A fixed point arithmetic library for SpiNNaker

A dissertation submitted to the University of Manchester for the degree of **Master of Science** in the **Faculty of Engineering and Physical Sciences**

-2012-

Mircea Moise

School of Computer Science

Contents

Word count: **27 960**

List of tables

List of figures

Abstract

The normal approach to implementing real number computation in computer architectures is to make use of specialised hardware. This is best for most scenarios, but for highly distributed energy-optimised systems such as SpiNNaker this may not be the case. The alternative is to use fixed point representation of numbers and a software implementation of the computations. This dissertation explores this approach in the context of the SpiNNaker project.

The work here presented aims to provide a set of routines that implement arithmetic operations for fixed-point numbers to be used for the SpiNNaker project. These routines should implement efficient and accurate algorithms. In the first stage of the project the requirements were gathered. In the second stage the mathematical techniques that have potential to be used are examined. A brief survey of these techniques is part of the dissertation. In the third stage for each arithmetic operation algorithms are developed and assessed. Details of the algorithms and results of assessment are part of this report. In the fourth stage the best algorithms for each operation are implemented. In the last stage an evaluation of the efficiency and accuracy of the routines is carried out.

The results showed that for most arithmetic operations (square root, logarithm, exponential, sine) the algorithms implemented are accurate and more efficient than the default C implementation. The analysis of the algorithms implemented for division showed that there is a need for optimisation to make the routines more efficient. A brief evaluation of the usability of the routines highlighting possible pitfalls of usage is presented.

This work described in this dissertation shows that efficient implementation of real number arithmetic operation in software using fixed-point arithmetic is possible. The fixed-point software library can be used in the development of the SpiNNaker project. Future work can be carried out to further optimize the algorithms, especially by writing code optimized for a specific assembly instruction set.

Declaration

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

Intellectual property statement

Acknowledgement

I want to thank my supervisor, Prof. Steve Furber, for his guidance throughout implementation and writing. I also want to thank the SpiNNaker team for the help provided to better understand the project.

I am grateful to "Dinu Patriciu" Foundation for awarding me the "Open Horizons" scholarship. I am also grateful to Romanian-American Foundation for the financial help provided for the second part of the year.

I wish to thank my dear friends Alexandra and Daniel for all the support they gave me this year. It would have been many times more difficult to finish this year without their help.

Finally, I want to express my gratitude to my family: Tatiana, Ioan and Oana for their unconditional support.

Chapter 1 Introduction

1.1 Motivation

The work presented in this dissertation has as its purpose the development of a fixed-point arithmetic library to be used in the SpiNNaker project. SpiNNaker "is a novel massively-parallel computer architecture, inspired by the fundamental structure and function of the human brain" [1]. Two of its main objectives are: to provide a platform for simulating neural networks in real-time and to investigate a new, massively-parallel computer architecture. The building block of SpiNNaker architecture is an ARM processor core which models 1000 *neurons*. Each of these cores is connected to 6 others enabling communication between neurons modelled by different processing units. The communication consists in transferring short messages (a few bytes). The SpiNNaker board is a Globally Asynchronous Locally Synchronous (GALS) system. This means that while the ARM chip runs in synchronous mode, the communications between the cores are asynchronous [2].

One of the main constraints that SpiNNaker faces is energy consumption. Because the goal is to have more than 1 million cores on a SpiNNaker machine, the core had to have a low energy footprint. The chosen processing unit was: ARM 968 (the variant without floating-point unit), which is a 32-bit RISC processor and "the smallest, lowest power ARM9 family processor" [3]. There is a variant of this chip with a floating-point unit but choosing it would have increased both the size of the die and the energy consumption.

One of the downsides of this type of chosen architecture is the lack of floating point computations. The software stack that runs on the SpiNNaker machine makes use of real number computations. In order to address this issue, fixed point representations of numbers and operations on integers are used. The set of native operations available includes: shift, add (subtract) and multiply. The absence of native division operation should be noted. All the arithmetic used by the software stack should be reduced to the operations named above that use fixed point representations of terms.

The current approach is that the developer builds his solution taking into account the constraints described. This approach has several disadvantages. Firstly, the development time increases greatly as the developer has to spend time on devising ways to overcome the constraints. Secondly, the solutions obtained may not always be efficient and may be

error prone. Thirdly, the code obtained is difficult to maintain or debug. This happens mainly because the concept of the separation of concerns (arithmetic computations and main task) cannot be respected.

The project will try to solve the problems generated by the lack of a floating point unit. The approach used is to develop a library of software routines that contain efficient implementations of common arithmetic operations using the basic operations that are available. This solution addresses the issues described above. The development time will decrease, as the developer only has to call one function. The efficiency will increase, as the functions will implement the best solution available. The obtained code will be easy to maintain and to debug, as the principle of the separation of concerns will be respected.

The set of routines will be written in Standard C and ARM assembly. The software stack for the SpiNNaker machine is developed using Standard C for portability and ARM assembly for efficiency purposes. The plan is to develop several libraries with different purposes:

- a C library encapsulating the representation system
- a C library containing efficient implementations of arithmetic operations
- a C library with debugging recording enabled
- an optimized library using ARM assembly

One of the aims of the software that runs on the SpiNNaker machine is to be portable to other architectures. One reason for that is the possibility given to developers to debug the code they write on their desktop machines without needing to use tools for simulating an ARM processor. Another important reason is to make the software useful to researchers that do not necessarily have a SpiNNaker machine. Writing the libraries in Standard C is a good way to ensure portability, as virtually any computer architecture has a C compiler. Since the main use of the libraries will be in conjunction with SpiNNaker machines that use ARM chips as cores, the choice is to have an optimized version built for the ARM architecture. If needed, other versions of the routines can be developed for other kinds of architecture in the future.

The main purpose of the libraries is to enable real number computations, and in order to achieve that, a representation system for the numbers has to be built. In order to separate concerns, a distinct library will be built to handle the data structures representing the numbers and the functions that facilitate the creation and the access to

those data structures. The main C library will contain efficient implementations for each important operation that is required. The debugging version library will have the same functionality as the main library and will be able to record information about how it is used: the range of operands, overflows or the frequency a function is called. This kind of data will be useful in fine tuning the algorithms and can give important information to the researchers.

1.2 Aim and Objectives

The aim of the project is to develop a set of C libraries containing efficient implementations of the basic arithmetic functions that use fixed point representations of the operands and the result.

These libraries will be used in environments that do not offer floating point arithmetic support. The libraries will be developed in the context of the SpiNNaker project but will be able to serve other systems that need floating point arithmetic and do not have the necessary hardware.

The project objectives are:

1. Assessing algorithms for fixed point computation
2. Devising a representation system for fixed point numbers
3. Choosing and implementing algorithms for arithmetic functions

The first step would be to research and assess the algorithms that can be used to implement the arithmetic operations needed, using basic operators: shift, add and multiply. This is important because it lays the ground for efficient software implementation. The second step is to devise a good representation system that accommodates the requirements. One of the aims of the representation system is to be easy to use and to prevent a loss of performance. The last step is to develop software implementations of the algorithms.

1.3 Report Structure

This dissertation describes the wider context in which the work fits, sets out the research methods to be used during the project, describes the mathematical background used and presents the results of analysing and implementing the algorithms.

Chapter 2 delimits the areas that the work will touch, describing the requirements and constraints of to the project.

Chapter 3 contains a description of the research methods used, stating their relevance in the context of the project.

