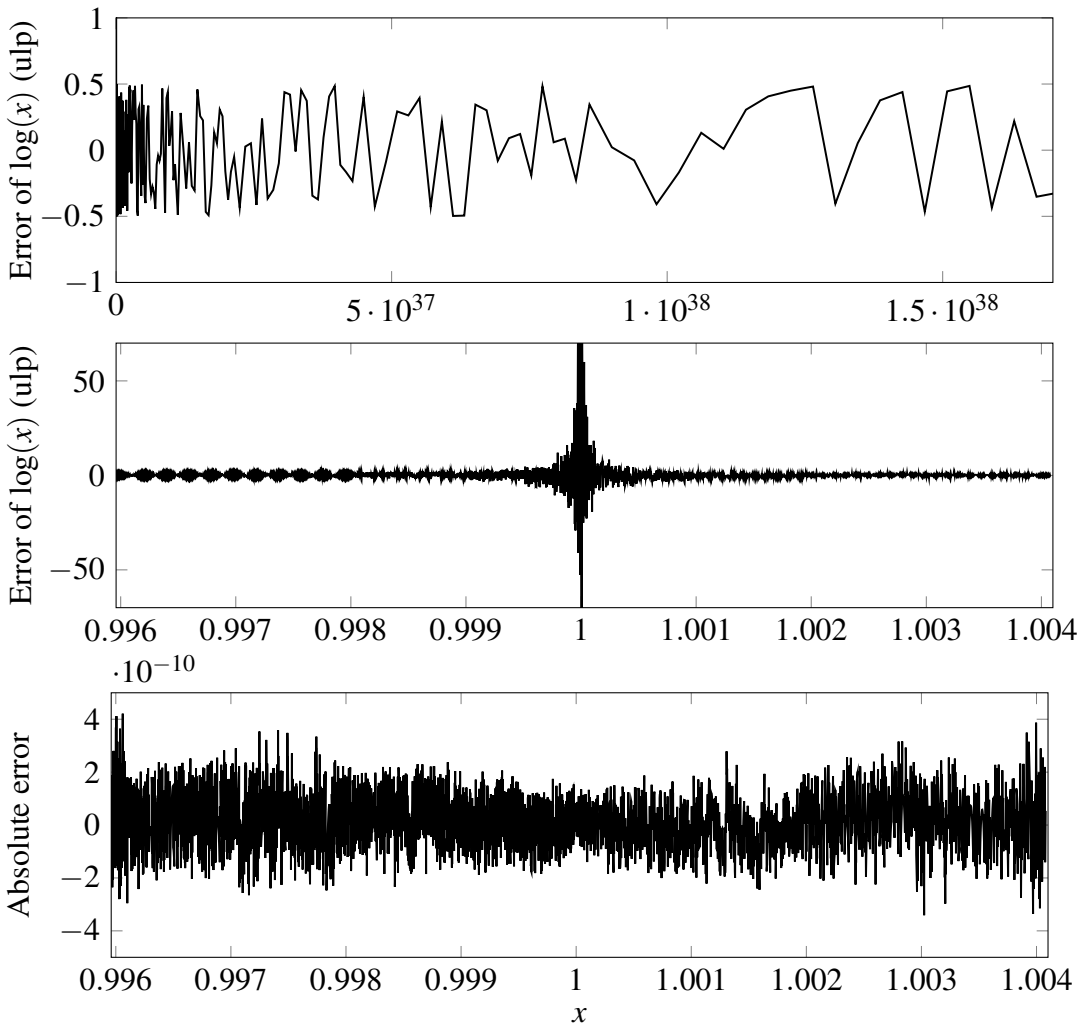

Figure 5.8: Accuracy of floating-point logarithm in the full accuracy configuration of $N_{cycles} = 16$ cycles (every 400 000th floating-point value in the logarithm function domain shown). Top: some values from the full input domain; middle: relative errors near $x = 1$; bottom: absolute errors near $x = 1$.

Some of the errors in the exponential function are plotted in Figure 5.7 which shows that most of the errors are below 0.5ulp (average error of 0.066ulp shows that most of the errors in fact are closer to 0ulp than to 1ulp).

In terms of the logarithm function, a large error of 192ulp appears for input arguments that are very close to 1, but are not exactly 1, which produce very small answers close to 0. However, since internally the computation is done in s3.35 fixed-point representation, there are not enough bits to represent these small values, and when a normalized floating-point answer is being constructed multiple zeros are propagated

at the bottom producing large relative errors. This happens for a very small range of inputs $x \in [0.9977340698..., 1.003865242...]$, where 14673 inputs produced an error larger than 1ulp.

Options for calculating this range using a polynomial approximation in high precision instead of running the iterative algorithm were considered, but significant hardware costs resulted due to a series of wide multipliers and adders (exhaustive tests were out of scope but 4th order polynomial seemed to be required). Instead a decision was made to leave this small range as it is because the absolute errors are very small, given that the values are close to 0 where steps between the neighbouring values are $\sim 2^{-126}$.

Figure 5.8 shows some values from running error measurements of the logarithm function in binary32 mode. First, it can be seen that most of the errors in the full input domain (all positive values greater than zero representable in floating-point) are below 0.5ulp (with an average error in total being 0.25ulp). Furthermore, while relative errors near $x = 1$ start to get quite large, absolute error in the most relatively inaccurate ranges are smaller than in the surrounding ranges. Therefore, while the maximum error of 192ulp is not meeting any of the accuracy requirements laid out by the floating-point standards, this happens only in a very small range of inputs and the absolute error is not significant, therefore not worth fixing by adding extra hardware resources.

5.4.2 General exponentiation function

Another useful feature that this module provides for SpiNNaker is that it is possible to compute a general exponentiation function for some limited range of arguments by using exp and log modes of the accelerator as follows: $x^a = (e^{\log(x)})^a = e^{\log(x)\times a}$. The delay of this function comprises of two invocations of the accelerator plus a multiplication, but generally this will be much faster than any software implementation. Furthermore, using the equation above, it is possible to achieve square root ($\sqrt{x} = x^{\frac{1}{2}} = e^{\log(x)\gg 1}$, where $\gg$ is arithmetic shift right), reciprocal ($\frac{1}{x} = x^{-1} = e^{-\log(x)}$) and sigmoid function $\frac{1}{1+e^{-x}} = (1+e^{-x})^{-1} = e^{-\log(1+e^{-x})}$ (although possibly this can be done faster using the exponential accelerator once and hardware division available in ARM M4F). All of these functions can be achieved by combining the accelerator for exp and log, and the typical ARM instructions for multiplication, shifting, and add/subtract. Most of these functions appear in SNN models, especially reciprocal and sigmoid for developing intrinsic currents in *Hodgkin-Huxley* type of neuron models [132] and power law weight change dependence ($\Delta w \propto w^{0.4}$) in STDP rule discussed by Morrison et al. [67].

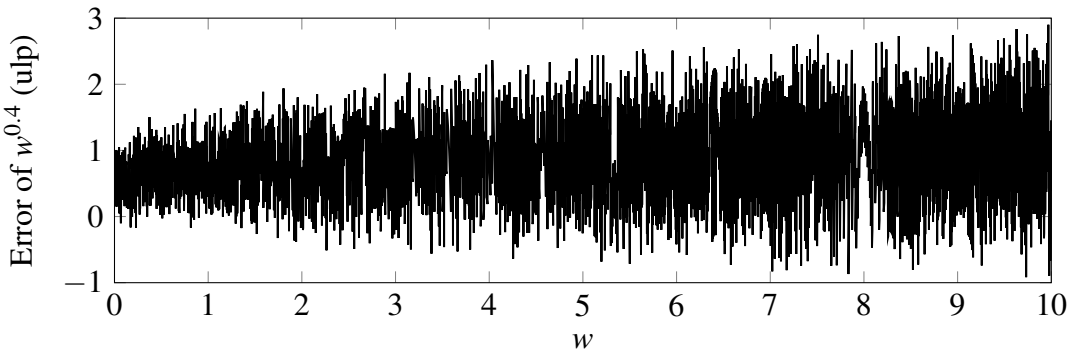

Figure 5.9: Accuracy of the general exponentiation function $w^{0.4}$ in s16.15 numerical format in the full accuracy configuration of $N_{cycles} = 16$ cycles, compared to C binary64 `pow(w, 0.4)` function.

While running exhaustive tests for all possible bases and powers is out of scope due to a very wide search space, some basic tests were ran using the power of 0.4 and assuming weights between 0 and 10 in s16.15 format for bases that might be used in the weight dependence of STDP developed by Morrison et al. [67]. A logarithm is taken of each weight value in the range $w \in (0, 10]$, the result is multiplied by 0.4 in u0.32 fixed-point format, the result is then rounded to the nearest s16.15 value and exponential of that is taken to get the final answer of $w^{0.4}$. In this test, the maximum error was 2ulp with an average of 0.65ulp in the range $w \in (0, 1]$ (which can be useful if weights have these specific bounds) and 3ulp with an average of 0.9ulp error in the full range.

Figure 5.9 shows a plot of ∼80 000 output errors in this range and, as expected, the errors grow as w grows due to the multiplier in the exponent. For this limited range of input and fixed exponent this might be a suitable faster solution than the software library for the general exponentiation function, but by no means a replacement to a very accurate implementation due to possibly very large errors as shown by Muller [16]. For a more in depth discussion about errors involved in this implementation see [29, Ch. 7] and [16, Ch. 13].

5.4.3 Synthesis study

A synthesis study of the presented design was executed, using the *makeChip* hosted design service platform [121] for the GLOBALFOUNDRIES 22FDX technology [122] in which the SpiNNaker2 chip is being developed. An ultra-low voltage *8t-CNRX*

Table 5.10: Synthesis of the exp-log accelerators with different numbers of iterations per clock cycle. The clock was constrained at 150 MHz

Iterations per cycle, I	Normalized area	SLVT cells	Timing met
1	1	2.7%	Yes
2	1.12	8.8%	Yes
4	1.22	34.5%	Yes
6	1.79	56.8%	Yes

Table 5.11: Synthesis of the exp-log accelerators with different numbers of iterations per clock cycle. Clock was constrained at 250 MHz

Iterations per cycle, I	Normalized area	SLVT cells	Timing met
1	1	38.1%	Yes
2	1.05	45.8%	Yes
4	1.36	63.3%	Yes
6	2.21	75.9%	No

standard-cell library with multiple voltage threshold options is used for implementation. The standard cells use the *adaptive body biasing* (ABB) technique for post-silicon adaptation of transistor threshold voltage [123, 124]. Namely two main categories of cells are used: LVT and SLVT cells — the former slower than the latter, but has significantly less leakage. A nominal supply voltage of 0.50 V is considered for low power operation. Synthesis is performed in a worst case operating condition at 0.45 V and $-40\,^{\circ}\mathrm{C}$.

Firstly, multiple accelerators were synthesised with different numbers of iteration modules placed per clock cycle (denoted by a variable I) which is interesting from the perspective that this algorithm is sequential and the more iterations are evaluated on each cycle the lower the latency to obtain some fixed accuracy answers. Similarly to the accelerator with $I = 2$ shown in Figure 5.4, in Tables 5.10 and 5.11 results are shown of multiple accelerators synthesised with $I \in \{1,2,4,6\}$ for different clock constraints $f_{clk} = 150\,\mathrm{MHz}$ (Table 5.10) and $f_{clk} = 250\,\mathrm{MHz}$ (Table 5.11). The tables list the area normalized to the area of the most unconstrained sample, which is the approximate area before *place and route*, the percentage of SLVT cells used (this gives us a convenient measure of major leakage increase if high and an indication that the synthesizer is struggling to meet the timing constraints), and whether timing was met or not after all of the possible optimisations (including usage of fast cells).

This data shows that when $f_{clk} = 150\,\text{MHz}$, for $I \in \{1,2\}$, a very small number of SLVT cells are required to meet timing constraints and for $I > 2$, a lot of SLVT cells are placed on the path with iterations to meet timing constraints (we know this since other logic blocks such as range reduction are not changing here). When $f_{clk} = 250\,\text{MHz}$, a significant number of SLVT cells are required for all the accelerator versions, most likely both in the iterative path and in the range reduction/reconstruction cycles. However, by observing the timing optimisation report, it was visible that when $f_{clk} = 250\,\text{MHz}$, $I \in \{1,2\}$, most of the SLVT cells were used on the range reduction and reconstruction paths, as the iterative path of the circuit has a smaller propagation delay. When $I > 2$, the percentage of SLVT cells rapidly increases to meet the timing constraints on the iterative path as at this point it is the slowest path of the circuit. Finally the limitations of this algorithm are reached when $f_{clk} = 250\,\text{MHz}$ and $I = 6$ where timing constraint cannot be met even with SLVT cells on the iterative path.

Next, a sweep of the clock speed constraint in the range $f_{clk} \in [50, 400]$ was performed, and area and leakage power (normalized) data was gathered to compare three accelerators with $I = 1$, $I = 2$ and $I = 4$ (Figures 5.10 and 5.11). It can be seen in Figure 5.10 that area is growing, as more LVT cells are used to parallelize various parts of the circuit to meet the timing constraint, until 150 MHz. After that point, the synthesizer starts using SLVT cells, of which it most likely needs fewer, hence the area decrease. It is clearly visible that the area difference between the accelerators with $I = 1$ and $I = 2$ is negligible whereas 4 iterations has almost double the area of the version with one iteration.

Figure 5.11 shows the leakage comparison in the same conditions. It can be seen that leakage for the accelerators with $I = \{1,2\}$ increases significantly from around $f_{clk} = 100\,\text{MHz}$ when the synthesizer starts adding SLVT cells on the range reduction and range reconstruction paths. On the other hand, the accelerator with $I = 4$ has a rapid increase in leakage straightaway due to increased usage of SLVT cells both on the iterative path, and range reduction and reconstruction paths. Once again, accelerators with $I = 1$ and $I = 2$ show minor differences in leakage, whereas the accelerator with $I = 4$ has almost order of magnitude more leakage on most of the clock frequencies.