Chapters 4 and 5 delineate the progress done so far in assessing algorithms, first enumerating general techniques and then describing for each operation the best approaches.

Chapter 6 describes the implementation of the algorithms detailed earlier, along with the auxiliary tools needed.

Chapter 7 presents the results of evaluating the software implemented

Chapter 8 details the conclusions of the project and describes the future work that can be done

Chapter 2 Context

In this section of the report a detailed description of the project's scope is provided. It sets out the wider context of the work and points out the requirements and constraints each domain sets on the project. The following sub-sections will talk about: SpiNNaker, the ARM core, number representation systems and existing software implementations for fixed-point arithmetic.

2.1 SpiNNaker

One of the aims of the SpiNNaker project is to create a platform that enables the simulation of 1 billion neurons [4]. In order to achieve that the system needs 1 million cores, each of them being responsible for the simulation of 1000 neurons [5]. The configuration that will support these requirements will contain: 1,036,800 cores and 7 terabytes of RAM, distributed over 57000 nodes [6]. Each node is a System-in-Package consisting of 18 ARM968 cores. Each core has its own local memory and dedicated peripherals: DMA (Direct Memory Access), counters, timers, interrupt controllers and communications controllers [2]. The node offers resources such as: boot ROM, shared on-chip RAM, a watchdog timer, a system controller and Ethernet interface [2]. One of the 18 cores of the node plays the role of the "monitor core" which takes care of system management functions, while the rest of the cores are used for application purposes [2].

Machine name	**No of nodes**	**ARM cores**	**Power (Watts)**	**Physical format (comparison)**	**Cooling method**
102	4	72	5 (5V, 1A)	Small circuit board	-
103	48	864	72 (12V,6A)	Double extended Eurocard	-
104	576	10 368	1 K	Desktop card frame	Fan cooling
105	5 760	103 680	10K	19" rack cabinet	Air conditioned
106	57 600	1036800	100K	10 X 19" rack cabinets	Air conditioned

Table 1 SpiNNaker machine description

In order to achieve the desired size in terms of cores (1 million) an incremental approach was used. At each step the new board steps up an order of magnitude in terms of cores used. Table 1 illustrates the size of the SpiNNaker machines in terms of logical

units (nodes and cores), the physical size (giving a comparison) and their energy consumption. The table also shows that for the bigger machines (105 and 106) physical size and energy consumption are constraints. The architecture of SpiNNaker should be built so it optimizes both the physical size and the energy consumption.

The chips in a node are placed in a grid in order to facilitate communication. A schema of how the grid looks is presented below. Each square represents a SpiNNaker core and the numbers written on it the position in the 2D grid (row, column).

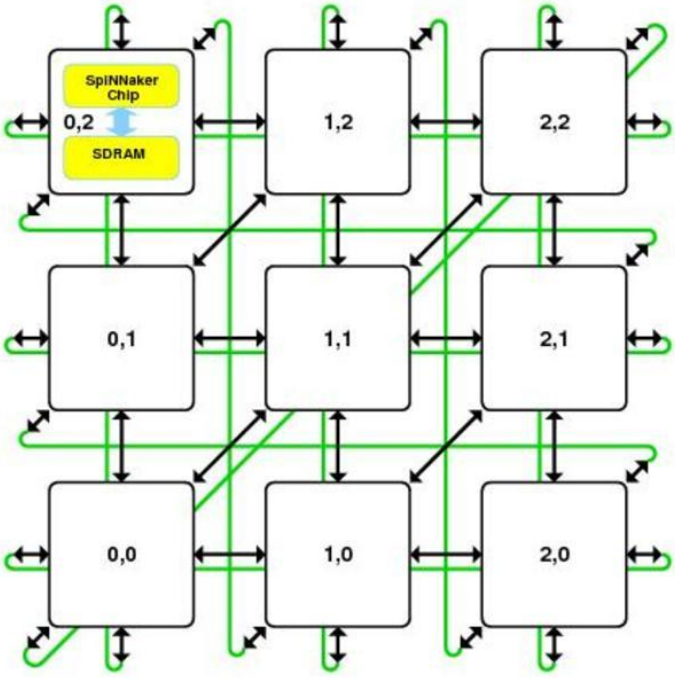

Figure 1 SpiNNaker chip connection [7]

Figure 1 shows how the individual cores are interconnected in a node. Each of them has 6 links connecting it with different neighbours. The communication between two different cores is made via the links shown. In order to emulate different structures of neuron population a software approach is used to select the links to be used in the actual communication between cores.

The local memory of an individual ARM core is divided between instruction and data memory [2]. The instruction memory has 32 kilobytes and the data memory 64 kilobytes [2]. This shows the *first requirement* for the libraries: the code should not be more than 32 kilobytes in size and data should not be larger than 64 kilobytes. These upper bounds for code and data are not realistic as the core should run other instructions and have access to other kinds of data. After considering the jobs that the core has to run, more

sensible targets were set. For instructions data an upper bound was not set as the desire is to have a small number of instructions (which should mean more efficient algorithms). For data the upper bound should be under 1 kilobyte, in the attempt to make it as low as possible, taking into consideration that a memory footprint larger than 1 kilobyte is not acceptable.

One of the purposes of the SpiNNaker project is to offer a tool for modelling neuron populations. Each individual ARM core is responsible for simulating up to 1000 neurons. This means that one of the functions of the ARM core is to implement the simulation of the neuron. A model of a neuron consists of rules that given an input from a spike, outputs a result. There are several models that can be used: the point-neuron, leaky integrate-and-fire, spike response and the Izhikevich model. All these are mathematical models describing the output using equations that contain parameters and the value of the input. These models require the use of real number representation and computation. The lack of a floating point unit on the cores and the need for the real number computation makes necessary a software implementation of it.

One of the experiments in neural sciences that have been run on the available SpiNNaker machines is modelling the Izhikevich neuronal dynamics [5]. The experiment and its implementation on SpiNNaker have been described in detail in [5] and [8]. This example is representative for the class of experiments that will be built and, in particular, for the kinds of arithmetic support needed. The equations that have to be computed rely on real number operations. In the above work, the steps done by the developers to overcome the lack of floating point support are detailed. These steps increased the development time, proving the need for a library to implement real-number operations.

Taking into consideration the Izhikevich model implementation and the development team experience with the system, several other requirements for the libraries emerged. From the standpoint of fixed point formats, the list of formats currently used in the development should be maintained and extended. A short enumeration of the fixed point formats includes: 8.8, 0.16, 16.16, 8.24, 5.8, 8.5. These formats can be classified depending on the total number of bits used, having: a 2 byte class (8.8, 0.16) – equivalent to the short data type, a 4 byte class (16.16, 8.24) – equivalent to the integer data type and a 13 bit class (5.8, 8.5). The representation system should accommodate these formats and be easily extendable with other kinds of format

(e.g. a high precision format such as 0.32). Another requirement regarding the representation system is that it should have a high degree of usability. This means that, for example, creating new numbers should be easy to do and should not bring other sources of error.

Modelling neuronal behaviour is an area where real number representation and computation is highly needed, but there are other areas of development that require it. One purpose of SpiNNaker is to act as distributed computer architecture. This would mean that the user should be able to run tasks efficiently on other architectures. The ability to use and compute real numbers is a basic feature for any computer systems. The current status enables using floating point numbers but with a huge cost in performance. The use of libraries being built will improve SpiNNaker's performance as a distributed architecture.

From the viewpoint of language used for development, two choices arise: using Standard C or ARM assembly. In the implementation of the Izhikevich model one method used to speed up computations was writing in assembly. Both choices have advantages and disadvantages. Standard C provides portability of the code, meaning that it can be used in different environments or SpiNNaker-like architectures that use a different chip. The trade-off for *portability* is *efficiency*. Writing code in ARM assembler will make the code not portable, but very efficient as it can access specific ARM instructions. Also, writing in assembly increases *efficiency*, as the programmer has a better control over the instructions to be run. In order to accommodate both *portability* and *efficiency*, a library written in Standard C will be developed and an optimized version of it will be written in ARM assembly. The way to write the optimized version for ARM is to modify the assembly code obtained after compiling the Standard C library. This is a better alternative than embedding ARM assembly code in C, because the latter approach is error-prone.

From the point of view of operations needed to be implemented, *division* is the most important. As can be seen in the papers describing the Izhikevich experiment, the division operation was avoided as the ARM chip does not have a native instruction for it. Although in this case division could be avoided, this is not always possible. Related to division, *modulo* is another desired operation. The list of operations of high importance is continued by: *exponential, logarithm (natural, binary, decimal), power, square root*. The next

operations, in terms of importance and usage, are *sigmoid* and *trigonometric functions (sine, cosine, tangent)*.

One important observation regarding the efficiency of the algorithms used is that it should depend only on the length in bits of the input terms and not on the values. The main reason for that is the fact that a precise measurement of the cycles used by the core to run a piece of code has to be available. If the efficiency[1] varies depending on the values of the input terms, a precise measurement of the cycles used cannot be provided.

2.2 ARM

The choice for the SpiNNaker core was an ARM processor. ARM is both an architecture and a type of processor for embedded devices. What makes this architecture a good fit for SpiNNaker is its simplicity that allows small implementations and, therefore, low energy footprint [9]. ARM is a common RISC (Reduced Instruction Set Computer) architecture, having features such as [9]:

- a load/store architecture
- simple addressing mode
- uniform fixed-length instructions

In addition to typical RISC features, the ARM architecture has several improvements [9]:

- control over the ALU (Arithmetic Logic Unit) and shifter in every data-processing instruction
- auto-increment and auto-decrement addressing modes
- conditional execution of all instructions

The processor model is the ARM968 which is part of the ARMv5 family. This processor is "a small footprint core for low power, data intensive applications", providing the smallest footprint for the highest power efficiency [3]. This processor can handle ARM instructions, Thumb instructions and DSP extension instructions. The length of an instruction is: 32 bits for ARM instructions and 16 bits for Thumb instructions [3]. The Thumb instruction set is a subset of the most frequent instructions that are synthesized to 16 bits for a higher code density. The instruction pipeline has 5 stages and the processor accesses a tightly coupled memory (TCM) [3]. The TCM is divided between: instruction memory (ITCM) and data memory (DTCM). The dimensions available for each memory are between 1KB and 4 MB in power-of-two

[1] Efficiency, in this case, means number of operations done by the core

increments [3]. The values chosen for SpiNNaker are: 32KB for ITCM and 64KB for DTCM.

The architecture allows the use of 4 condition flags: negative, zero, carry and overflow. Almost every instruction has a 4-bit condition field with 16 possible values: one for unconditional execution, one for instructions that do not allow conditional execution and 14 different conditions including: tests for equality and non-equality; tests for inequalities and tests for each condition flag [9]. If the condition is met then the instruction runs normally, if not, it does not run at all. The instruction set contains 6 types of instruction [9]:

- branch instructions
- data-processing instructions
- status register transfer instructions
- load and store instructions
- coprocessor instructions
- exception-generating instructions

In the context of the project the first two types of instructions are the most important. There are 4 types of data-processing instruction: for ALU, comparison, multiply and count leading zeros [9]. Every *arithmetic/logic* instruction performs both the operation it designates and an optional shift. Also, this kind of instruction updates the condition flags. The *comparison* instruction does not write an output to a register, it only updates the condition flags. The *multiply* instructions takes as input two 32-bit operands and outputs either a 32-bit result or a 64-bit one. The *count leading zeros* (CLZ) instruction returns the number of leading zeros.

For division and floating-point number representation, the ARM chip relies on its Vector Floating-Point (VFP) coprocessor. The VFP is compliant with the IEEE 754 (floating point) standard [9]. The VFP also has a software implementation in order to cope with trapped floating-point exceptions [9]. The variant chosen for the SpiNNaker core does not have a VFP coprocessor making the VFP instructions unavailable. This means, for example, that a division operation written in C and compiled for this architecture will be transformed into a long sequence of instructions, expensive in terms of cycles used. The technology used for building the chip is 130 nm, making the total size of the VFP around 1 mm^2 while the total size of the ARM chip is around 3.5 mm^2 which means an increase of around 28% in size. Also the energy consumption for the VFP unit is around 0.4 mW/MHz, much more than the value of 0.14 mW/MHz for the

ARM chip. These numbers show how disadvantageous the choice of using a floating point unit would have been.

	ARM 968 without VFP unit	ARM 968 with VFP unit
Size (mm^2)	3.5	4.5
Power used/clock speed (mW/MHz)	0.14	0.54

Table 2 Impact of VFP unit on ARM core

Table 2 shows a comparison between the ARM 968 core without VFP unit and the core with VFP unit. It can be seen that there is an increase of 28% in size and of almost 300% in energy consumption. The use of the VFP version in the hardware design would have had an important negative impact on the total size and energy consumption of the SpiNNaker machines (especially for the large machines)

2.3 Number representation systems

In order to represent real numbers on computers several methods were devised over time. Floating-point representation is the most commonly used. In this kind of representation every number can be seen as a function of 4 values: *sign, digits, base* and *exponent*. The formula for representing a number *n* is:

$$n = sign * digits * base^{exponent}$$

The *base* is usually 2 in common computer architectures. In order to store the values for sign, digits and exponent, 32 bits are used. These bits are used in this way: the first bit for the sign, the next 8 bits for the exponent and the remaining 23 bits for digits. To represent the exponent a bias equal to 127 is added, meaning its range would be [-127, 128]. Using the floating point representation allows a large range of numbers to be expressed. The disadvantage of this representation is that arithmetic operations are more cumbersome than for simpler representation systems. For example, addition requires an intricate algorithm if the exponents of the terms are not equal. This is a problem especially for embedded systems where chips have to be small in size and have to run code efficiently.

In contrast to the floating - point representation, the fixed-point representation assigns a fixed number of bits for both the integer part and the fractional part. The range of numbers represented by this system is smaller than for floating-point. The

advantage of this type of representation is the fact that arithmetic operations can be easily done on it. For example, to add two fixed point numbers, two integer add operations are needed (using a carry) – one for the integer part and one for the fractional part. This means that the fixed point representation does not require additional hardware (all the operation needed can be performed on an ALU) or intricate software implementation. All these features make fixed-point representation the best choice for embedded systems architectures (e.g. DSPs – Digital Signal Processors). The downside of not using specialized hardware for floating point is that hardware also implements arithmetic operations such as: division, multiply, power, logarithm and trigonometric functions. In order to use those operations in an environment without a floating point unit, a software implementation is required.

2.4 Software implementation for fixed point arithmetic

Efforts have been made to introduce the fixed point representation system into the C programming language. One such effort was the technical report: ISO/IEC TR 18037:2004 [10] which is known as the Embedded C standard. Its description is [10]:

"ISO/IEC TR 18037:2004 deals with the following topics: extensions to support fixed-point arithmetic, named address spaces, and basic I/O hardware addressing."