Further observation from this data is that leakage power is a major problem at high clock frequencies — unlike chip area, leakage keeps increasing supralinearly with the increasing clock frequency. Also, since there is only a minor increase in leakage by going from 1 to 2 iterations, it is clear that all of the leakage is mainly in range reduction/reconstruction paths in those accelerators — not surprising since those paths

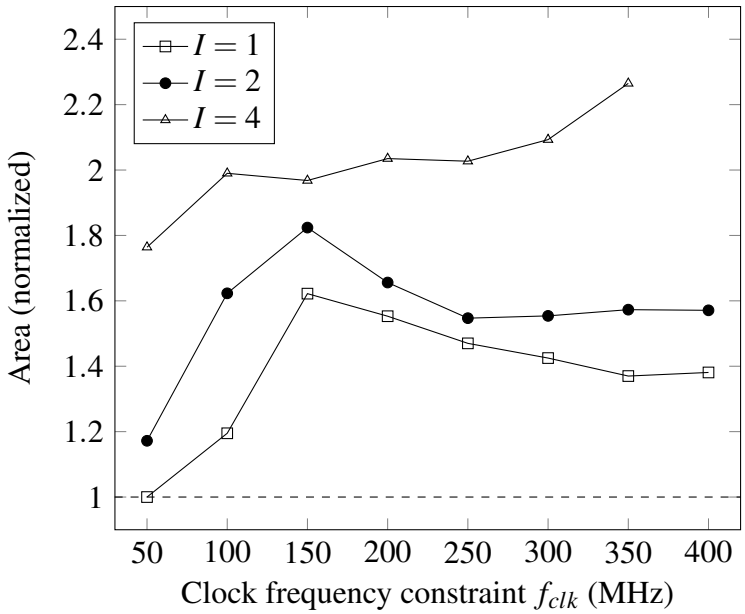

Figure 5.10: Circuit area of the accelerator when synthesised with different clock constraints. Three accelerator versions are shown with 1, 2, and 4 iteration units per clock cycle.

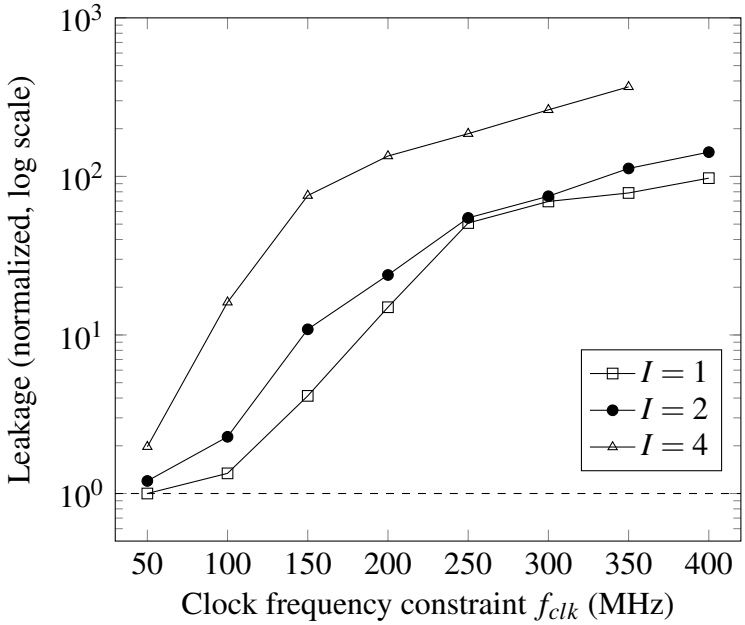

Figure 5.11: Leakage of the accelerator when synthesised with different clock constraints. Three accelerator versions are shown with 1, 2, and 4 iteration units per clock cycle.

contain count-leading-zero modules, shifters, and carry-propagate adders while the iterative part is fast due to the use of carry-save representation as explained above.

5.4.4 Accuracy-power-latency-area trade-offs

Figures 5.12 and 5.13 summarise various tradeoffs for a subset of accelerators that have been synthesized with two clock constraints: at 150MHz and 250MHz. It is clear from these diagrams that the accelerator with $I = 2$ is a good choice for SpiNNaker2, with the aim to decrease leakage power. By trading off a bit of area and leakage compared with the accelerator $I = 1$, a substantial decrease in latency, for the same accuracy settings, is achieved. This is not the case with the accelerators with $I = 4$, since that tradeoff does not work that well due to disproportional increase in leakage. However, the option $I = 4$ would be useful in order to minimize latency and dynamic power by paying the cost in leakage — at $f_{clk} = 150$MHz even higher settings $I = 6$ were possible to synthesize with timing constraints met, even further pushing the latency at the cost of leakage power (Table 5.10).

5.4.5 Power analysis

The prototype core (PE), which will be used in the SpiNNaker2 chip, was equipped with the presented elementary function accelerator (version with $I = 2$ iterations per clock cycle). The power consumption of the whole PE is analysed in a typical process condition at worst case power conditions of 0.8 V at 85 °C. SpiNNaker2 PEs are designed to work in a nominal supply voltage of 0.5 V, but 0.8 V is used here since dynamic voltage and frequency scaling (DVFS) will enable fast operation at a second performance level of 0.8 V as shown by Höppner et al. [61].

Some basic tests of the accelerator were performed by running realistic software test cases including an arbitrary number of calls to the exponential and logarithm functions, on a netlist of a complete PE. Tables 5.12 and 5.13 provide comparison of the accelerator to some software libraries of exponential and logarithm functions. It can be seen that at some area cost we can obtain a much higher throughput exponential function with very small energy consumption compared to software implementation.

Furthermore, in Table 5.14 provided is a comparison of the exponential accelerator and in Table 5.15 the natural logarithm accelerator from this study to some other similar designs available in the literature. It can be seen that the presented solution has an

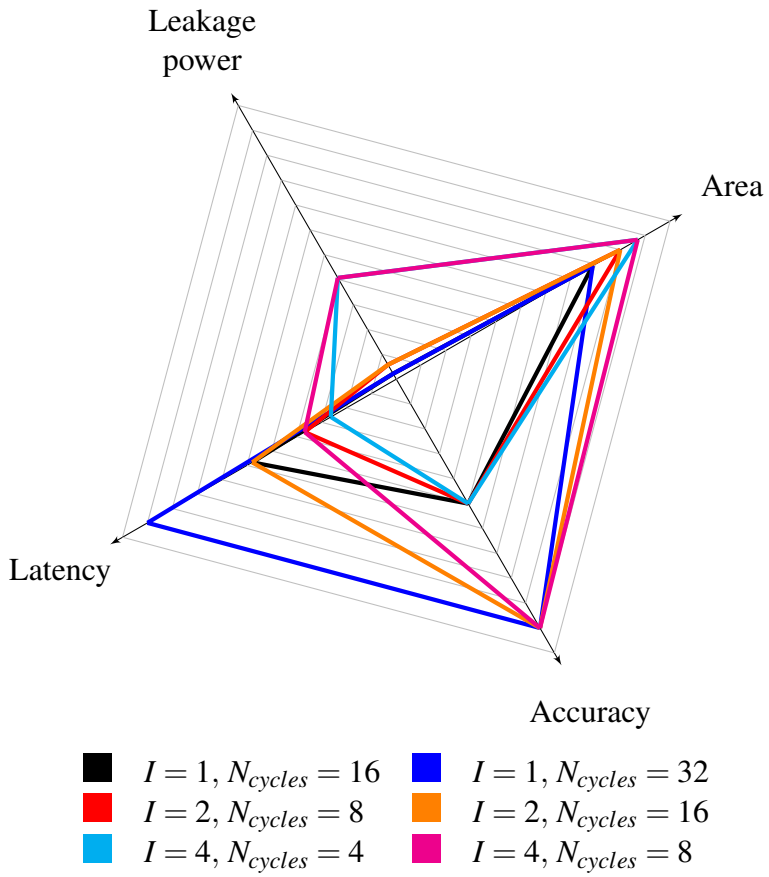

Figure 5.12: Tradeoffs with various versions of the accelerator for $f_{clk} = 150$MHz. The main parameters are I, N_{cycles}, and f_{clk}, where I is number of iteration units per cycle, N_{cycles} is software-programmable number of iterations, and f_{clk} is the clock frequency constraint in the synthesis.

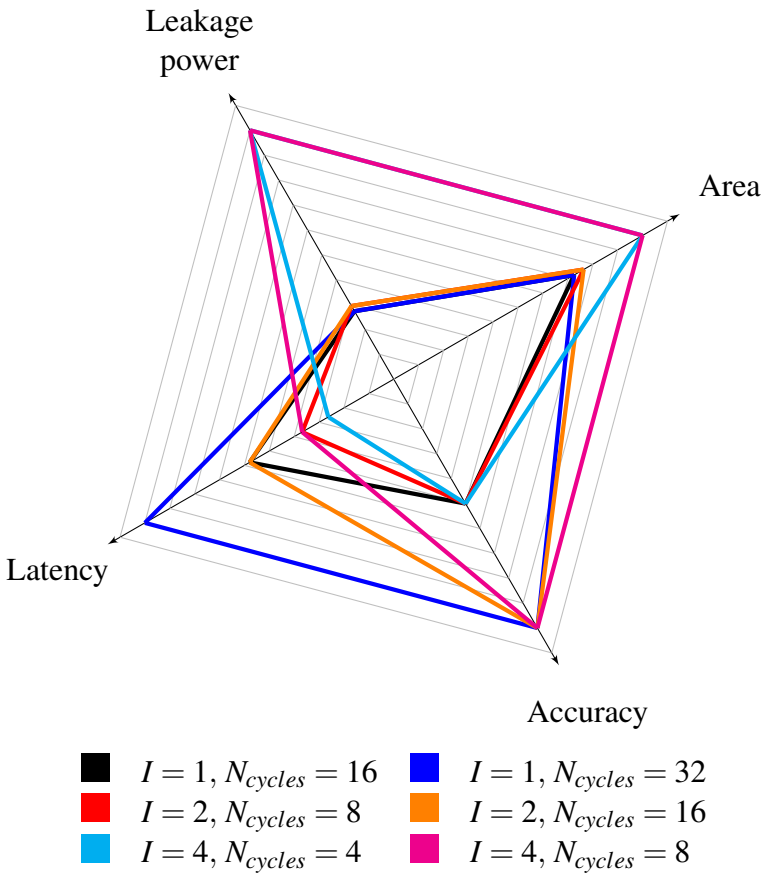

Figure 5.13: Tradeoffs with various versions of the accelerator for $f_{clk} = 250$MHz. The main parameters are I, N_{cycles}, and f_{clk}, where I is number of iteration units per cycle, N_{cycles} is software-programmable number of iterations, and f_{clk} is the clock frequency constraint in the synthesis.

Table 5.12: Comparison of the exponential accelerator synthesised with a complete PE at operating conditions of 0.5 V and 200 MHz clock frequency, to software implementation. Software fixed-point exponential function speed is reported by Partzsch et al. [59] and floating-point result is obtained by running `expf()` function from the GCC `math.h` library on the ARM M4F. Yan et al. [21] report even higher numbers of these functions tested in software on a SpiNNaker2 prototype (106 for fixed-point and 163 for floating-point). Power estimates shown are of the whole PE in all cases. $*$ - includes 2 cycles for reading and writing operations.

	Exp accelerator	Fixed-point soft. exp	binary32 soft. exp
Latency	7–22 cycles/exp*	95 cycles/exp	120 cycles/exp
Throughput	9–28.6M exp/s	2.1M exp/s	1.7M exp/s
Energy per exp	0.3–0.93 nJ/exp	4.43 nJ/exp	5.6 nJ/exp
Total area	6519 µm^2	-	-

Table 5.13: Comparison of the logarithm accelerator synthesised with a complete PE at operating conditions of 0.5 V and 200 MHz clock frequency, to software implementation. Floating-point logarithm function speed is obtained by running `logf()` function from the GCC `math.h` library on the ARM M4F. $*$ - includes 2 cycles for reading and writing operations.

	Log accelerator	binary32 soft. log
Latency	7–22 cycles/exp*	140 cycles/log
Throughput	9–28.6M log/s	1.4M log/s
Energy per log	0.3–0.93 nJ/log	6.5 nJ/log
Total area	6519 µm^2	-

advantage by providing options for controlling the energy and accuracy of the accelerator as well as providing different input/output formats. Additionally, the presented implementation achieves a strictly increasing, or monotonic, function — unfortunately none of the designs, that are compared with, report this property.

However, by using the iterative algorithm (which makes it hard to introduce pipelined operation) and introducing more control of the unit we pay in throughput compared to other designs available. However, since there is no use case where a large number of function invocations is needed one after the other, the lack of pipelining support is not a major downside. Figure 5.14 shows the layout of a single PE after place and route, with the exp-log accelerator highlighted.

Figure 5.14: Layout of a PE after place and route. Cells marked ...*macro* bundled at the north-west corner are the local memory. The rest of the cells at the south-east corner belong to an ARM M4F based PE. Out of that, cells highlighted in white belong to the exp-log accelerator with $I = 2$ iterations. Picture provided by Stefan Scholze.

Table 5.14: Comparison of the exponential function accelerator synthesised with a complete PE at operating conditions of 0.5 V and 200 MHz clock frequency, to similar designs. $*$ — accelerator only (no processor involved); † — includes logarithm function in the same design.

	This work†	[59]	[141]	[145]
Technology	22 nm	28 nm	FPGA	65nm
Throughput	9M–28.6M exp/s	83M exp/s	4.4M exp/s	24.8M exp/s
Pipelined	No	Yes	No	No
Energy per exp	0.3–0.93 nJ/exp	0.44 nJ/exp	-	0.002 nJ/exp*
Format	fixed & float	fixed	float	fixed
Monotonic	Yes	-	-	-
Accuracy control	Yes	No	No	No
Multi-format	Yes	No	No	No
Area	6519 μm^2	10 800 μm^2	-	20 700 μm^2

Table 5.15: Comparison of the logarithm function accelerator synthesised with a complete PE at operating conditions of 0.5 V and 250 MHz clock frequency, to similar designs. † — includes exponential function in the same design. ‡ — average from 4 presented results on 2 different FPGAs.

	This work†	[141]	[142]
Technology	22 nm	FPGA	FPGA
Throughput	9M–28.6M log/s	5.5M log/s	20.2M log/s‡
Pipelined	No	No	Yes
Energy per exp	0.3–0.93 nJ/log	-	-
Format	fixed & float	float	float
Monotonic	Yes	-	-
Accuracy control	Yes	No	No
Multi-format	Yes	No	No
Area	6519 μm^2	-	-

5.5 Testing in silicon

A version of the accelerator containing only the fixed-point formats [126] was tested in JIB1 prototype chip manufactured in 22nm technology. Due to time constraints, only some quick tests could be run; specifically, multiple invocations of the exponential and logarithm functions were performed and confirmed that the expected numbers are returned (as in Verilog testing). Figure 5.15 shows the board with four prototype JIB1 chips on which the accelerator was tested. Each JIB1 chip contains 8 PEs [63].

Figure 5.15: Photo of the board containing four JIB1 chips (photo provided by Sebastian Höppner).