In the document [11] new data types and operations for fixed-point representation are described. Six primary fixed-point types were defined: *short _Frac, _Frac, long _Frac, short _Accum, _Accum, long _Accum*. For each of the types described above an unsigned version exists (e.g. unsigned short _Fract). Each signed or unsigned type can be saturated or not, this status being defined by the existence of the *_Sat* keyword (e.g. _Sat unsigned short _Fract) [11]. The document also defines arithmetic operations that can be used in conjunction with these types. The list of operators contains: unary operators (+, -, !); binary operators (+, -, *, /); shift (<<, >>); relation operators (<, ==, >, !=); assignment operators (=, +=, -=). It should be noted that this standard does not define in any way how the operation will be implemented; it only defines the types and the operators. Another important observation is that the standard is not yet implemented in common compilers. For example, the GNU C Compiler (GCC) and ARM C Complier (ARMCC) do not have support for fixed point types and operators defined by the Embedded C standard. The standard can be used as a starting point for the representation system that will be built.

Another effort to bring fixed point arithmetic into the C language is an open source project: *libfixmath* [12]. It represents a library implementing the custom math C library's functions for numbers using the fixed point format 16.16 . It implements the following types of functions: saturated arithmetic, multiplication, division, square root and trigonometric. The disadvantage of this library is that it only accepts one type of fixed point format. The library can be benchmarked against the libraries that will be produced in order to have a sense of how the latter performs.

2.5 Summary

In this chapter the context of the project was presented. It was described in the context of the larger effort of which this project is part (SpiNNaker) and highlighted the reasons why real number computations are needed. It also detailed the main architecture that is the final target of the libraries produced (ARM). A comparison between two common ways of representing real numbers (floating point versus fixed point) was also provided in this chapter. In the final part of the chapter, a short survey of the existing efforts towards creating a set of fixed-point libraries is presented.

Chapter 3 Research Methods

This chapter describes the methodology used during the project. The first subsection will describe the project's phases, while the rest of the subsection will detail the methods and tools used.

3.1 Project outline

The project is structured in several phases according to what has to be achieved in each stage. Each phase has different objectives, tackles different parts of the project's scope and implies distinct research methods. The breakup in stages is done as follows:

1. *First phase*: understanding the SpiNNaker system and gathering requirements
2. *Second phase*: assessing algorithms for efficient implementation of the operations
3. *Third phase*: devising and implementing a representation system and implementing the algorithms assessed in the second phase
4. *Fourth phase*: benchmark the way the libraries perform in the context of SpiNNaker

In the *first phase* a deeper understanding of the SpiNNaker architecture is developed. This is an important step in order for the project to fit in with the existing SpiNNaker efforts. During this stage requirements for what the system needs in terms of arithmetic computations were gathered. Also, an inventory of the constraints faced by the libraries was maintained. The main way the information was collected, was by talking with the development team and being part of their bi-monthly meetings. Also, an important part of the research was reading articles about SpiNNaker and investigating current support for fixed point arithmetic. A concise presentation of the results of this first phase was presented in the second chapter of this report *Context*.

In the *second phase* a survey of the literature was done to select techniques or algorithms that can be used to make efficient computation of arithmetic functions. These techniques and algorithms are analysed and assessed for the general use case and for the particular context of running on ARM cores. The way the literature survey was done and a description of the methodology used to analyse and assess the algorithms will be presented in the next subsections of this chapter. A presentation of the results of this phase can be found in chapter 4 of this report, *Progress*.

The *third phase* consists first of devising and implementing the representation system and then implementing the algorithms assessed in the previous stage. This phase includes 3 iterations. During the first iteration, the representation system and basic arithmetic functions (+, -, multiply) will be implemented. In the second iteration algorithms for division, modulo and power will be implemented while in the last

iteration algorithms for exponential, logarithm and sigmoid will be implemented. In the third iteration an optimized version for ARM will also be built. Considerations about the tools used for implementation and the workflow will be presented later in this chapter.

In the *fourth phase* benchmarking of the implementation will be done to assess the impact on SpiNNaker performance and ARM fixed point computations. Details about how it will be done are given later in this chapter.

3.2 Methodology for assessing algorithms

After requirement gathering and understanding SpiNNaker's architecture the next stage in the project is finding algorithms that can be used to implement the selected operations. The first step in this process is surveying the literature concerning arithmetic operation implementation. The scope of investigation consists of fixed-point arithmetic computation algorithms and, also, floating point arithmetic computation algorithms. The latter case is useful because the algorithms for computation are usually indifferent regarding representation systems. Some algorithms can be dedicated to or can be working more efficiently with one kind of representation system, but in general the algorithms described for floating point can be used for fixed point numbers. Another source of documentation is math literature regarding ways of approximating values of different functions. Understanding the way an algorithm works helps when trying to improve it. Existing implementations in commonly used systems of fixed-point computation are also valuable pieces of information.

The algorithms and techniques assessed have to meet some criteria before being further analysed. The first characteristic is that the only operations implied by the algorithm are additions and multiplications. Algorithms and techniques relying on division cannot be implemented. An observation is that algorithms that try to avoid multiplications, considering them computational expensive, are probably not very efficient in the ARM context. This is because on ARM architecture multiplication is a relatively inexpensive instruction, and trying to avoid it in an algorithm usually leads to an increase in other operations (additions, shifts) therefore decreasing the algorithm's efficiency. The second characteristic is that the complexity of the algorithms in term of operations should depend on the length of the input and on the precision the result should have. This is the reflection of one of the system's requirements. The third

characteristic is that the algorithm should not use huge amounts of pre-computed data. This is due to the data limit inside the chip.

If one algorithm meets the criteria described above then a detailed assessment will be done. This will determine 3 key characteristics in evaluation:

1. Number of operations needed (shifts, adds, multiplications, comparisons)
2. Size of data needed
3. Value of the error

The number of operations needed is just a rough approximation because during the implementation phase this number is reduced by using optimizations. In order to determine an approximation for the number of operations, a pseudo-code variant of the algorithm is devised. The size of the data needed accounts for data needed for permanent storage (i.e. lookup tables) and not the data needed for variables during code execution. Like the number of operations, the amount is just an approximation. These numbers should be expressed as a function of the precision of the algorithm.

Expressing the value of error is more difficult than estimating the number of operations. The difficulty arises from the fact that, usually, the error is not constant on the interval considered. It may vary, and the variations can be very high. The best way to present the value of the error is through plotting a graph. Along with displaying the graph, important data about it is attached, such as: maximum and minimum value or median error. Also considerations about the error can be drawn from the graph such as if the error varies too much or if it is in the bounds set. In order for the graph to accommodate large variations of the error and to give a better understanding the precision of the algorithm, the function *f* plotted will be:

$$\mathrm{f}(x) = \log_2 |approx(\mathrm{x}) - \ function(\mathrm{x})|$$

Where approx(x) is the value estimated by the algorithm and function(x) is the actual value. This formula allows a large scale because of its use of logarithms. Another important property is that on the y axis the precision in bits can easily be seen. For example, for an algorithm and one number x, *f(x) = -10* means that the precision of the algorithm in that point is 10 bits. Using this characteristic, thresholds can be set as lines parallel with the x axis on the plot. Considering that the most used formats have a maximum of 16 bits precision for the fractional part, a general threshold was set at *f(x) = -16* for all the algorithms.

One important aspect to note is the precision of the intermediate results. During the implementation the intermediary results will also be fixed point numbers generating new sources of error. Because details of the implementation can be subject to change, the exact error cannot be computed accurately. In order to accommodate that, two different ways to express the error will be used: the first using infinite (i.e. very large) precision of intermediary results and the other using 16 bits of precision for intermediary results. The first approach is useful because it can easily be implemented and the values obtained probably will not be far from the implementation. The second approach gives a more realistic image of the error, but it requires more implementation effort. The tool used to plot the error was Matlab. More details about the tool usage will be provided later in this chapter.

3.3 Tools used

A range of different tools will be used during the project. On the stage of assessing the algorithms Matlab will be the main tool used. Matlab is "a high-performance language for technical computing" [13]. It integrates "computation, visualization, and programming in an easy-to-use environment" [13]. The main feature used was its ability to plot graphs easily. Because implementing an algorithm is straightforward and the results can be easily analysed, Matlab was used to implement the first versions of the algorithms. Also, it meant that altering the algorithms' parameters and structure to improve performance was easily done. Matlab proves to be an excellent tool for analysing algorithms.

In the stage of implementing algorithms *Eclipse for C/C++* [14] and *ARM Developer Suite* will be used. Eclipse is an open-source Integrated Development Environment (IDE) providing all the facilities needed for programming. It will be used to develop code for C and run it on a desktop environment. The ARM Developer Suite is software comprising of an IDE for ARM assembly development and a simulator for ARM architectures. The software will be used to write code in ARM assembly. For debugging and benchmarking purposes *Komodo* [15] is used. Its main features are: enabling hardware debugging of a physical board and emulating in software ARM cores [15].

In order to control the versions of the code *Git* was used as version control system. Using a versioning system is required considering the size of the code written (both in terms of lines of code and also in terms of the number of files) and the extent in time of the implementation (3 months). Keeping track of the progress made by analysing the

commit messages, protecting the work from potential unwanted deletes and the ability to revert to a previous working version are important reasons to adopt the use of a versioning system. An important feature of a versioning system is the ability to *branch* at some point in development. This enables having two different current versions of the codebase. This feature proves helpful when there is the need to experiment with new approaches without disrupting the existing code. *Git* is a distributed revision control and source code management system developed by Linus Torvalds. One advantage of it being distributed is that multiple repositories exist at the same point. In this case there were 2 different repositories: the local one and the remote one hosted by Github.com.

One of the requirements of the testing framework was that it creates large files containing test-cases. As this task cannot be done successfully by hand, the use a programming language to build them is needed. Ruby was chosen as the tool for it. It has the main advantage of handling strings and files easily (unlike C or Java). *Ruby* is a dynamic general purpose object-oriented language influenced by other language such as: Perl, Smalltalk, Eiffel and Lisp. The main focus of the language is to make coding more efficient (in terms of time and line of codes) and to make it easily readable for humans.

In the benchmarking stage, one of the tools used will be a SpiNNaker board. Depending on the availability of the boards, one existing version will be used.

3.4 Evaluation strategy

The evaluation of the results can be separated into 3 steps: *benchmarking one operation, benchmarking using a complex algorithm involving more than one operation* and *benchmarking with SpiNNaker*

In the *benchmarking one operation* stage, a test consisting of several computations for the same operation will be run on different environments. The results looked for are accuracy level (comparing them with the ones obtained in the first analysis of the algorithms) and the time needed to run the computations. First the test will run on each environment without the use of the libraries, using the compiler's way to replace operations. The environments used are defined below:

- x86 desktop architecture using C library
- ARM architecture using C library
- ARM architecture using an optimized ARM library

Benchmarking using a complex algorithm requires running a test that comprises all (or most) of the functions implemented on each environment described above. First the

test will be run without the use of libraries for benchmarking purposes. The test will contain the algorithm for solving a real problem common for the SpiNNaker system.

Benchmarking with SpiNNaker requires rewriting the Izhikevich model so it uses the libraries developed. Two versions will be developed: one using the C library, another using the ARM optimized library. The results in terms of number of operations and time will be compared with the results obtained with the initial implementation previous to the work.

3.5 Summary

This chapter has presented the methods and tools used during through the project. In the first section the organization in stages of work was detailed. The second section described the general method of analysing the algorithms. The following section presented the tools when in assessing and implemented the algorithms. The last section shortly described the evaluation strategy planned to be used.

Chapter 4 Theoretical Background

In this section general techniques for approximating function values are described. The list of techniques used comprises: Taylor series, Chebyshev polynomials, Newton-Raphson iteration, De Lugish (radix) algorithms. Also this section will contain considerations of the normalization to the [0.5,1) interval.

4.1 Taylor series approximation

The Taylor series is one of the most used techniques in approximating function values. Its popularity is due to the fact that the function that needs to be estimated is approximated by a polynomial. This makes computational difficult functions easy to evaluate. A Taylor series can be seen as an infinite sum of powers. The approximation is around one point, being more accurate when the value approximated is closer to the point [16]. The precondition for a function to be approximated using a Taylor series is that it is continuous and has an *n-th* degree derivative, where *n* is arbitrarily big. The formula for the Taylor series of function f around point x_o is:

$$approx(x) = f(x_0) + (x - x_0)f'(x_0) + (x - x_0)^2 \frac{f''(x_0)}{2!} + \cdots \quad (1)$$

$$approx(x) = f(x_0) + \sum_{k=1}^{\infty}(x - x_0)^k * \frac{f^{(k)}(x_0)}{k!} \quad (2)$$

A commonly used variant of the Taylor series approximation is the Maclaurin series, the difference being that $x_0 = 0$ in the latter. This variant is more used because usually the values of a function and its derivatives of the origin are easier to compute. Equation (2) becomes for the Maclaurin approximation:

$$approx(x) = f(0) + \sum_{k=1}^{\infty} x^k * \frac{f^{(k)}(0)}{k!} \quad (3)$$

The approximation is done for a degree *p*, where p is finite so the formula becomes:

$$approx_p(x) = f(x_0) + \sum_{k=1}^{p}(x - x_0)^k * \frac{f^{(k)}(x_0)}{k!} \quad (4)$$

The value of the error in the Taylor series can be expressed in several different ways. The form known as the *Lagrange remainder* [16] is:

$$error_p(x) = f(x) - approx_p(x) = \frac{f^{(p+1)}(\varepsilon)}{(p+1)!} * (x - x_0)^{p+1} ;$$
$$\varepsilon \; between \; x_0 \; and \; x \quad (5)$$

Another approach to estimate the remainder (error) is the form named *Cauchy remainder* [17]:

$$\text{error}_p(x) = \frac{(x-\varepsilon)^p}{p!} * (x - x_0) * f^{(p+1)}(\varepsilon) \; where \; \varepsilon \; between \; x \; and \; x_0 \quad (6)$$

The formulas (5) and (6) prove that the closer x is to x_0, the smaller is the error.

The computational considerations about using Taylor Series are described in [18]. A naive implementation approach would be to choose a degree p for approximation that will cause the error to fall in the required bounds. The next step would be to choose a value for x_0, pre-compute the value in x_0 for the first p derivatives of the function f to be estimated and pre-compute the first p coefficients of the power series. The algorithm would run like this, where FD[k] is the value of the k derivative of f at x0 and C [k] is the value of the k coefficient of power series) :

```
v := f(x0)
d := x – x0
for i:=1 to p do
   v := v + d*FD[i]*C[i]
   d := d * (x-x0)
end
return v
```

This algorithm requires 2*p values to be pre-computed and 4*p+2 operations. Specific implementations for different functions can improve highly on those numbers.