5.6 Previous work

Most of the elementary function accelerator designs available in the literature are designed and evaluated in the context of FPGAs, taking into account different resources available on different FPGAs. For example, some FPGAs contain DSP blocks to do certain operations in various precisions and numerical formats, and designers usually focus on how to optimize the usage of those blocks. As not many designs are available in the literature that were evaluated in an environment similar to ours, that is, 22nm technology with different cell types available, an exhaustive comparison of the presented accelerator to the works in literature cannot be performed.

Another difference is that the authors try to run the FPGA designs as fast as possible, so any given function is evaluated in terms of the maximum timing-clean speed that can be achieved. However, in this case there is no such freedom because if we try to speed up the circuit, more expensive faster cells will be used and power consumption will increase.

Furthermore, it is very hard to find any designs that include multiple numerical formats, and both exponential and logarithm functions in one design sharing hardware resources. Finally, accuracy control is even more unpopular and is not widely explored yet, with an approach of meeting 1ulp accuracy as dictated by the standards usually taken without considering including some control for reduced accuracy and faster operation. Monotonicity is also not checked in any of the designs found in the literature. Table 5.14 provides some comparison of the area/energy/latency to other

designs. In this section some papers are reviewed in more detail, focusing on the algorithmic choices and tradeoffs and compared to the choices made in the current study.

In all likelihood state-of-the-art design of the binary32 floating-point exponential and logarithm functions is published by Langhammer and Pasca [142]. The main goal of the paper is to evaluate the best way to use the resources of two FPGA models (which include for example binary32 floating-point adders and multipliers on the board) for implementing the two elementary functions, and the main goal is to utilize DSP and memory blocks as much as possible and use less FPGA logic.

Range reduction of the logarithm function is performed in a similar fashion to the work presented in this thesis, but the main part calculating the logarithm in the reduced range uses Taylor expansion. Additionally, there is a second level of range reduction for when the argument is not close enough to 1 and the Taylor series cannot be used. The core part of the Taylor expansion uses three multipliers and two adders. For range reduction ($k \times \log(2)$, where k is an 8-bit exponent if the input argument is normalized), a 256x40bit table is used, but the authors mention that using a multiplier by constant is also an option. In total, three LUTs, 4 multipliers (only one multiplies by a constant) and 7 adders have been used.

Exponential is similarly performed using Taylor series, after appropriate range reduction steps are taken. Due to the multipliers and LUTs used, this design would probably result in a very large chip area and leakage power. However, the main advantage is that a pipelined operation can be used and a high throughput of exp/log operations can be obtained. The authors also report the accuracy of 3ulp to meet the OpenCL standard [143]. There was no comment on whether the hardware resources can be shared between the functions, but judging from the architectural diagrams, hardware for doing Taylor series is different for exponential and logarithm, and the infrastructure for range reduction is also very different for each function (which is the case in general) and therefore the designs are almost completely decoupled from each other, causing more circuit area utilization if both functions are wanted. Finally, a wide array of different versions of the accelerators are synthesised — exponential has the latency between 20–34 clock cycles of various different frequencies and the logarithm has 31–44 clock cycles, again for various different clock frequencies that were meeting timing constraints.

Another relatively recent paper exploring floating-point exponential on FPGAs is [146]. The algorithm performs two levels of range reduction until the input is small enough to do a Taylor series expansion and obtain the exponential. The design again,

as the design by Langhammer and Pasca [142], uses a number of multipliers, and is optimized for high throughput of functions and deep pipelining without considering the total logic area and power that would result if implemented as a circuit from scratch, rather than an FPGA. The design is part of the FloPoCo arithmetic hardware generation tool [147] which can be used to generate VHDL designs for specified accuracy and pipeline depth. For a design that is generated with FloPoCo targeting binary32 floating-point number format, the authors report a 3-23 cycle latency for clock frequencies between 180-500mhz. Finally, this design has 1ulp accuracy.

5.7 Trading off processors per chip for accelerators

In other technologies, such as FPGAs, various tradeoffs are explored for optimizing the hardware given some set of target applications, allowing to generate microprocessor-based systems that are smaller, lower power, and faster on those target applications. For example, Wold et al. [148] explored when should certain instructions from processors be removed and emulated in hardware, or alternatively, when should highly used software functions be designed in hardware, which extends the instruction set of the processor. An approach taken in this thesis, to optimize certain numerical functions is similar. However, one tradeoff dimension is interesting to consider in SpiNNaker, that of accelerators in each processor per chip versus using the chip area that accelerators occupy for adding more processors per chip — which approach allows to simulate larger SNNs?

It is useful to consider Amdahl's law in trying to maximize the size of networks that SpiNNaker2 can simulate. There are two main approaches for doing that.

- We have a neural network with the goal to simulate it efficiently. We can assume that we have theoretically infinite parallelism (1 million cores on the full SpiNNaker machine or 10 million on SpiNNaker2). We start by allocating an arbitrary number of neurons to each core, run the application and find that it is too slow. Next start exploiting more parallelism of which we have infinite resources, and in SpiNNaker this is done by allocating less neurons per core, and using more cores. By going this route we eventually hit Amdahl's law, so that the serial part of the SpiNNaker software is not allowing further speedup. This is the point when accelerators on the serial part of the simulation (updating a synapse for example) can push the performance further — adding more cores per chip will not impact how far performance can be pushed in this case.

- We have a machine with a limited number of cores and we want to simulate as many neurons/synapses as possible in that machine. In this case, we try to find the maximum number of neurons/synapses that each core can simulate in some time constraint. It is easy to see that if we speed up parts of the algorithm for simulating each neuron by accelerating some highly used functions, more neurons can be evaluated by each core in a given time.

A benchmark with a complex plasticity rule at the moment is a structural plasticity algorithm explored by Yan et al. [21]. The authors show that approximately 43% of processor cycles available in one simulation timestep are spent in software exponential function which is called for each synapse update. Following the classification methods explored by Wold et al. [148] this function therefore would be classified as frequently used and therefore best to be done in hardware. Calculating exponentials in hardware allowed the number of synapses that each core can simulate to increase from 1900 to 4100 in [21], which means more than 100% increase per chip (roughly extra 19000 synapses, assuming 100 cores per chip). This can further be pushed by trading off accuracy for speed which is programmable in the accelerators presented in this thesis.

Adding more cores per chip instead of the accelerators would not provide substantial improvement in this case: assuming that the exponential accelerators use 2% of the chip area, roughly 2 more processors could be added (assuming 100 processors per chip) and this would result in only $2 \times 1900 = 3600$ extra synapses per chip. Furthermore, the complexity of plasticity rules can potentially increase further in the future applications of SpiNNaker, requiring for example multiple exponentials per update. However, it is worth to note that, as discussed by Wold et al. [148], in the applications where exponential function is classified as having infrequent usage, accelerators would not provide much advantage, and having more cores per chip would be a better approach to go for.

5.8 Conclusion

In this chapter an elementary function accelerator for exponential and natural logarithm functions, with accuracy control and multiple input/output formats was presented, which can be adapted to specific designs with various constraints by controlling how many iteration modules are run per clock cycle. The design considerations come down to a 4-dimensional problem of optimising *power(leakage)-area-accuracy-latency*; synthesis results were shown covering some of the versions of the accelerator

in this space. Low energy consumption and additional options for reducing it further by controlling the accuracy of the unit will provide a suitable elementary function accelerator for neuromorphic applications, where low power is the main priority and full numerical accuracy is most likely not required in most applications.

The contributions of this chapter are as follows.

- A working choice of d_n for the carry-save shift-add algorithm for evaluating the logarithm function (Section 5.2.3) was provided. An extension of the shift-add algorithm is presented by Muller [16], where he covers exponential for signed-digit and carry-save representations, and logarithm in signed-digit representation only, without showing carry-save implementation and not mentioning why it was not included. Therefore this contribution is most likely new.
- A C model of the shift-add algorithms was developed (Appendix C).
- An architecture for a single iteration of the shift-add algorithm was shown, which can be used for computing both exponential and logarithm function.
- Optimization of the look-up table of d_n choice in the shift-add algorithms was shown, which can be used to reduce the size of this table if the ranges of the inputs are smaller than the full convergence domains of the shift-add algorithms. This was also not discussed by Muller [16] and is therefore probably a novel result in that direction.
- A full design of the exp/log accelerator in two fixed-point formats and the binary32 floating-point format with configurable accuracy was presented, and its accuracy and monotonicity were evaluated in a wide array of configurations. A version of the accelerator with fixed-point formats was included in the prototype SpiNNaker2 chip (JIB1) which was manufactured in 22nm technology. The overall extension of the arithmetic hardware, which includes the random number generator [149], the rounding accelerator presented in Chapter 4, and the exponential/logarithm accelerator, is currently estimated to take approximately only 3 % of the area and 7 % of the leakage of one PE of the SpiNNaker2 chip.
- A synthesis study of the various versions of the accelerator was performed using a 22nm cell library. This synthesis study is novel as it does not seem to have ever been performed in the available literature on these elementary function algorithms — usually an approximation of the costs of various parts of these circuits is used to reason about the comparisons.

- The final accelerator with $I = 2$ iterations per clock cycle and all the numerical formats is, at the time of writing, included in the upcoming SpiNNaker2 prototype chip (JIB2) and the final SpiNNaker2 chip.

In terms of further work on this kind of accelerator, the main path would be to experiment with higher-radix implementations of the iterative algorithm [16, 150] to speed up the iterative part of the accelerator — as described by Muller [16, Sec. 10.1], a radix-2^k version of the iterative algorithm implementation presented here would converge in n/k iterations, instead of n when radix-2 is used, to give n-bit accuracy. With a high enough radix, the loop in the main iteration cycle of the module can be removed so that pipelined operation could be implemented.

The difficulty with higher radix algorithms would be larger look-up tables for natural logarithm entries in the L_n iteration; also the choice of d_n value, which in higher radix can adopt more values than $d_n \in \{-1, 0, 1\}$, and on each iteration would result in more sophisticated d_n look-up tables. Following are some more observations from private communications with Miloš Ercegovac and his published research. In the original paper evaluating radix-16 shift-add algorithms, Ercegovac [150] concluded the following ratios for a class of radix-2/radix-16 algorithms: cost 2/3, cycle time 1/3, and latency 4/3. Therefore the advantage of using radix-16 is not that clear, if any. In terms of energy, latency is reduced, but cost per cycle is increasing and therefore some research is required here to understand the energy requirements at higher radices. Piñeiro et al. [151] looked at logarithm, using the shift-add algorithm that was used in this study, with radices between 8 and 1024. The conclusion was that there is no advantage to using very high radices of 512 and 1024 due to exponential growth in the size of look-up tables required with increasing radix. The best option were radices of 128 and 254 for high-speed execution, but not for area. For area it was concluded that radices of 16, 32, and 64 were the most suitable options.

If a radix option could be found that decreases latency from 22 cycles in the presented accelerator without adding a lot of area overhead, it would be a good option for a system similar to SpiNNaker2. The C model could first be extended for easier development and testing of the algorithm and then moving that to Verilog should be straightforward. With that, similar synthesis studies could be done to find any improvement in area and leakage of higher radices in this algorithm.

Further work on evaluation, once the JIB2 prototype chip or the SpiNNaker2 chip is available, would be to explore how accuracy control can impact various neuron and synaptic plasticity models, and how well does a general exponentiation function

implemented with a method that is widely known to be numerically inaccurate (Section 5.4.2) work on the synaptic plasticity models that require it.

5.9 Acknowledgements

The author thanks Stefan Scholze, Sebastian Höppner and Andreas Dixius at the Technical University of Dresden for helping with synthesis and power evaluation, Gengting Liu and Delong Shang for pointing out places that can be optimized in the iteration module, and Prof. Miloš Ercegovac for providing comments about possible improvements of radix-2 shift-and-add algorithms.

Chapter 6

Conclusion

This thesis explores the development of arithmetic accelerators for the second version of the digital neuromorphic chip SpiNNaker, with the main goal of improving the speed and energy efficiency of spiking neural network simulations. As not all possible accelerators, but only the ones running arithmetic functions, are the focus of this thesis, naturally, numerical accuracy was also addressed and some issues in the current SpiNNaker software were discovered and explored in detail. Here a summary of the contributions of this thesis is given with some possible future directions in this type of research.

6.1 Summary of the research

This thesis has addressed the design of two accelerators for SpiNNaker2, preceded by some numerical accuracy exploration done on the current version of SpiNNaker.

In Chapter 3 the numerical accuracy issues demonstrated by Hopkins and Furber [14] were addressed and it was shown that the major spike lags observed by the authors were due to lack of rounding in the constants and in the fixed-point multiplication routine provided by GCC. Furthermore, mixed-precision arithmetic was employed and it was shown how spike lags can be reduced further. The summary of results was that the 19th spike lag from the Izhikevich neuron stimulated by a constant current was reduced from approximately 60 ms to 0.1 ms when a neuron ODE is solved with RK2 Midpoint solver at $\Delta t = 0.1ms$. The overhead to the speed of the ODE solver is negligible if rounding is not enabled compared to the version used by Hopkins and Furber [14]; if rounding is enabled, the ODE solver is $\sim 1.8\times$ slower — however, considering more than an order of magnitude improvement in numerical accuracy of

the neuron, this price paid in latency is worth it. It is worth noting as well that if saturation would be turned on in Hopkins and Furber [14] version of the ODE solver (which did not use mixed-precision arithmetic), all the presented versions in Chapter 3 would then be faster, since mixed-precision multiplication requires no saturation and no shifting in some cases. Due to numerical error reduction, this work will improve the *reproducibility* of scientific results on SpiNNaker using this neuron model, which has recently been a subject of interest by Trensch et al. [15].

Next, in Chapter 3, it was shown how to improve the accuracy of the exponential decay function $e^{-\frac{\Delta t}{\tau}}$ by using mixed-precision and without modifying the underlying implementation of `expk()` function available as part of the SpiNNaker software stack in s16.15 fixed-point format. This should help address some numerical accuracy issues in the current version of SpiNNaker and the Santos chip (Early SpiNNaker2 prototype) which has a hardware exponential function in s16.15 fixed-point format.

Finally, in Chapter 3 it was shown how a new plasticity rule can be developed on SpiNNaker, which involves three factors and is the first of such complexity implemented on SpiNNaker. The speed and numerical accuracy were evaluated and this work has been used in the development of a classical conditioning experiment on SpiNNaker [11], and is currently being tried in the model of *basal ganglia* on SpiNNaker [152].

Following this, in Chapter 4, stochastic rounding of fixed-point multiplication results was investigated to further improve the spike lag of the Izhikevich neuron model. It appears that the improvements shown in Chapter 3 are not as apparent when the neuron parameters are changed to a FS neuron and when a different ODE solver is used. This variability in results is also evident when floating-point arithmetic is used. However, stochastic rounding was shown to produce a very low spike lag across four different ODE solvers in two different configurations of the Izhikevich neuron.

Later in the chapter, the number of random bits required in stochastic rounding was explored by using the same neuron model in two configurations and four ODE solvers. Spike lag increase was shown for different widths of random number and it was shown that a large number of random bits is not required to achieve good accuracy with SR.

Lastly, informed by these improvements in accuracy by testing the ODE solvers on the current generation SpiNNaker chip with software-implemented SR, a decision was made to build an accelerator to do stochastic rounding faster on SpiNNaker2 in hardware. Since SpiNNaker2 already has a PRNG in hardware, the overhead of adding rounding support is very small. The accelerator is fully configurable in terms of which

bits to round, supports signed and unsigned arithmetic, and can round 64 or 32-bit fixed-point values or binary32 floating-point values to *bfloat16*.

Finally, In Chapter 5 a design of the exponential and logarithm accelerator for SpiNNaker2 was presented. The prototype chip Santos already has an accelerator for exponential [59] but the support for accuracy control and floating-point, as well as the logarithm function were not added there. Some improvements to the underlying shift-add algorithms for calculating exponential and logarithm functions were shown, specifically, a carry-save version of the algorithm for logarithm was derived, and some optimisations for the look-up tables that are used on each iteration of the algorithms were found.

The accuracy of the accelerator was evaluated and most of the functions and numerical formats can achieve 1ulp or 2ulp accuracy, with the exception of logarithm in binary32 floating-point format. However, it was shown that while relative error of 192ulp can happen in a narrow range of inputs, absolute error is very small. Accuracy control was also developed and evaluated for some configurations. Various versions of the accelerator were synthesised in a 22 nm library and a fixed-point version of the accelerator [126] was even tested in silicon, since it was added to the prototype chip JIB1 which was recently manufactured.

Lastly, using exponential and logarithm functions, the general exponentiation function can be developed for some limited range of arguments and it was demonstrated how to do it with an exploration of errors in some specific cases. The general exponentiation function can be useful in power law weight dependence in STDP learning rules as noticed by Knight and Nowotny [70].

6.2 Further work

Here are some suggestions for further work in addition to the suggestions already mentioned in each of the chapters.

First of all, multiple improvements to the default GCC's fixed-point libraries was shown, including rounding and mixed-precision multiplication. These can be used to improve accuracy on other neuron models on SpiNNaker from what was explored in this thesis. Stochastic rounding can also be explored in various parts of the SpiNNaker toolchain, for example plasticity where weights are scaled to different formats and as a result some accuracy in the addition operations can be lost. Additionally, various tradeoffs can be explored to do stochastic rounding faster on SpiNNaker. This can

include random number generation and reusing random numbers for multiple rounding operations, either by using the full random number each time for multiple roundings or using different subsets of bits of each random number.

On SpiNNaker2, it would be interesting to explore the accuracy and speed of neuron models, such as Izhikevich or AdEx, which require using ODE solvers, in fixed-point arithmetic, done in software with and without the SR accelerator, and done in software but using the floating-point hardware of the ARM M4F. The floating-point unit is potentially a very complicated unit and can require more energy than fixed-point arithmetic in conjunction with the SR accelerator, but as indicated by the results in this thesis, it does not necessarily give more accuracy than fixed point with SR. Furthermore, exploration of accuracy-speed-energy tradeoffs of various plasticity models by tuning the accuracy control of the exponential function accelerator would be another possible path to take. For example, the reward-based structural plasticity model implemented by Yan et al. [21] which used floating point but did not require 1ulp accuracy of exponential function.

For future digital neuromorphic chips, various approximation techniques can be added fully into hardware (both general-purpose processors and accelerators), for example underdesigning the floating-point and integer components so that they can have small probabilities of error in them. A good example is a multiplier by Kulkarni et al. [153]. However, such techniques, called *approximate computing*, in general, have first to be done in an FPGA or first simulated in software to measure the impact on a wide array of target SNN applications. This would allow minimization of logic area and more PEs on the chip as well as minimize energy per operation. Additionally, approximate computing can be explored at the spiking neural network level as shown by Sen et al. [154] who designed a framework for approximating neuron behaviour by ignoring selected neuron updates believed not to impact the neuron based on various statistics such as average spiking rates or membrane potentials of the neurons. This improves both compute and memory energy, since less synaptic data has to be fetched and less computation done in updating the neurons.

6.3 Overall summary

In summary, the numerical accuracy results presented in this thesis will improve various current and future models of the current version of SpiNNaker, which usually suffers from the lack of support for floating-point arithmetic and has to simulate real

numbers using fixed-point arithmetic. On the other hand, the accelerators developed for SpiNNaker2 as part of this thesis should further improve both the numerical accuracy and the energy of the simulations as initial experiments in developing complex plasticity rules performed by Yan et al. [21] indicate. Finally, the results on reducing the numerical error of ODE solvers using stochastic rounding (which generally seems to be a very novel result, even if just demonstrated experimentally in this thesis) indicate a lot of potential for further research in this space, perhaps experimenting with stochastic rounding in scientific computing algorithms for a wider array of applications than was done here. If stochastic rounding can be shown to provide further improvements in a lot of algorithms and in reduced-precision floating-point arithmetic, in all likelihood it would be a useful addition to the next generation of the IEEE numerical standards and arithmetic hardware of the future processors — a potentially good direction to take for future research.

Acronyms

AHB Advanced High-Performance Bus. 104, 106, 116

CLZ Count Leading Zeros. 127, 129

CORDIC COordinate Rotation DIgital Computer. 116

CPU Central Processing Unit. 24, 25, 39–41, 48, 49

DA Dopamine. 78, 81

DMA Direct-Memory-Access Controller. 26, 42, 46, 47

DSP Digital Signal Processing. 47, 101, 115, 155, 156

DTCM Data Tightly-Coupled Memory. 41, 42, 78

FPGA Field-Programmable Gate Array. 25, 47–49, 101, 136, 154–157, 166

FPU Floating-Point Unit. 23, 26, 47, 52, 104, 110, 115

FS Fast Spiking. 97, 99–101, 164

GCC GNU Compiler Collection. 31, 52, 53, 57, 58, 60, 61, 63, 64, 88, 108, 129, 137, 152, 163, 165

GPU Graphics Processing Unit. 25, 47–49

HPC High-Performance Computing. 44

ISO International Organization for Standardization. 31, 102, 136

ITCM Instruction Tightly-Coupled Memory. 41, 42

LIF Leaky Integrate and Fire. 40, 44, 81, 111

LSB Least Significant Bit. 30, 35, 129, 138, 141

LTD Long-Term Depression. 69, 70

LTP Long-Term Potentiation. 69, 70

LUT Look-Up Table. 65, 76, 78–80, 82, 114, 133, 156

LVT Low-Voltage Threshold. 106, 146, 147

MSB Most Significant Bit. 37, 61, 128

ODE Ordinary Differential Equation. 11, 27, 52–64, 84, 85, 87, 88, 90, 92, 96, 97, 99–103, 108, 110–112, 163, 164, 166, 167

PE Processing Element. 25, 47, 108, 109, 149, 152–154, 159, 166

PRNG Psuedo-random Number Generator. 37, 85, 91, 92, 95, 104, 106, 109, 110, 164

RD Round Down. 37, 38, 93, 94, 97, 99–102, 110

RK Runge-Kutta. 52–57, 59–64, 84, 88, 99, 100, 110, 163

RN Round-to-Nearest. 37, 64, 93, 94, 97, 99, 102, 110, 112

RS Regular Spiking. 54, 84, 97, 99–101

RZ Round-towards-Zero. 38

SLVT Super-Low-Voltage Threshold. 106, 146, 147

SNN Spiking Neural Network. 11, 19–24, 27, 41, 44, 47, 48, 113, 144, 157, 166

SR Stochastic Rounding. 37, 87–95, 99–103, 106, 110–112, 164, 166

STDP Spike-Timing-Dependent Plasticity. 20, 39, 45, 46, 48, 51, 67–74, 78, 79, 81–83, 90, 113–116, 144, 145, 165

ulp Unit of Least Precision or Unit in the Last Place. 29, 35, 36, 129, 141, 142

Bibliography

[1] H. Markram, K. Meier, T. Lippert, S. Grillner, R. Frackowiak, S. Dehaene, A. Knoll, H. Sompolinsky, K. Verstreken, J. DeFelipe, S. Grant, J.-P. Changeux, and A. Saria. Introducing the Human Brain Project. *Procedia Computer Science*, 7:39 – 42, 2011. ISSN 1877-0509. doi: 10.1016/j.procs.2011.12.015. URL http://www.sciencedirect.com/science/article/pii/S1877050911006806. Proceedings of the 2nd European Future Technologies Conference and Exhibition 2011. [p. 20.]

[2] C. Mead. Neuromorphic electronic systems. *Proceedings of the IEEE*, 78(10):1629–1636, Oct. 1990. ISSN 0018-9219. doi: 10.1109/5.58356. [p. 20.]

[3] C. S. Thakur, J. L. Molin, G. Cauwenberghs, G. Indiveri, K. Kumar, N. Qiao, J. Schemmel, R. Wang, E. Chicca, J. Olson Hasler, J.-s. Seo, S. Yu, Y. Cao, A. van Schaik, and R. Etienne-Cummings. Large-scale neuromorphic spiking array processors: A quest to mimic the brain. *Frontiers in Neuroscience*, 12(891), Dec. 2018. ISSN 1662-453X. doi: 10.3389/fnins.2018.00891. URL https://www.frontiersin.org/article/10.3389/fnins.2018.00891. [pp. 20 and 39.]

[4] S. B. Furber. Large-scale neuromorphic computing systems. *Journal of Neural Engineering*, 13(5), Aug. 2016. ISSN 1741-2560. doi: 10.1088/1741-2560/13/5/051001. URL http://stacks.iop.org/1741-2552/13/i=5/a=051001. [p. 20.]

[5] E. Painkras, L. A. Plana, J. D. Garside, S. Temple, F. Galluppi, C. Patterson, D. R. Lester, A. D. Brown, and S. B. Furber. SpiNNaker: A 1-W 18-core system-on-chip for massively-parallel neural network simulation. *IEEE Journal of Solid-State Circuits*, 48(8):1943–1953, Aug. 2013. ISSN 0018-9200. doi: 10.1109/JSSC.2013.2259038. [pp. 20 and 46.]

[6] S. B. Furber, D. R. Lester, L. A. Plana, J. D. Garside, E. Painkras, S. Temple, and A. D. Brown. Overview of the SpiNNaker System Architecture. *IEEE Transactions on Computers*, 62(12):2454–2467, Dec. 2013. ISSN 0018-9340. doi: 10.1109/TC.2012.142. [pp. 20 and 42.]

[7] S. B. Furber, F. Galluppi, S. Temple, and L. A. Plana. The SpiNNaker Project. *Proceedings of the IEEE*, 102(5):652–665, May 2014. ISSN 0018-9219. doi: 10.1109/JPROC.2014.2304638. [pp. 20 and 42.]

[8] D. E. Feldman. The spike-timing dependence of plasticity. *Neuron*, 75(4):556 – 571, Aug. 2012. ISSN 0896-6273. doi: 10.1016/j.neuron.2012.08.001. [pp. 20 and 69.]

[9] J. C. Knight, P. J. Tully, B. A. Kaplan, A. Lansner, and S. B. Furber. Large-scale simulations of plastic neural networks on neuromorphic hardware. *Frontiers in Neuroanatomy*, 10(37), Apr. 2016. ISSN 1662-5129. doi: 10.3389/fnana.2016.00037. URL `https://www.frontiersin.org/articles/10.3389/fnana.2016.00037/full`. [pp. 21, 45, 114, and 115.]

[10] S. J. van Albada, A. G. Rowley, J. Senk, M. Hopkins, M. Schmidt, A. B. Stokes, D. R. Lester, M. Diesmann, and S. B. Furber. Performance comparison of the digital neuromorphic hardware SpiNNaker and the neural network simulation software NEST for a full-scale cortical microcircuit model. *Frontiers in Neuroscience*, 12(291), May 2018. ISSN 1662-453X. doi: 10.3389/fnins.2018.00291. URL `https://www.frontiersin.org/article/10.3389/fnins.2018.00291`. [pp. 21 and 22.]

[11] M. Mikaitis, G. Pineda García, J. C. Knight, and S. B. Furber. Neuromodulated synaptic plasticity on the SpiNNaker neuromorphic system. *Frontiers in Neuroscience*, 12(105), Feb. 2018. ISSN 1662-453X. doi: 10.3389/fnins.2018.00105. URL `https://www.frontiersin.org/article/10.3389/fnins.2018.00105`. [pp. 21, 51, 69, 78, 79, 80, 82, 115, and 164.]

[12] S. Friedmann, J. Schemmel, A. Grübl, A. Hartel, M. Hock, and K. Meier. Demonstrating hybrid learning in a flexible neuromorphic hardware system. *IEEE Transactions on Biomedical Circuits and Systems*, 11(1):128–142, Feb. 2017. ISSN 1932-4545. doi: 10.1109/TBCAS.2016.2579164. [pp. 21 and 39.]

[13] M. Davies, N. Srinivasa, T. H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C. K. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y. H. Weng, A. Wild, Y. Yang, and H. Wang. Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro*, 38(1):82–99, Jan. 2018. ISSN 0272-1732. doi: 10.1109/MM.2018.112130359. [pp. 21, 40, 41, and 90.]

[14] M. Hopkins and S. B. Furber. Accuracy and efficiency in fixed-point neural ODE solvers. *Neural Computation*, 27(10):2148–2182, Sept. 2015. ISSN 1530888X. doi: 10.1162/NECO_a_00772. [pp. 22, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 87, 88, 96, 97, 99, 100, 108, 112, 113, 163, and 164.]

[15] G. Trensch, R. Gutzen, I. Blundell, M. Denker, and A. Morrison. Rigorous neural network simulations: A model substantiation methodology for increasing the correctness of simulation results in the absence of experimental validation data. *Frontiers in Neuroinformatics*, 12(81), Nov. 2018. ISSN 1662-5196. doi: 10.3389/fninf.2018.00081. URL `https://www.frontiersin.org/article/10.3389/fninf.2018.00081`. [pp. 22, 53, 58, 59, 62, 88, 99, 108, and 164.]