4.2 Chebyshev polynomials

One downside of the Taylor series approximation is the fact that the error "tends to have an odd distribution" over the selected interval [18]. Omondi makes the claim [18] that using *first kind Chebyshev polynomials* can improve the properties of approximation. A mathematical description of Chebyshev polynomials can be found elsewhere [19]. Mathematical proofs are beyond the scope of this dissertation. The definition of a n degree first kind Chebyshev polynomial (TN) is [19]:

$$T_n(x) = \cos(n * \arccos(x)) \quad (7)$$

First degree polynomials are [20]:

$$T_0(x) = 1 \; ; \; T_1(x) = x;$$

$$T_2(x) = 2 * x^2 - 1;$$

$$T_3(x) = 4 * x^3 - 3 * x;$$

The recursive formula for determining T_n is [19]:

$$T_n(x) = 2 * x * T_{n-1}(x) - T_{n-2}(x); \quad (8)$$

This formula makes computation of Chebyshev polynomials at one point (x) accessible considering the computational requirements (memory and the number of operations needed)

First kind Chebyshev polynomials are *orthogonal polynomials* [19]. The authors [21] make the claim in their work that "occasionally power expansions whose coefficients are not determined by the Taylor expansion can operate more effectively than Taylor series itself". They also claim that using orthogonal expansion (with the help of orthogonal polynomials) is a solution better than Taylor series estimation [21]. Omondi [18] reiterates their claim and gives a practical implementation plan using first kind Chebyshev polynomials. The first step is to determine the value for x^n as a formula in terms of Chebyshev polynomials. The values for the first powers are given below [20] (a formula for arbitrary n is found too):

$$x = T_1(x)$$

$$x^2 = 0.5 * (T_0(x) + T_2(x))$$

$$x^3 = 0.25 * (3 * T_1(x) + T_3(x))$$

The second step is to choose a Taylor series expansion (using a high degree) and to replace each power of x with the corresponding formulas. The result will be a formula such as [18]:

$$approx_p(x) = \sum_{k=0}^{p} c_k * T_k(x) \quad (9)$$

The observation by Omondi [18] is that when k is large c_k is much smaller than the corresponding *k-th* coefficient of the Taylor series so the same accuracy can be achieved with a smaller degree. To demonstrate its conclusion Omondi gives an example using the e^x expansion, showing that the first 4 terms in the Chebyshev expansion give a better approximation that the first 5 terms of Taylor series approximation [18].

A naive implementation of the corresponding algorithm is similar to the one presented for the Taylor series, the difference being the recursion formula to compute T_n . As for the Taylor implementation, depending on the function to be approximated further optimizations can be done.

4.3 Newton-Raphson iteration

Using iteration is one good way of approximating the values of a function. The Newton-Raphson iteration (known also as Newton's method) is the best known and easy to use type of iteration. In order to find the value of *f(y)* the equation: $f^{-1}(x) - y = 0$ where x is the unknown has to be solved. Considering $g(x) = f^{-1}(x) - y$ then Newton-Raphson iteration has the following recurrence formula [22]:

$$x_{i+1} = x_i - \frac{g(x_i)}{g'(x_i)} \quad (10)$$

Figure 2 shows how Newton-Raphson iterations evolve towards the root. At each step a better approximation of the result is obtained

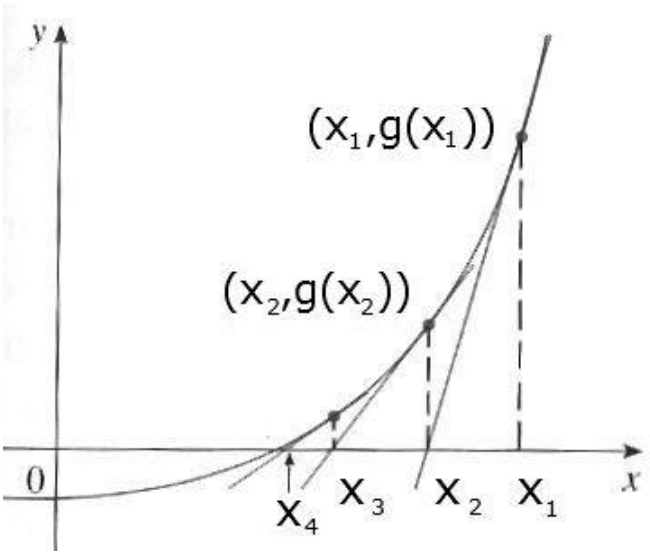

Figure 2 Newton Raphson iteration [23]

. The initial value for x_0 has to be an acceptable estimate for *f(y)*. Error analysis [24] showed that the method has a *quadratic error approximation*. This means that the error at step *i+1* is comparable to the square of the error at step *i*. Other considerations about the method and the mathematical demonstrations can be found elsewhere [22] , [24] as they are beyond the scope of this report.

From the standpoint of implementation, the algorithm depends very much on the function to be computed. Equation (10) for a given function *g* could be hard or impossible to compute. The restriction of not using division further reduces the types of functions that can be used in place of *g*. Omondi [18] presents several ways to chose function *g* for different algorithms. Algorithms to compute the inverse or the square root exist. The main advantage of this method is its *quadratic approximation*, making it an

efficient method. One downside is that the starting point should be an acceptable estimate of the value to be computed.

4.4 CORDIC algorithms

CORDIC (**CO**ordinate **R**otation **Di**gital **C**omputer) methods consist of a set of algorithms designed to compute trigonometric and hyperbolic functions. The technique was first described by Jack Volder in 1959 [25]. The purpose of the article was to describe a "unique computing technique for solving trigonometric relations involved in plane coordinate rotation" [25]. The paper described computing the sine and cosine and also provided a description of the hardware needed to perform computation. Later, John Walter published his paper [26] which contains the general description of the CORDIC algorithm and how can it be used to compute a range of different functions: trigonometric, hyperbolic, exponentiation, logarithm and square root. These algorithms were at first implemented in hardware [25] and, later, were used in software too.

CORDIC algorithms are mainly used for computing trigonometric and hyperbolic functions and this justifies their presence in this dissertation. CORDIC algorithms are simple and efficient methods that use only additions and shifts to compute the desired value. Multiplications are avoided and replaced with shifts as multiplication has a higher overhead than shift. At the time of its design (50s – 60s) multiplication was much more expensive than a shift, while for current CPUs (especially for ARM 968) the difference is not that important any more.

The CORDIC algorithms are iterating, digit-by-digit methods (linear estimation). This means that at each iteration the accuracy is increased by one digit. For example, to compute the result with 16 bits accuracy (after the binary point) the algorithm needs roughly 16 operations. This may prove to be a downside as the number of operations required can end up high (over 50). Considering the fact that only simple operations are used (addition and shift), the method has the potential to be useful.

The algorithm considers a point $\mathbf{v_0}$ in a coordinate system and starts from its set of coordinates: $\mathbf{x_0}$, $\mathbf{y_0}$ and its angle with horizontal axis $\mathbf{w_0}$. The aim is to modify the position of the point (by rotating by a fixed amount) towards a point in the space that has as a parameter (either one of the coordinates or angle) the desired value. The figure 3 shows the way the algorithm works. At the i-th iteration v_i parameters' are computed. At each step at a decision is taken whether the rotation should be clockwise or

anticlockwise. In this case if the current value of the angle is greater than the target of the rotation is clockwise and counter clockwise if it is smaller.