[16] J.-M. Muller. *Elementary Functions: Algorithms and Implementation*. Birkhäuser, Boston, MA, US, 3rd edition, 2016. ISBN 978-1-4899-7983-4. doi: 10.1007/978-1-4899-7983-4. [pp. 23, 29, 115, 116, 117, 118, 119, 120, 121, 122, 130, 132, 133, 137, 145, 159, and 160.]

[17] V. Innocente. Floating point in experimental HEP data processing (aka reconstruction). Online (accessed: 25/3/20) `https://indico.cern.ch/event/202688/contributions/1487953/attachments/305609/426817/FPinEHEP.pdf`, Sep. 2012. [p. 23.]

[18] D. Piparo. The VDT mathematical library: A modern reimplementation of cephes. Online (accessed: 25/3/20) `https://indico.cern.ch/event/202688/contributions/1487957/attachments/305612/426820/OpenLabFPWorkshop_9_2012.pdf`, Sep. 2012. [p. 23.]

[19] D. Piparo. Introduction to HEP numerical computing, challenges in data reconstruction and simulation. Online (accessed: 25/3/20) `https://indico.cern.ch/event/313684/contributions/1687767/attachments/600510/826487/OpenlabNumericalComputingWorkshop4thInstance.pdf`, May 2014. [p. 23.]

[20] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. *IEEE Micro*, 35(3):10–22, May 2015. ISSN 0272-1732. doi: 10.1109/MM.2015.42. [p. 23.]

[21] Y. Yan, D. Kappel, F. Neumärker, J. Partzsch, B. Vogginger, S. Höppner, S. Furber, W. Maass, R. Legenstein, and C. Mayr. Efficient reward-based structural plasticity on a SpiNNaker 2 prototype. *IEEE Transactions on Biomedical Circuits and Systems*, 13(3):579–591, Mar. 2019. ISSN 1932-4545. doi: 10.1109/TBCAS.2019.2906401. [pp. 23, 47, 64, 115, 152, 158, 166, and 167.]

[22] IEEE Computer Society. *IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2019 (revision of IEEE Std 754-2008)*. Institute of Electrical and Electronics Engineers, Piscataway, NJ, USA, Jul. 2019. ISBN 978-1-5044-5924-2. doi: 10.1109/IEEESTD.2019.8766229. [pp. 23, 27, 33, 86, and 136.]

[23] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2002. ISBN 0-89871-521-0. doi: 10.1137/1.9780898718027. [pp. 24 and 89.]

[24] X. Jin. *Parallel Simulation of Neural Networks On SpiNNaker Universal Neuromorphic Hardware*. PhD thesis, University of Manchester, School of Computer Science, 2010. [p. 24.]

[25] E. M. Izhikevich. Which model to use for cortical spiking neurons? *IEEE Transactions on Neural Networks*, 15(5):1063–1070, Sep. 2004. ISSN 1045-9227. doi: 10.1109/TNN.2004.832719. [p. 24.]

[26] M. Horowitz. 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, pages 10–14, San Francisco, CA, USA, Feb. 2014. doi: 10.1109/ISSCC.2014.6757323. [pp. 25 and 86.]

[27] T. Palmer. Build imprecise supercomputers. *Nature*, 526(7571):32–33, Sep. 2015. ISSN 0028-0836. doi: 10.1038/526032a. [pp. 25 and 111.]

[28] J. Han and M. Orshansky. Approximate computing: An emerging paradigm for energy-efficient design. In *18th IEEE European Test Symposium*, Avignon, France, May 2013. doi: 10.1109/ETS.2013.6569370. [pp. 25, 86, and 115.]

[29] W. J. Cody. *Software Manual for the Elementary Functions (Prentice-Hall Series in Computational Mathematics)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1980. ISBN 0138220646. [pp. 29 and 145.]

[30] B. Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, Inc., New York, NY, USA, 2000. ISBN 0-19-512583-5. [p. 29.]

[31] M. D. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, Jun. 2003. ISBN 9781558607989. doi: 10.1016/B978-1-55860-798-9.X5000-3. [pp. 29, 116, and 131.]

[32] J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond, N. Revol, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser, Boston, MA, US, 2nd edition, 2018. ISBN 978-3-319-76526-6. doi: 10.1007/978-3-319-76526-6. [pp. 29 and 33.]

[33] M. Hopkins, M. Mikaitis, D. R. Lester, and S. B. Furber. Stochastic rounding and reduced-precision fixed-point arithmetic for solving neural ordinary differential equations. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 378(2166), Jan. 2020. ISSN 1471-2962. doi: 10.1098/rsta.2019.0052. URL https://royalsocietypublishing.org/doi/abs/10.1098/rsta.2019.0052. [pp. 29, 85, 95, 110, and 112.]

[34] ISO/IEC. *Programming languages — C — Extensions to support embedded processors, ISO/IEC TR 18037:2008*. 2nd edition, Jun. 2008. ISBN 978-0580636868. URL https://www.iso.org/standard/51126.html. [pp. 31, 57, 58, 102, 115, and 136.]

[35] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software*, 33(2), Jun. 2007. ISSN 0098-3500. doi: 10.1145/1236463.1236468. URL https://doi.org/10.1145/1236463.1236468. [p. 33.]

[36] Intel. BFLOAT16 – hardware numerics definition. Online (accessed: 13/05/20) https://software.intel.com/sites/default/files/managed/40/8b/bf16-hardware-numerics-definition-white-paper.pdf, Nov. 2018. [pp. 33, 48, and 86.]

[37] J.-M. Muller. On the definition of ulp(x). Technical Report No 5504, INRIA, Feb. 2005. URL https://hal.inria.fr/inria-00070503. [pp. 35, 36, and 141.]

[38] W. Kahan. A logarithm too clever by half. Online (accessed: 13/05/20) https://people.eecs.berkeley.edu/~wkahan/LOG10HAF.TXT, Aug. 2004. [pp. 35 and 36.]

[39] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. Deep learning with limited numerical precision. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning*, volume 37 of *JMLR Workshop and Conference Proceedings*, Lille, France, Jul. 2015. URL http://proceedings.mlr.press/v37/gupta15.pdf. [pp. 37, 86, 89, 101, and 111.]

[40] M. Höhfeld and S. E. Fahlman. Probabilistic rounding in neural network learning with limited precision. *Neurocomputing*, 4(6):291 – 299, 1992. ISSN 0925-2312. doi: 10.1016/0925-2312(92)90014-G. URL http://www.sciencedirect.com/science/article/pii/092523129290014G. [pp. 37 and 89.]

[41] A. F. Tenca, S. Park, and L. A. Tawalbeh. Carry-save representation is shift-unsafe: the problem and its solution. *IEEE Transactions on Computers*, 55(5): 630–635, May 2006. ISSN 0018-9340. doi: 10.1109/TC.2006.70. [p. 38.]

[42] R. Brette and W. Gerstner. Adaptive exponential integrate-and-fire model as an effective description of neuronal activity. *Journal of Neurophysiology*, 94 (5):3637–3642, Nov. 2005. doi: 10.1152/jn.00686.2005. URL https://journals.physiology.org/doi/full/10.1152/jn.00686.2005. PMID: 16014787. [pp. 39 and 115.]

[43] W. Gerstner, W. M. Kistler, R. Naud, and L. Paninski. *Neuronal Dynamics:*

From Single Neurons to Networks and Models of Cognition. Cambridge University Press, New York, NY, USA, Aug. 2014. ISBN 9781107447615. doi: 10.1017/CBO9781107447615. [pp. 39 and 115.]

[44] J. Schemmel, J. Fieres, and K. Meier. Wafer-scale integration of analog neural networks. In *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, pages 431–438, Hong Kong, China, Jun. 2008. doi: 10.1109/IJCNN.2008.4633828. [p. 39.]

[45] J. Schemmel, A. Grubl, K. Meier, and E. Mueller. Implementing synaptic plasticity in a VLSI spiking neural network model. In *The 2006 IEEE International Joint Conference on Neural Networks*, Vancouver, BC, Canada, Jul. 2006. doi: 10.1109/IJCNN.2006.246651. [p. 39.]

[46] S. Friedmann, N. Frémaux, J. Schemmel, W. Gerstner, and K. Meier. Reward-based learning under hardware constraints—using a RISC processor embedded in a neuromorphic substrate. *Frontiers in Neuroscience*, 7(160), Sep. 2013. ISSN 1662-453X. doi: 10.3389/fnins.2013.00160. URL `http://journal.frontiersin.org/article/10.3389/fnins.2013.00160`. [p. 39.]

[47] T. Wunderlich, A. F. Kungl, E. Müller, A. Hartel, Y. Stradmann, S. A. Aamir, A. Grübl, A. Heimbrecht, K. Schreiber, D. Stöckel, C. Pehle, S. Billaudelle, G. Kiene, C. Mauch, J. Schemmel, K. Meier, and M. A. Petrovici. Demonstrating advantages of neuromorphic computation: A pilot study. *Frontiers in Neuroscience*, 13(260), Mar. 2019. ISSN 1662-453X. doi: 10.3389/fnins.2019.00260. URL `https://www.frontiersin.org/article/10.3389/fnins.2019.00260`. [p. 39.]

[48] S. Moradi, N. Qiao, F. Stefanini, and G. Indiveri. A scalable multicore architecture with heterogeneous memory structures for dynamic neuromorphic asynchronous processors (DYNAPs). *IEEE Transactions on Biomedical Circuits and Systems*, 12(1):106–122, Nov. 2017. ISSN 1932-4545. doi: 10.1109/TBCAS.2017.2759700. [p. 40.]

[49] P. Merolla, J. Arthur, F. Akopyan, N. Imam, R. Manohar, and D. S. Modha. A digital neurosynaptic core using embedded crossbar memory with 45pJ per spike in 45nm. In *2011 IEEE Custom Integrated Circuits Conference*, San Jose, CA, USA, Sep. 2011. doi: 10.1109/CICC.2011.6055294. [p. 41.]

[50] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, B. Brezzo, I. Vo, S. K. Esser, R. Appuswamy, B. Taba, A. Amir, M. D. Flickner, W. P. Risk, R. Manohar, and D. S. Modha. A million spiking-neuron integrated circuit with communication network and interface. *Science*, 345(6197):668–673, Aug. 2014. ISSN 0036-8075. doi: 10.1126/science.1254642. URL https://science.sciencemag.org/content/345/6197/668. [p. 41.]

[51] A. G. D. Rowley, C. Brenninkmeijer, S. Davidson, D. Fellows, A. Gait, D. R. Lester, L. A. Plana, O. Rhodes, A. B. Stokes, and S. B. Furber. SpiNNTools: The execution engine for the SpiNNaker platform. *Frontiers in Neuroscience*, 13(231), Mar. 2019. ISSN 1662-453X. doi: 10.3389/fnins.2019.00231. URL https://www.frontiersin.org/article/10.3389/fnins.2019.00231. [pp. 43 and 46.]

[52] A. Davison, D. Brüderle, J. Eppler, J. Kremkow, E. Muller, D. Pecevski, L. Perrinet, and P. Yger. PyNN: a common interface for neuronal network simulators. *Frontiers in Neuroinformatics*, 2(11), Jan. 2009. ISSN 1662-5196. doi: 10.3389/neuro.11.011.2008. URL https://www.frontiersin.org/article/10.3389/neuro.11.011.2008. [pp. 43, 44, and 74.]

[53] O. Rhodes, P. A. Bogdan, C. Brenninkmeijer, S. Davidson, D. Fellows, A. Gait, D. R. Lester, M. Mikaitis, L. A. Plana, A. G. D. Rowley, A. B. Stokes, and S. B. Furber. sPyNNaker: A software package for running PyNN simulations on SpiNNaker. *Frontiers in Neuroscience*, 12(816), Nov. 2018. ISSN 1662-453X. doi: 10.3389/fnins.2018.00816. URL https://www.frontiersin.org/article/10.3389/fnins.2018.00816. [pp. 44, 46, 52, 74, 76, 88, and 112.]

[54] P. U. Diehl and M. Cook. Efficient implementation of STDP rules on SpiNNaker neuromorphic hardware. In *2014 International Joint Conference on Neural Networks*, pages 4288–4295, Beijing, China, Jul. 2014. doi: 10.1109/IJCNN.2014.6889876. [p. 45.]

[55] J. C. Knight and S. B. Furber. Synapse-centric mapping of cortical models to the SpiNNaker neuromorphic architecture. *Frontiers in Neuroscience*, 10(420), Sep. 2016. ISSN 1662-453X. doi: 10.3389/fnins.2016.00420. URL http:

//journal.frontiersin.org/article/10.3389/fnins.2016.00420. [pp. 45, 81, 82, and 114.]

[56] J. C. Knight. *Plasticity in large-scale neuromorphic models of the neocortex*. PhD thesis, University of Manchester, School of Computer Science, 2016. [pp. 45 and 76.]

[57] S. Song, K. D. Miller, and L. F. Abbott. Competitive Hebbian learning through spike-timing-dependent synaptic plasticity. *Nature neuroscience*, 3(9): 919–926, Sep. 2000. ISSN 1097-6256. doi: 10.1038/78829. URL https://www.nature.com/articles/nn0900_919. [pp. 45 and 71.]

[58] A. Morrison, M. Diesmann, and W. Gerstner. Phenomenological models of synaptic plasticity based on spike timing. *Biological Cybernetics*, 98(6):459–478, May 2008. doi: 10.1007/s00422-008-0233-1. [pp. 45, 69, 70, and 71.]

[59] J. Partzsch, S. Höppner, M. Eberlein, R. Schüffny, C. Mayr, D. R. Lester, and S. B. Furber. A fixed point exponential function accelerator for a neuromorphic many-core system. In *2017 IEEE International Symposium on Circuits and Systems*, Baltimore, MD, USA, May 2017. doi: 10.1109/ISCAS.2017.8050528. [pp. 46, 64, 65, 68, 115, 116, 137, 152, 154, and 165.]

[60] S. Höppner, Y. Yan, B. Vogginger, A. Dixius, J. Partzsch, F. Neumärker, S. Hartmann, S. Schiefer, S. Scholze, G. Ellguth, L. Cederstroem, M. Eberlein, C. Mayr, S. Temple, L. A. Plana, J. D. Garside, S. Davison, D. R. Lester, and S. B. Furber. Dynamic voltage and frequency scaling for neuromorphic many-core systems. In *2017 IEEE International Symposium on Circuits and Systems*, Baltimore, MD, USA, May 2017. doi: 10.1109/ISCAS.2017.8050656. [p. 46.]

[61] S. Höppner, B. Vogginger, Y. Yan, A. Dixius, S. Scholze, J. Partzsch, F. Neumärker, S. Hartmann, S. Schiefer, G. Ellguth, L. Cederstroem, L. A. Plana, J. D. Garside, S. B. Furber, and C. Mayr. Dynamic power management for neuromorphic many-core systems. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 66(8):2973–2986, Aug. 2019. ISSN 1549-8328. doi: 10.1109/TCSI.2019.2911898. [pp. 46, 47, and 149.]

[62] ARM. Arm Cortex-M4 processor, technical reference manual, 2015. Revision: r0p1. [pp. 47, 103, and 115.]

[63] S. Höppner and C. Mayr. SpiNNaker2—towards extremely efficient digital neuromorphics and multi-scale brain emulation. In *2018 Neuro-inspired Computional Elements Workshop*, Portland, OR, US, 2018. URL https://niceworkshop.org/wp-content/uploads/2018/05/2-27-SHoppner-SpiNNaker2.pdf. [pp. 47 and 154.]

[64] C. Liu, G. Bellec, B. Vogginger, D. Kappel, J. Partzsch, F. Neumärker, S. Höppner, W. Maass, S. B. Furber, R. Legenstein, and C. G. Mayr. Memory-efficient deep learning on a SpiNNaker 2 prototype. *Frontiers in Neuroscience*, 12(840), Nov. 2018. ISSN 1662-453X. doi: 10.3389/fnins.2018.00840. URL https://www.frontiersin.org/article/10.3389/fnins.2018.00840. [pp. 47 and 64.]

[65] M.-O. Gewaltig and M. Diesmann. NEST (NEural Simulation Tool). *Scholarpedia*, 2(4), 2007. doi: 10.4249/scholarpedia.1430. revision #130182. [p. 48.]

[66] A. L. Hodgkin and A. F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of Physiology*, 117(4):500–544, Aug. 1952. doi: 10.1113/jphysiol.1952.sp004764. URL https://physoc.onlinelibrary.wiley.com/doi/abs/10.1113/jphysiol.1952.sp004764. [p. 48.]

[67] A. Morrison, A. Aertsen, and M. Diesmann. Spike-timing-dependent plasticity in balanced random networks. *Neural Computation*, 19(6):1437–1467, Jun. 2007. doi: 10.1162/neco.2007.19.6.1437. [pp. 48, 71, 144, and 145.]

[68] W. Potjans, A. Morrison, and M. Diesmann. Enabling functional neural circuit simulations with distributed computing of neuromodulated plasticity. *Frontiers in Computational Neuroscience*, 4(141), Nov. 2010. ISSN 1662-5188. doi: 10.3389/fncom.2010.00141. URL http://journal.frontiersin.org/article/10.3389/fncom.2010.00141. [p. 48.]

[69] E. Yavuz, J. Turner, and T. Nowotny. GeNN: a code generation framework for accelerated brain simulations. *Scientific Reports*, 6(18854), Jan. 2016. doi: 10.1038/srep18854. URL https://www.nature.com/articles/srep18854. [p. 48.]

[70] J. C. Knight and T. Nowotny. GPUs outperform current HPC and neuromorphic solutions in terms of speed and energy when simulating a highly-connected

cortical model. *Frontiers in Neuroscience*, 12(941), Dec. 2018. ISSN 1662-453X. doi: 10.3389/fnins.2018.00941. URL https://www.frontiersin.org/article/10.3389/fnins.2018.00941. [pp. 48 and 165.]

[71] D. Pani, P. Meloni, G. Tuveri, F. Palumbo, P. Massobrio, and L. Raffo. An FPGA platform for real-time simulation of spiking neuronal networks. *Frontiers in Neuroscience*, 11(90), Feb. 2017. ISSN 1662-453X. doi: 10.3389/fnins.2017.00090. URL https://www.frontiersin.org/article/10.3389/fnins.2017.00090. [p. 48.]

[72] J. Gustafson and I. Yonemoto. Beating floating point at its own game: Posit arithmetic. *Supercomputing Frontiers and Innovations: an International Journal*, 4(2):71–86, Jun. 2017. ISSN 2409-6008. doi: 10.14529/jsfi170206. [pp. 48 and 86.]

[73] F. de Dinechin, L. Forget, J.-M. Muller, and Y. Uguen. Posits: The good, the bad and the ugly. In *Proceedings of the Conference for Next Generation Arithmetic 2019*, Singapore, Singapore, Mar. 2019. Association for Computing Machinery. doi: 10.1145/3316279.3316285. [p. 48.]

[74] M. Mikaitis. Issues with rounding in the GCC implementation of the ISO 18037:2008 standard fixed-point arithmetic. *arXiv preprint arXiv:2001.01496*, Jan. 2020. URL https://arxiv.org/abs/2001.01496. To appear in the proceedings of the 27th IEEE symposium on computer arithmetic. [pp. 51, 52, and 60.]

[75] E. M. Izhikevich. Simple model of spiking neurons. *IEEE Transactions on Neural Networks*, 14(6):1569–1572, Nov. 2003. ISSN 1045-9227. doi: 10.1109/TNN.2003.820440. [pp. 51, 52, 59, and 97.]

[76] X. Jin, S. B. Furber, and J. V. Woods. Efficient modelling of spiking neural networks on a scalable chip multiprocessor. In *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, pages 2812–2819, Hong Kong, China, Jun. 2008. doi: 10.1109/IJCNN.2008.4634194. [pp. 52, 82, and 108.]

[77] GNU. Options that control optimization. Online: (accessed: 26/03/20) https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html, . [p. 53.]

[78] GNU. Routines for fixed-point fractional emulation. Online (accessed: 14/08/19) https://gcc.gnu.org/onlinedocs/gccint/Fixed-point-fractional-library-routines.html, . [p. 60.]

[79] SpiNNakerManchester. SpiNN common. Online (accessed: 10/09/19) https://github.com/SpiNNakerManchester/spinn_common/blob/master/src/stdfix-exp.c. [pp. 64 and 65.]

[80] P. Ardin, F. Peng, M. Mangan, K. Lagogiannis, and B. Webb. Using an insect mushroom body circuit to encode route memory in complex natural environments. *PLOS Computational Biology*, 12(2), Feb. 2016. doi: 10.1371/journal.pcbi.1004683. [p. 69.]

[81] D. O. Hebb. *The organization of behavior*. Wiley & Sons, New York, NY, US, 1949. ISBN 978-0805843002. [p. 69.]

[82] H. Markram, J. Lübke, M. Frotscher, and B. Sakmann. Regulation of synaptic efficacy by coincidence of postsynaptic APs and EPSPs. *Science*, 275(5297): 213–215, Jan. 1997. doi: 10.1126/science.275.5297.213. [p. 69.]

[83] N. Frémaux and W. Gerstner. Neuromodulated spike-timing-dependent plasticity, and theory of three-factor learning rules. *Frontiers in Neural Circuits*, 9(85), Jan. 2016. ISSN 1662-5110. doi: 10.3389/fncir.2015.00085. URL http://journal.frontiersin.org/article/10.3389/fncir.2015.00085. [p. 69.]

[84] G.-q. Bi and M.-m. Poo. Synaptic modifications in cultured hippocampal neurons: Dependence on spike timing, synaptic strength, and postsynaptic cell type. *Journal of Neuroscience*, 18(24):10464–10472, Dec. 1998. doi: 10.1523/JNEUROSCI.18-24-10464.1998. [p. 70.]

[85] J. Rubin, D. D. Lee, and H. Sompolinsky. Equilibrium properties of temporally asymmetric Hebbian plasticity. *Physical Review Letters*, 86:364–367, Jan. 2001. doi: 10.1103/PhysRevLett.86.364. [p. 71.]

[86] J.-P. Pfister and W. Gerstner. Triplets of spikes in a model of spike timing-dependent plasticity. *The Journal of Neuroscience*, 26(38):9673–82, Sep. 2006. ISSN 1529-2401. doi: 10.1523/JNEUROSCI.1425-06.2006. [p. 71.]

[87] C. Clopath, L. Büsing, E. Vasilaki, and W. Gerstner. Connectivity reflects coding: a model of voltage-based STDP with homeostasis. *Nature neuroscience*, 13(3):344–352, Mar. 2010. ISSN 1546-1726. doi: 10.1038/nn.2479. [pp. 71 and 114.]

[88] W. Schultz. Multiple reward signals in the brain. *Nature Reviews Neuroscience*, 1(3):199–207, Dec. 2000. doi: 10.1038/35044563. [p. 71.]

[89] W. Gerstner, M. Lehmann, V. Liakoni, D. Corneil, and J. Brea. Eligibility traces and plasticity on behavioral time scales: Experimental support of neoHebbian three-factor learning rules. *Frontiers in Neural Circuits*, 12(53), Jul. 2018. ISSN 1662-5110. doi: 10.3389/fncir.2018.00053. URL `https://www.frontiersin.org/article/10.3389/fncir.2018.00053`. [p. 71.]

[90] E. M. Izhikevich. Solving the distal reward problem through linkage of STDP and dopamine signaling. *Cerebral Cortex*, 17(10):2443—2452, Oct. 2007. doi: doi:10.1093/cercor/bhl152. [pp. 71 and 72.]

[91] R. V. Florian. Reinforcement learning through modulation of spike-timing-dependent synaptic plasticity. *Neural Computation*, 19(6):1468–1502, Apr. 2007. doi: 10.1162/neco.2007.19.6.1468. URL `https://www.mitpressjournals.org/doi/10.1162/neco.2007.19.6.1468`. [p. 71.]

[92] E. Nichols, B. Gardner, and A. Gruning. Supervised learning on the SpiNNaker neuromorphic hardware. Jul. 2017. doi: 10.5281/zenodo.823273. Technical report, University of Surrey. [pp. 73 and 76.]

[93] C. Beaulieu and M. Colonnier. Number and size of neurons and synapses in the motor cortex of cats raised in different environmental complexities. *Journal of Comparative Neurology*, 289(1):178–187, Nov. 1989. doi: 10.1002/cne.902890115. [p. 82.]

[94] B. Pakkenberg, D. Pelvig, L. Marner, M. J. Bundgaard, H. J. G. Gundersen, J. R. Nyengaard, and L. Regeur. Aging and the human neocortex. *Experimental Gerontology*, 38(1–2):95 – 99, Jan. 2003. ISSN 0531-5565. doi: 10.1016/S0531-5565(02)00151-1. [p. 82.]

[95] V. Braitenberg and A. Schüz. *Cortex: statistics and geometry of neuronal connectivity*. Springer, Berlin, Heidelberg, 1998. ISBN 978-3-662-03735-5. doi: 10.1007/978-3-662-03733-1. [p. 82.]

[96] G. Buzsáki and K. Mizuseki. The log-dynamic brain: how skewed distributions affect network operations. *Nature Reviews Neuroscience*, 15:264–278, Feb. 2014. doi: 10.1038/nrn3687. URL https://www.nature.com/articles/nrn3687. [p. 82.]

[97] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the ACM/IEEE 44th Annual International Symposium on Computer Architecture*, Toronto, ON, Canada, Jun. 2017. IEEE. ISBN 978-1-4503-4892-8. doi: 10.1145/3079856.3080246. [p. 86.]

[98] J. Johnson. Rethinking floating point for deep learning. *arXiv preprint arXiv:1811.01721*, Nov. 2018. URL https://arxiv.org/abs/1811.01721. [p. 86.]

[99] R. Morris. Tapered floating point: A new floating-point representation. *IEEE Transactions on Computers*, C-20(12):1578–1579, Dec. 1971. ISSN 0018-9340. doi: 10.1109/T-C.1971.223174. [p. 86.]

[100] W. Dally. High-performance hardware for machine learning. Online (accessed: 25/3/20) https://berkeley-deep-learning.github.io/cs294-dl-f16/slides/DL_HW_Berkeley_0916.pdf. [p. 86.]

[101] P. Micikevicius, J. Alben, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, H. Wu, S. Narang, G. Diamos, and E. Elsen. Mixed precision training. In *Proceedings of the 6th International Conference on Learning Representations*, Vancouver, BC, Canada, 2018. URL https://openreview.net/forum?id=r1gs9JgRZ. [p. 86.]

[102] J. Vanderkooy and S. P. Lipshitz. Dither in digital audio. *Journal of the Audio Engineering Society*, 35(12):966–975, Dec. 1987. URL http://www.aes.org/e-lib/browse.cfm?elib=5173. [p. 87.]

[103] J. Vanderkooy and S. P. Lipshitz. Digital dither: Processing with resolution far below the least significant bit. In *7th International Conference: Audio in Digital Times*, Toronto, Canada, May 1989. URL http://www.aes.org/e-lib/browse.cfm?elib=5482. [p. 87.]

[104] J. Miao, K. He, A. Gerstlauer, and M. Orshansky. Modeling and synthesis of quality-energy optimal approximate adders. In *2012 IEEE/ACM International Conference on Computer-Aided Design*, pages 728–735, San Jose, CA, USA, Nov. 2012. doi: 10.1145/2429384.2429542. [p. 87.]

[105] G. Forsythe. Reprint of a note on rounding-off errors. *SIAM Review*, 1(1):66–67, 1959. doi: 10.1137/1001011. URL https://epubs.siam.org/doi/10.1137/1001011. [p. 89.]

[106] N. S. Scott, F. Jézéquel, C. Denis, and J.-M. Chesneaux. Numerical 'health check' for scientific codes: The CADNA approach. *Computer Physics Communications*, 176(8):507–521, Apr. 2007. doi: 10.1016/j.cpc.2007.01.005. [p. 89.]

[107] R. Alt and J. Vignes. *Stochastic Arithmetic as a Model of Granular Computing*, pages 33–54. Wiley, New York, 2008. ISBN 978-0-470-03554-2. doi: 10.1002/9780470724163.ch2. [p. 89.]

[108] L. K. Müller and G. Indiveri. Rounding methods for neural networks with low resolution synaptic weights. *arXiv preprint arXiv:1504.05767*, Apr. 2015. URL https://arxiv.org/abs/1504.05767. [p. 89.]

[109] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. In *Advances in Neural Information Processing Systems 31*, Montréal, Canada, Dec. 2018. [p. 90.]

[110] T. Gokmen, M. J. Rasch, and W. Haensch. Training LSTM networks with resistive cross-point devices. *Frontiers in Neuroscience*, 12(745), Oct. 2018. ISSN 1662-453X. doi: 10.3389/fnins.2018.00745. URL https://www.frontiersin.org/article/10.3389/fnins.2018.00745. [p. 90.]

[111] N. J. Higham and S. Pranesh. Simulating low precision floating-point arithmetic. *SIAM Journal on Scientific Computing*, 41(5):C585–C602, Oct. 2019. doi: 10.1137/19M1251308. [pp. 94 and 95.]