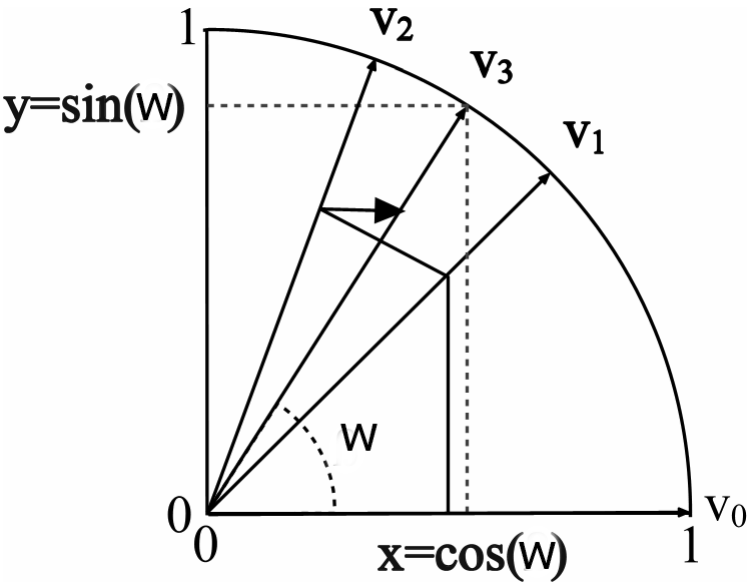

Figure 3 CORDIC description [27]

There are 2 ways of using the algorithm:

- Rotation mode
- Vectoring mode

In the **rotation mode** the angle is modified at each step to hit a certain value while the value sought for is one of the coordinates (on x axis cosine; on y axis sine). In the **vectoring mode** one of the coordinates is modified so it hits a certain value (usually 0) while the value sought for is the other coordinate. The vectoring mode is mainly used for polar to rectangular conversion [25]. Figure 3 shows the algorithm in rotation mode.

The mathematical proof of the recurrence has been shown [25] and [28]. In rotation mode (aiming to compute sine and cosine) the recurrence equations look like:

$$x_{n+1} = x_n - s * 2^{-n-1} * y_n$$

$$y_{n+1} = y_n + s * 2^{-n-1} * z_n$$

$$w_{n+1} = w_n - s * \tan^{-1} 2^{-n-1}$$

In the equations, **s** is either 1 or -1 depending if w_n is smaller or greater than the value of the angle whose sin or cosine is computed. At the end of iterations the value corresponding to **x** is the value for cosine and the value for **y** is the value for sinus. The implementation needs to use a lookup table for the values $\tan^{-1} 2^{-n-1}$.

4.5 Normalization algorithms

Another class of method that can be used, along with polynomial approximations and iteration, is *normalization algorithms*. A normalization algorithm is characterized by 2 recurrences where one converges to a constant (0 or 1 usually) and the other to the desired function [29]. A detailed description of the general technique is given in [29]. If the convergence to the constant is realized by adding or subtracting a value the technique is called *additive normalization* (the constant in this case being 0). If the convergence to the constant is made by multiplications the technique is called *multiplicative normalization* (the constant being 1).

One particular class of the *normalization algorithms* is called *De Lugish algorithms* after the name of the person who did the foundation work [18]. The particular feature of this class of algorithms is they that can be used to estimate several functions such as: logarithm, exponential or division [18]. *De Lugish* algorithms use a redundant signed-digit set [18]. The algorithm uses the following recurrences [18]:

$$D_{k+1} = D_k * d_k \quad (11)$$

$$Q_{k+1} = Q_k * d_k \quad (12)$$

$$d_k = 1 + s_k * 2^{-k} \quad (13)$$

$$s_k \epsilon \{-1,0,1\} \quad (14)$$

In those formulas D_k converges to 1 and Q_k converges to the desired values. The value for s_k depends on how close D_k is to 1. If D_k is much greater than 1 the value chosen is -1; if D_k is much smaller than 1 the value chosen is 1 while if D_k is *comparable* with 1 the value chosen is 0. The interval described by the term *comparable* becomes smaller as *k* becomes bigger. This variation of the algorithm is also called radix-2. If instead of 2 in equation (13) a number *n* had been used, the variant would have been called radix-n. The usual values for n are 2, 4 and 8. By changing the radix the redundant signed-digit set has to change (the set described in equation (14)). For example the set for radix 4 is {-2,-1,0,1,2}.

Error analysis [18] shows that the algorithm has a linear error approximation. Considering a binary system of representation this means that using a radix-2 algorithm we can obtain a precision of *p* bits after the fractional point using *p* iterations [18]. Using radix-4, a precision of 2**p* bits can be obtained using *p* iterations [18]. This is a downside

of the method. Another disadvantage of the method is the need to use a large number of pre-computed values.

4.6 Normalization to shorter intervals

Sometimes the algorithms described above require the value taken as input is in a fixed small interval rather than having a big range. For example, the Taylor series approximation works better when the input interval is small and around one fixed value (0 or 1 usually). The Newton-Raphson iteration needs a good approximation of its starting value in order to perform better, and simple polynomial approximation of function can easily be done when the range of the input is rather small.

As seen in the methods described above normalization to the [0.5, 1) interval offers all the benefits described above. Normalization to that interval can be done using the following formula:

$$x = x^* * 2^k$$

In the formula above x^* is in the interval [0.5,1) and k is an integer. In order for this normalization to be useful, the function f to be computed for x should have the following property:

$$f(x) = f(x^* * 2^k) = g(f(x^*), f(2^k))$$

Where $f(2^k)$ is easily computed (worst case is read from a lookup table) and function g is easy to evaluate (it involves additions or multiplications). Depending on the function f further optimizations can be achieved in order to reduce the number of operations needed.

It is important to note that, depending on the function g, the error with which $f(x^*)$ is computed should be less than the acceptable error so the overall error for $f(x)$ is acceptable. In other words, if g represents an addition the error at which $f(x^*)$ is computed is the same as the overall error. If g represents a multiplication the error rises depending on the value: $f(2^k)$. Further considerations will be made for any particular case as it greatly depends on the functions f and g.

The implementation of normalization proves to be rather difficult. The way to compute it is to find the number of leading-zeros in number's binary representation. Some general techniques involving a C implementation have been described in [30]. These methods require at least 10 operations. This number is rather large taking into

consideration that it is only an initial step for one algorithm. In the case of ARM assembly, there is a special instruction described earlier in the report named *Count Leading Zeros*. This means that the optimized ARM library can make use of this instruction reducing the number of operations by 9.

Depending on the algorithm, normalization to custom intervals can greatly improve the performance. One example is normalization to a small interval around a value for which the Taylor expansion is easily calculated. Another example is normalization to a small interval for which several intermediary values are saved in a lookup table.

4.7 Use of lookup tables

Lookup tables are one of the first methods used to compute arithmetic functions. The principle is simple: having the input, you are looking for the closest entry in the table and read the value corresponding to it. Modern algorithms are using lookup tables more efficiently. Instead of mapping a full interval with some step, they keep a smaller set of intermediary values and use them in combination with other methods. For example, a lookup table can be used to obtain a good starting value for an iterating algorithm such as Newton-Rapshon. Another example would be using Taylor series expansion around the closest value to the input. It has been shown earlier that for Taylor expansion, the closer the input is to the value around which the expansion is done, the better the output approximates the actual value.