[112] D. Malone. To what does the harmonic series converge? *Irish Mathematical Society Bulletin*, (71):59–66, May 2013. ISSN 0791-5578. URL http://mural.maynoothuniversity.ie/6008/. [p. 94.]

[113] P. Blanchard, N. J. Higham, and T. Mary. A class of fast and accurate summation algorithms. *SIAM Journal on Scientific Computing*, 42(3):A1541–A1557, May 2020. doi: 10.1137/19M1257780. [p. 94.]

[114] G. Marsaglia and A. Zaman. The KISS generator. Technical report, Department of Statistics, Florida State University, Tallahassee, FL, USA, 1993. [p. 95.]

[115] P. L'Ecuyer and R. Simard. TestU01: A C library for empirical testing of random number generators. *ACM Transactions on Mathematical Software*, 33(4), Aug. 2007. ISSN 0098-3500. doi: 10.1145/1268776.1268777. [p. 95.]

[116] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, USA, 2nd edition, 1992. ISBN 0521431085. [p. 95.]

[117] J. C. Butcher. *Numerical Methods for Ordinary Differential Equations*. John Wiley & Sons, 3rd edition, 2016. ISBN 9781119121503. doi: 10.1002/9781119121534. [p. 97.]

[118] R. P. K. Chan and A. Y. J. Tsai. On explicit two-derivative Runge-Kutta methods. *Numerical Algorithms*, 53:171–194, Nov. 2009. doi: 10.1007/s11075-009-9349-1. [p. 97.]

[119] M. Mikaitis. Stochastic rounding: Algorithms and hardware accelerator. *arXiv preprint arXiv:2001.01501*, Jan. 2020. URL https://arxiv.org/abs/2001.01501. [p. 103.]

[120] ARM. Cortex-M4 Devices, Generic User Guide, 2010. [p. 103.]

[121] Online. URL http://www.makechip.design. [pp. 106, 116, and 145.]

[122] R. Carter, J. Mazurier, L. Pirro, J. U. Sachse, P. Baars, J. Faul, C. Grass, G. Grasshoff, P. Javorka, T. Kammler, A. Preusse, S. Nielsen, T. Heller, J. Schmidt, H. Niebojewski, P. Y. Chou, E. Smith, E. Erben, C. Metze, C. Bao, Y. Andee, I. Aydin, S. Morvan, J. Bernard, E. Bourjot, T. Feudel, D. Harame, R. Nelluri, H. J. Thees, L. M-Meskamp, J. Kluth, R. Mulfinger, M. Rashed, R. Taylor, C. Weintraub, J. Hoentschel, M. Vinet, J. Schaeffer, and B. Rice. 22nm FDSOI technology for emerging mobile, internet-of-things, and RF applications. In *2016 IEEE International Electron Devices Meeting*, San Francisco, CA, USA, Dec. 2016. doi: 10.1109/IEDM.2016.7838029. [pp. 106, 116, and 145.]

[123] S. Höppner, H. Eisenreich, D. Walter, U. Steeb, A. S. Clifford Dmello, R. Sinkwitz, H. Bauer, A. Oefelein, F. Schraut, J. Schreiter, R. Niebsch, S. Scherzer, U. Hensel, J. Winkler, and M. Orgis. How to achieve world-leading energy efficiency using 22FDX with adaptive body biasing on an Arm Cortex-M4 IoT SoC. In *49th European Solid-State Device Research Conference*, pages 66–69, Cracow, Poland, Sep. 2019. doi: 10.1109/ESSDERC.2019.8901768. [pp. 106 and 146.]

[124] S. Höppner, H. Eisenreich, D. Walter, A. Scharfe, A. Oefelein, F. Schraut, J. Schreiter, T. Riedel, H. Bauer, R. Niebsch, S. Scherzer, T. Hocker, S. Scholze, S. Henker, M. Nossmann, U. Hensel, and H. Prengel. Adaptive body bias aware implementation for ultra-low-voltage designs in 22FDX technology. *IEEE Transactions on Circuits and Systems II: Express Briefs*, Dec. 2019. ISSN 1558-3791. doi: 10.1109/TCSII.2019.2959544. [pp. 106 and 146.]

[125] A. Dawson, P. D. Düben, D. A. MacLeod, and T. N. Palmer. Reliable low precision simulations in land surface models. *Climate Dynamics*, 51:2657–2666, 2018. doi: 10.1007/s00382-017-4034-x. [p. 111.]

[126] M. Mikaitis, D. R. Lester, D. Shang, S. B. Furber, G. Liu, J. D. Garside, S. Scholze, S. Höppner, and A. Dixius. Approximate fixed-point elementary function accelerator for the SpiNNaker-2 neuromorphic chip. In *IEEE 25th Symposium on Computer Arithmetic*, pages 37–44, Amherst, MA, USA, Jun. 2018. doi: 10.1109/ARITH.2018.8464785. [pp. 113, 154, and 165.]

[127] X. Jin, A. Rast, F. Galluppi, S. Davies, and S. B. Furber. Implementing spike-timing-dependent plasticity on SpiNNaker neuromorphic hardware. In *The*

2010 International Joint Conference on Neural Networks, Barcelona, Spain, Jul. 2010. IEEE. doi: 10.1109/IJCNN.2010.5596372. [p. 114.]

[128] F. Galluppi, X. Lagorce, E. Stromatias, M. Pfeiffer, L. A. Plana, S. B. Furber, and R. B. Benosman. A framework for plasticity implementation on the SpiNNaker neural architecture. *Frontiers in Neuroscience*, 8(429), Jan. 2015. ISSN 1662-453X. doi: 10.3389/fnins.2014.00429. URL http://journal.frontiersin.org/article/10.3389/fnins.2014.00429. [p. 114.]

[129] J. Harrison, T. Kubaska, S. Story, M. S. Labs, and I. Corporation. The computation of transcendental functions on the IA-64 architecture. Online (accessed: 19/05/20) https://www.cl.cam.ac.uk/~jrh13/papers/itj.pdf, Nov. 1999. [p. 114.]

[130] A. Suresh, B. N. Swamy, E. Rohou, and A. Seznec. Intercepting functions for memoization: A case study using transcendental functions. *ACM Transactions on Architecture and Code Optimization*, 12(2):18:1–18:23, Jun. 2015. doi: 10.1145/2751559. URL https://dl.acm.org/doi/10.1145/2751559. [p. 114.]

[131] B. Vogginger, R. Schüffny, A. Lansner, L. Cederström, J. Partzsch, and S. Höppner. Reducing the computational footprint for real-time BCPNN learning. *Frontiers in Neuroscience*, 9(2), Jan. 2015. ISSN 1662-453X. doi: 10.3389/fnins.2015.00002. URL https://www.frontiersin.org/article/10.3389/fnins.2015.00002. [pp. 114 and 115.]

[132] S. Hill and G. Tononi. Modeling sleep and wakefulness in the thalamocortical system. *Journal of Neurophysiology*, 93(3):1671–1698, Mar. 2005. doi: 10.1152/jn.00915.2004. [pp. 114 and 144.]

[133] A. Avižienis. Signed-digit number representations for fast parallel arithmetic. *IRE Transactions on Electronic Computers*, EC-10(3):389–400, Sept. 1961. ISSN 0367-9950. doi: 10.1109/TEC.1961.5219227. [p. 115.]

[134] M. D. Ercegovac. On approximate arithmetic. In *2013 Asilomar Conference on Signals, Systems and Computers*, pages 126–130, Nov. 2013. doi: 10.1109/ACSSC.2013.6810243. [p. 115.]

[135] S. H. Nawab, A. V. Oppenheim, A. P. Chandrakasan, J. M. Winograd, and J. T. Ludwig. Approximate signal processing. *Journal of VLSI signal processing*

systems for signal, image and video technology, 15:177–200, Jan. 1997. doi: 10.1023/A:1007986707921. [p. 115.]

[136] T. Pfeil, T. Potjans, S. Schrader, W. Potjans, J. Schemmel, M. Diesmann, and K. Meier. Is a 4-bit synaptic weight resolution enough? — constraints on enabling spike-timing dependent plasticity in neuromorphic hardware. *Frontiers in Neuroscience*, 6(90), Jul. 2012. ISSN 1662-453X. doi: 10.3389/fnins.2012.00090. URL https://www.frontiersin.org/article/10.3389/fnins.2012.00090. [p. 116.]

[137] J. S. Walther. A unified algorithm for elementary functions. In *Proceedings of the May 18-20, 1971, Spring Joint Computer Conference*, AFIPS '71 (Spring), pages 379–385, New York, NY, USA, May 1971. ACM. doi: 10.1145/1478786.1478840. URL http://doi.acm.org/10.1145/1478786.1478840. [p. 116.]

[138] J.-C. Bajard, S. Kla, and J.-M. Muller. BKM: a new hardware algorithm for complex elementary functions. *IEEE Transactions on Computers*, 43(8):955–963, Aug. 1994. ISSN 0018-9340. doi: 10.1109/12.295857. [p. 116.]

[139] V. Kantabutra. On hardware for computing exponential and trigonometric functions. *IEEE Transactions on Computers*, 45(3):328–339, Mar. 1996. ISSN 0018-9340. doi: 10.1109/12.485571. [p. 116.]

[140] W. F. Wong and E. Goto. Fast evaluation of the elementary functions in single precision. *IEEE Transactions on Computers*, 44(3):453–457, Mar. 1995. ISSN 00189340. doi: 10.1109/12.372037. [p. 128.]

[141] J. Detrey, F. de Dinechin, and X. Pujol. Return of the hardware floating-point elementary function. In *18th IEEE Symposium on Computer Arithmetic*, pages 161–168, Montepellier, France, Jun. 2007. doi: 10.1109/ARITH.2007.29. [pp. 128 and 154.]

[142] M. Langhammer and B. Pasca. Single precision logarithm and exponential architectures for hard floating-point enabled FPGAs. *IEEE Transactions on Computers*, 66(12):2031–2043, Dec. 2017. ISSN 0018-9340. doi: 10.1109/TC.2017.2703923. [pp. 128, 136, 154, 156, and 157.]

[143] K. O. W. Group. The OpenCL C 2.0 specification. Online (accessed: 14/05/20) https://www.khronos.org/registry/OpenCL/specs/2.2/pdf/OpenCL_C.pdf, 2019. [pp. 136 and 156.]

[144] GNU. Known maximum errors in math functions. Online (accessed: 14/08/19) https://www.gnu.org/software/libc/manual/html_node/Errors-in-Math-Functions.html, . [p. 137.]

[145] P. Nilsson, A. U. R. Shaik, R. Gangarajaiah, and E. Hertz. Hardware implementation of the exponential function using Taylor series. In *2014 NORCHIP*, Tampere, Finland, Oct. 2014. doi: 10.1109/NORCHIP.2014.7004740. [p. 154.]

[146] F. de Dinechin and B. Pasca. Floating-point exponential functions for DSP-enabled FPGAs. In *2010 International Conference on Field-Programmable Technology*, pages 110–117, Beijing, China, Dec. 2010. doi: 10.1109/FPT.2010.5681764. [p. 156.]

[147] F. de Dinechin and B. Pasca. Designing custom arithmetic data paths with FloPoCo. *IEEE Design & Test of Computers*, 28(4):18–27, Jul. 2011. ISSN 0740-7475. doi: 10.1109/MDT.2011.44. [p. 157.]

[148] A. Wold, D. Koch, and J. Torresen. Design techniques for increasing performance and resource utilization of reconfigurable soft CPUs. In *IEEE 15th International Symposium on Design and Diagnostics of Electronic Circuits Systems*, Tallinn, Estonia, Apr. 2012. doi: 10.1109/DDECS.2012.6219024. [pp. 157 and 158.]

[149] F. Neumarker, S. Höppner, A. Dixius, and C. Mayr. True random number generation from bang-bang ADPLL jitter. In *2016 IEEE Nordic Circuits and Systems Conference*, Copenhagen, Denmark, Nov 2016. doi: 10.1109/NORCHIP.2016.7792875. [p. 159.]

[150] M. D. Ercegovac. Radix-16 evaluation of certain elementary functions. *IEEE Transactions on Computers*, C-22(6):561–566, Jun. 1973. ISSN 0018-9340. doi: 10.1109/TC.1973.5009107. [p. 160.]

[151] J.-A. Piñeiro, M. D. Ercegovac, and J. D. Bruguera. High-radix logarithm with selection by rounding: Algorithm and implementation. *Journal of VLSI signal processing systems for signal, image and video technology*, 40:109–123, May 2005. doi: 10.1007/s11265-005-4941-7. [p. 160.]

[152] B. Sen-Bhattacharya, S. James, O. Rhodes, I. Sugiarto, A. Rowley, A. B. Stokes, K. Gurney, and S. B. Furber. Building a spiking neural network

model of the basal ganglia on SpiNNaker. *IEEE Transactions on Cognitive and Developmental Systems*, 10(3):823–836, Sep. 2018. ISSN 2379-8920. doi: 10.1109/TCDS.2018.2797426. [p. 164.]

[153] P. Kulkarni, P. Gupta, and M. Ercegovac. Trading accuracy for power with an underdesigned multiplier architecture. In *24th International Conference on VLSI Design*, pages 346–351, Chennai, India, Jan. 2011. doi: 10.1109/VLSID.2011.51. [p. 166.]

[154] S. Sen, S. Venkataramani, and A. Raghunathan. Approximate computing for spiking neural networks. In *Design, Automation & Test in Europe Conference & Exhibition 2017*, pages 193–198, Lausanne, Switzerland, Mar. 2017. doi: 10.23919/DATE.2017.7926981. [p. 166.]

Appendix A

Test script for the Izhikevich neuron

```
"""
Simple Izhikevich neuron test with constant input
"""
import spynnaker8 as p

# Simulation timestep (used in the ODE solver as h)
p.setup(timestep=0.1)

# Regular spiking neuron parameters
cell_params_izk = {'a': 0.02,
                'b': 0.2,
                'c': -65,
                'd': 8,
                'v': -75,
                'u': 0,
                'tau_syn_E': 50.0,
                'tau_syn_I': 50.0,
                'i_offset': 0.0
                }

# Create a population with a single Izhikevich neuron
populations = list()
populations.append(p.Population(1, p.Izhikevich, cell_params_izk,
                label="pop_1"))

```

```
26 populations[0].record("v")
27 populations[0].record("spikes")
28
29 # Run for 60ms, apply DC current and then run further
30 p.run(60)
31 populations[0].set(i_offset=4.775)
32 p.run(3000)
33
34 # Get spike times from SpiNNaker
35 spikes = populations[0].get_data('spikes')
36 print spikes.segments[0].spiketrains[0]
37
38 p.end()
```

Appendix B

Test script for neuromodulated STDP

```
"""
A simple test for neuromodulated STDP.
"""

try:
    import pyNN.spiNNaker as sim
except Exception:
    import spynnaker8 as sim

# Simulation settings
timestep = 1.0
duration = 3000

# Set-up some spike times
t_pre = [1500, 2400] # Pre-synaptic neuron times
t_post = [1502] # Post-synaptic neuron stimuli time
t_dopamine = [1600] # Dopaminergic neuron spike times

# Main parameters from Izhikevich 2007,
# doi:10.1093/cercor/bhl152
tau_c = 1000 # Eligibility trace decay time constant
tau_d = 200 # Dopamine trace decay time constant
DA_concentration = 0.1 # Dopamine trace step increase size

# Initial weight
```

```
26 rewarded_syn_weight = 0.0
27
28 # LIF neuron parameters
29 cell_params = {'cm': 0.3,
30         'i_offset': 0.0,
31         'tau_m': 10.0,
32         'tau_refrac': 4.0,
33         'tau_syn_E': 1.0,
34         'tau_syn_I': 1.0,
35         'v_reset': -70.0,
36         'v_rest': -65.0,
37         'v_thresh': -55.4
38        }
39
40 sim.setup(timestep=timestep)
41
42 pre_pop = sim.Population(1, sim.SpikeSourceArray,
43   {'spike_times': t_pre})
44
45 # Create a population of dopaminergic neurons for reward
46 reward_pop = sim.Population(1, sim.SpikeSourceArray,
47   {'spike_times': t_dopamine}, label='reward')
48
49 # Stimulus for post synaptic population
50 post_stim = sim.Population(1, sim.SpikeSourceArray,
51   {'spike_times': t_post})
52
53 # Create post synaptic population which will be modulated
54 # by DA concentration.
55 post_pop = sim.Population(1,
56     sim.IF_curr_exp_izhikevich_neuromodulation,
57   cell_params, label='post1')
58
59 # Create STDP dynamics with neuromodulation
60 synapse_dynamics = sim.STDPMechanism(
61     timing_dependence=sim.IzhikevichNeuromodulation(
62     tau_plus=10, tau_minus=12,
63     A_plus=1, A_minus=1,
```

```
64    tau_c=1000, tau_d=200),
65    weight_dependence=sim.MultiplicativeWeightDependence(
66    w_min=0, w_max=20),
67      weight=0.0,
68    neuromodulation=True);
69
70  # Create dopaminergic connection
71  reward_projection = sim.Projection(reward_pop, post_pop,
72    sim.AllToAllConnector(),
73      synapse_type=sim.StaticSynapse(weight=DA_concentration),
74    receptor_type='reward', label='reward synapses')
75
76  # Stimulate post-synaptic neuron
77  sim.Projection(post_stim, post_pop,
78    sim.AllToAllConnector(),
79      synapse_type=sim.StaticSynapse(weight=6),
80    receptor_type='excitatory')
81
82  # Create a plastic connection between pre and post neurons
83  plastic_projection = sim.Projection(pre_pop, post_pop,
84    sim.AllToAllConnector(),
85    synapse_type=synapse_dynamics,
86    receptor_type='excitatory', label='Pre-post projection')
87
88  sim.run(duration)
89
90  # End simulation on SpiNNaker
91  print "Final weight: " + repr(plastic_projection.get('weight', 'list'))
92
93  sim.end()
```

Appendix C

C model for the iterative algorithm to compute exp and log

```c
#include <stdio.h>
#include <stdint.h>
#include <math.h>
#include <stdbool.h>
#include <stdlib.h>

// Number of bits in the fractional part of the internal
// fixed-point representation
#define FRACT_BITS 35

// Carry save number
typedef struct {
   int64_t s; // Intermediate sum
   int64_t c; // Intermediate carries
} cs_t;

double exp_log_iterative (uint64_t x, unsigned int n,
   bool exp_not_log, bool print);
uint64_t cs_to_binary (cs_t x);

cs_t CSA_64 (uint64_t x, uint64_t y, uint64_t z);
cs_t CSA_64_4to2 (uint64_t x, uint64_t y, uint64_t z, uint64_t o,
   int c_in_0, int c_in_1);
```

```
24 cs_t right_shifter (cs_t x, int shift_by);
25
26 double fixed_to_double (uint64_t x);
27 uint64_t double_to_fixed (double x);
28 static int64_t log_table [];
29 static int64_t log_table_neg [];
30
31
32 int main ()
33 {
34    // Test single invocation of the exponential function
35    double test = exp_log_iterative(double_to_fixed(0.8), 32, 1, 1);
36    printf("%.30f␣\n", test);
37
38    return (0);
39 }
40
41 // Two's complement log_table [n] = log (1 + 2^(-n))
42 // in s3.60 format.
43 static int64_t log_table [] = {
44 1764759978338438563 7u,
45 1797927463120390886 7u,
46 1818947707478505773 8u,
47 1831094947902343209 7u,
48 1837684864350872647 7u,
49 1841126676670022963 1u,
50 1842886896364080133 6u,
51 1843777187664203583 1u,
52 1844224924733561091 2u,
53 1844449447005999815 0u,
54 1844561872320087087 8u,
55 1844618126115036091 1u,
56 1844646263308698794 6u,
57 1844660334481043189 3u,
58 1844667370711277022 3u,
59 1844670888987432277 4u,
60 1844672648165772356 3u,
61 1844673527765008366 9u,
```

```
18446739675671429099u,
18446741874688393213u,
18446742974198448128u,
18446743523953868800u,
18446743798831677440u,
18446743936270606336u,
18446744004990076928u,
18446744039349813760u,
18446744056529682560u,
18446744065119617056u,
18446744069414584328u,
18446744071562067970u,
18446744072635809792u,
18446744073172680704u,
18446744073441116160u,
18446744073575333888u,
18446744073642442752u,
18446744073675997184u,
18446744073692774400u,
18446744073701163008u,
18446744073705357312u,
18446744073707454464u,
18446744073708503040u,
18446744073709027328u,
18446744073709289472u,
18446744073709420544u,
18446744073709486080u,
18446744073709518848u,
18446744073709535232u,
18446744073709543424u,
18446744073709547520u,
18446744073709549568u,
18446744073709550592u,
18446744073709551104u,
18446744073709551360u,
18446744073709551488u,
18446744073709551552u,
18446744073709551584u,
```

```
100 18446744073709551600u,
101 18446744073709551608u,
102 18446744073709551612u,
103 18446744073709551614u,
104 18446744073709551615u,
105 0,
106 0,
107 0,
108 0
109 };
110
111 // log_table [n] = log (1 - 2^(-n)) * (-1) in s3.60 format.
112 static int64_t log_table_neg [] = {
113 0,
114 799144290325165979,
115 331674847819523230,
116 153951214096912252,
117 74407848895029353,
118 36603757030154788,
119 18156619410792733,
120 9042567959264482,
121 4512418694204213,
122 2254001704453199,
123 1126450020832802,
124 563087437130417,
125 281509342042454,
126 140746078989035,
127 70370891748697,
128 35184908970667,
129 17592320263509,
130 8796126576811,
131 4398054899733,
132 2199025352707,
133 1099512152064,
134 549755944960,
135 274877939712,
136 137438961664,
137 68719478784,
```

```
34359738880,
17179869312,
8589934624,
4294967304,
2147483650,
1073741825,
536870912,
268435456,
134217728,
67108864,
33554432,
16777216,
8388608,
4194304,
2097152,
1048576,
524288,
262144,
131072,
65536,
32768,
16384,
8192,
4096,
2048,
1024,
512,
256,
128,
64,
32,
16,
8,
4,
2,
1,
1,
0,
```

```
0,
0};

// EXP(x)/LN(x) algorithm in carry-save representation
//
// Algorithm is on page 139, Chapter 8
// of J-M Muller's textbook on Elementary Functions, 3ed.
// 10.1007/978-1-4899-7983-4
//
// Input x is the number to be exponentiated (in the interval
// [-1.2,~0.86) and in a chosen format); n is the number
// of iterations.
// For logarithm, x in in interval [~0.4,~3.4].
double exp_log_iterative (uint64_t x, unsigned int n,
    bool exp_not_log, bool print)
{
   unsigned int i;
   cs_t E = {.s = 0, .c = 0};
   cs_t L = {.s = 0, .c = 0};

   // Initialize E_1 and L_1
   if (exp_not_log) {
      E.s = 1152921504606846976 >> (60 - FRACT_BITS); // 1.0
      L.s = x;
   }
   else {
      E.s = x;
      L.s = 0;
   }

   // Start from iteration 1 as ln (1 - 2^0) is not defined.
   i = 1;
   for (; i <= n; i++) {
      // Calculate L* or lambda*
      cs_t tmp = {.s = 0, .c = 0};
      if (!exp_not_log) {
         // Get En - 1
         cs_t tmp0 = CSA_64 (E.s, E.c,
```

```
214             0xffffffffffffffff << FRACT_BITS); // -1
215         tmp.s = tmp0.s << (i);
216         tmp.c = tmp0.c << (i);
217       } else {
218         tmp.s = L.s << (i);
219         tmp.c = L.c << (i);
220       }
221
222       // Leave 3 integer and 1 fractional bits
223       cs_t L_star_cs = right_shifter(tmp, FRACT_BITS-1);
224       L_star_cs.s = L_star_cs.s & 0xF;
225       L_star_cs.c = L_star_cs.c & 0xF;
226
227       // Convert from carry-save to non-redundant
228       // representation
229       uint64_t L_star_b = cs_to_binary(L_star_cs);
230       L_star_b = L_star_b & 0xF;
231
232       // Choose d_n
233       int d = 0;
234       if (exp_not_log)
235         if (L_star_b == 0x0 || L_star_b == 0x1
236             || L_star_b == 0x2 || L_star_b == 0x3)
237                 d = 1;
238         else if (L_star_b == 0xA || L_star_b == 0xB
239             || L_star_b == 0xC || L_star_b == 0xD)
240                 d = -1;
241         else if (L_star_b == 0xE || L_star_b == 0xF)
242                 d = 0;
243         else {
244             printf("Impossible␣case␣at␣iteration␣%d␣L_star␣%llx␣\n",
245                 i, L_star_b);}
246       else
247         if (L_star_b == 0x0 || L_star_b == 0xF
248             || (L_star_b==0x1 && i==1))
249                 d = 0;
250         else if (L_star_b == 0xA || L_star_b == 0xB
251             || L_star_b == 0xC || L_star_b == 0xD
```

```
252             || L_star_b == 0xE) d = 1;
253          else if (L_star_b == 0x1 || L_star_b == 0x2
254             || L_star_b == 0x3 || L_star_b == 0x4
255             || L_star_b == 0x5 || L_star_b == 0x6
256             || L_star_b == 0x7 || L_star_b == 0x8
257             || L_star_b == 0x9) d = -1;
258          else
259             printf("Impossible␣case␣at␣iteration␣%d␣alpha_*␣%llx␣\n",
260                i, L_star_b);
261
262       // Calculate E_n+1 and L_n+1
263       if (d == -1) {
264          L = CSA_64 (L.s, L.c, log_table_neg[i] >> (60-FRACT_BITS));
265          cs_t temp0 = right_shifter(E, i);
266          E = CSA_64_4to2 (E.s, E.c, ~temp0.s, ~temp0.c, 1, 1);
267       }
268       else if (d == 1) {
269          L = CSA_64 (L.s, L.c, (log_table[i]) >> (60-FRACT_BITS));
270          cs_t temp0 = right_shifter(E, i);
271          E = CSA_64_4to2 (E.s, E.c, temp0.s,temp0.c, 0, 0);
272       }
273    }
274
275    // Use a single ripple carry 32-bit adder to get the answer
276    // in non-redundant form.
277    if (exp_not_log)
278       return fixed_to_double(cs_to_binary(E));
279    else
280       return fixed_to_double(cs_to_binary(L));
281 }
282
283 // Carry-save 64bit adder made out of 3:2 compressors
284 // (full-adders). Adds three 64-bit numbers and produces
285 // two 64-bit numbers - intermediate sum and carry.
286 cs_t CSA_64 (uint64_t x, uint64_t y, uint64_t z)
287 {
288    cs_t result = {0, 0};
289    result.s = x ^ y ^ z;
```

```
290     result.c = ((x & y) | (x & z) | (z & y)) << 1;
291     result.s = result.s;
292     result.c = result.c;
293 
294     return result;
295 }
296 
297 // Fast 4:2 compressor (page 123 of "Digital Arithmetic"
298 // by Ercegovac and Lang, 10.1016/B978-1-55860-798-9.X5000-3)
299 cs_t CSA_64_4to2 (uint64_t x, uint64_t y, uint64_t z, uint64_t o,
300     int c_in_0, int c_in_1)
301 {
302     cs_t result = {0, 0};
303     uint64_t majority = (((x&y) | (y&z) | (x&z)) << 1) + c_in_0;
304     uint64_t odd_parity = ((x^y) ^ (z^o));
305 
306     result.s = odd_parity ^ majority;
307     result.c = (((odd_parity & majority)
308         + (~odd_parity & o)) << 1) + c_in_1 ;
309 
310     return result;
311 }
312 
313 // Standard carry-propagate adder
314 uint64_t cs_to_binary (cs_t x)
315 {
316     return x.s + x.c;
317 }
318 
319 // Carry-save number right shifter
320 cs_t right_shifter (cs_t x, int shift_by)
321 {
322     x.s = x.s >> shift_by;
323     x.c = x.c >> shift_by;
324 
325     return x;
326 }
327 
```

```
328 double fixed_to_double (uint64_t x)
329 {
330     int negative = (x & ((uint64_t)0x1 << (FRACT_BITS+4))) != 0;
331     if (negative)
332         x = ~x + 1;
333 
334     uint64_t integer = x >> FRACT_BITS;
335     double exponent =
336        (double)(x - (integer << FRACT_BITS)) /
337           ((uint64_t)1 << FRACT_BITS);
338 
339     double result = integer + exponent;
340     if (negative) result = result * -1;
341     return result;
342 }
343 
344 uint64_t double_to_fixed (double x)
345 {
346     bool negative = x < 0;
347     if (negative) x = -1 * x;
348     uint64_t temp = floor(x);
349     uint64_t result = (uint64_t)((temp << FRACT_BITS)
350         + ((uint64_t)1 << FRACT_BITS) * (x - temp));
351     if (negative) result = ~result + 1;
352     return result;
353 }
```