There are 2 main overheads in the use of lookup tables:

- Space occupied by the table
- Computations needed to find the entry in the table

In the current context (SpiNNaker and the ARM 968), where important constraints are put on both space and the number of operations required, these overheads should be treated with attention.

In order to reduce the size of the lookup table several observations should be taken into consideration. The first observation is that the size of the lookup tables should be very small. Considering the total space constraint (less than 1 KB), the lookup tables should have a number of entries in the order of tens (10 – 100). A second observation is that, where possible, lookup tables should be reused. This means experimenting with different algorithms for operations to find possibilities for reusing lookup tables.

Reducing the number of operations needed to find the entry in the table is also important in order to keep the algorithm efficient. A simple way of achieving that is that entries in the lookup table should be like: $entry_k = k * 2^a$; where *a* is fixed number. Using this approach if the input has *b* digits then the first *b-a* digits represent the entry in the lookup table *k*.

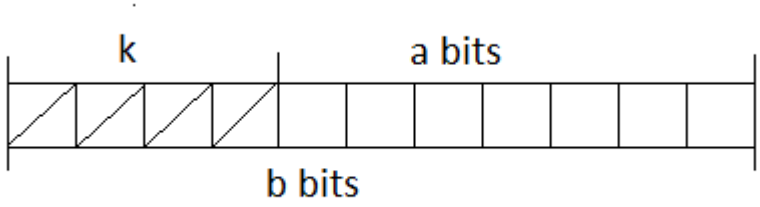

Figure 4 Input bit representation

Figure 4 shows the bit representation of an input. In this, case **a** is 7 and **b** is 11 meaning that the first 4 digits determine the table entry and also that the size of the table should be $2^4 = 16$ entries.

4.8 Summary

This chapter summarizes the general techniques that might be useful for implementing algorithms for the operations required. The presentation of each method is brief, focusing on the potential impact on the project. Most of the mathematical details are referred to in other academic works because such a presentation does not represent an objective of the project.

A comprehensive set of methods was detailed. These methods will be used to describe the algorithms that will be implemented. The next chapter "Algorithm assessment" is using the findings in this chapter to describe the algorithms.

Chapter 5 Algorithm assessment

This section presents the algorithms devised for each operation considered. An analysis of accuracy and efficiency for each algorithm is described. The set of operations considered is: division, square root, logarithm, exponential and sine.

5.1 Division

The first operation analysed will be division, as it is the most important. In the literature a large number of ways to compute *a* / *b* can be found, but this subsection will discuss the following ones:

1. *Newton-Raphson iteration*
2. *Functional iteration*

Both approaches follow the same basic flow: normalization to the [0.5, 1) interval, computing the inverse and constructing the result. The methods differ in the way they compute the inverse. A graphical representation of both algorithms is presented in Figure 5.

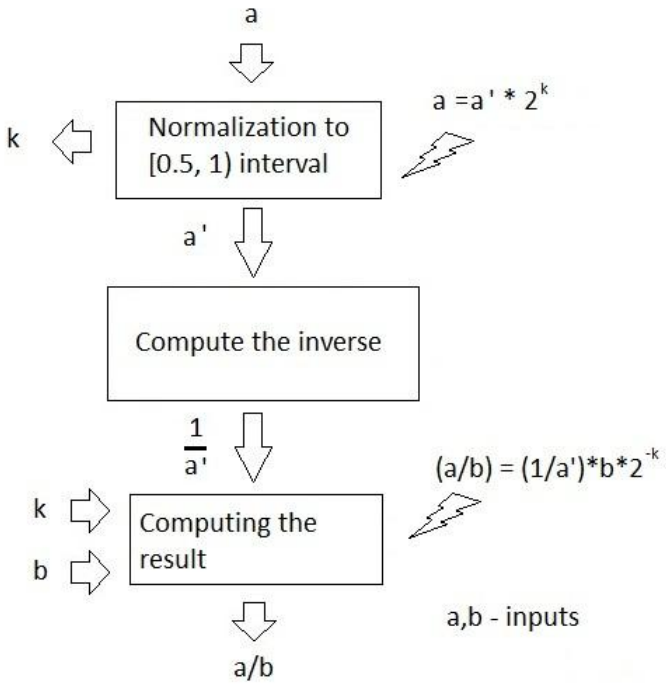

Figure 5 Graphical representation of division algorithm

In Figure 5, the input for the algorithm is **a** and **b** and the output should be **a/b**.

The *Newton-Raphson iteration* method for division is described elsewhere [18]. Considering *a* / *b* the operation to compute, the method first computes *1* / *b* and then multiplies the result by *a*. The function *g* used for iteration is:

$$g(x) = b - \frac{1}{x} \quad (15)$$

$$g'(x) = \frac{1}{x^2} \quad (16)$$

This means that the recurrence looks like:

$$x_{i+1} = x_i * (2 - b * x_i) \quad (17)$$

The algorithm's efficiency depends on the starting value x_0. In order to accommodate that Omondi used normalization to the [0.5,1) interval [18]. This means writing *b* as:

$$b = b^* * 2^k \text{ where } b^* \epsilon [0.5,1) \text{ and } k \epsilon \mathbb{Z} \quad (18)$$

In formulas (15) and (16) *b* will be replaced with b^* . In order to compute *1* / *b,* the result obtained using the iteration will be multiplied by 2^{-k} . The initial value x_0 should be an estimate of *1* / b^* . The formula used in [18] is:

$$x_0 = -\frac{32}{17} * b^* + \frac{48}{17} \quad (19)$$

The algorithm was implemented using 4 iterations. The following plot is the error result:

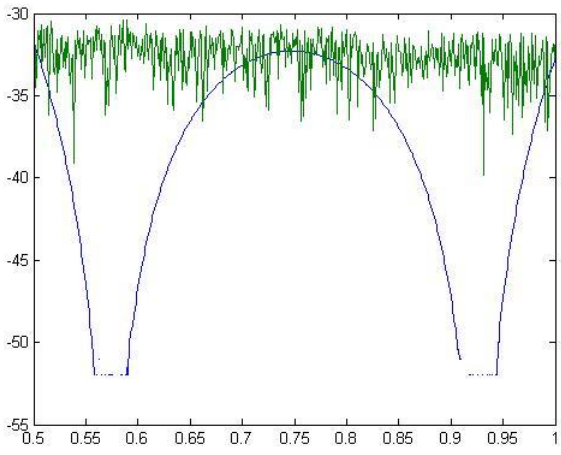

Figure 6 Division Newton Raphson

The plot in Figure 6 is constructed using the methods described in the Research Methods chapter. The *blue* line on the plot is the error representation when using infinite (i.e. large number) precision in bits after the fractional point. The green line in the plot is

the error representation when using 31 bits of precision after the fractional point. It can be seen from the plot that the error oscillates around -31 (31 bits of precision).

A pseudo-code version of the algorithm is described in Appendix A. The cost of the algorithm in operations is: 13 + no of operations needed for normalization (see subsection about normalization).

Functional iteration computes first *1 / b*, and then obtains *a / b*. A normalization like the one in formula (18) is done. The approximation used is:

$$\frac{1}{1+x} = (1-x) * (1+x^2) * (1+x^4) * \dots \quad (20)$$

Making notations: $t = b^* - 1$ and $y = 1 - t$ then:

$$\frac{1}{1+t} = \frac{1}{b^*} = y * (1+t^2) * (1+t^4) * \dots \quad (21)$$

Using 4 iterations the result was:

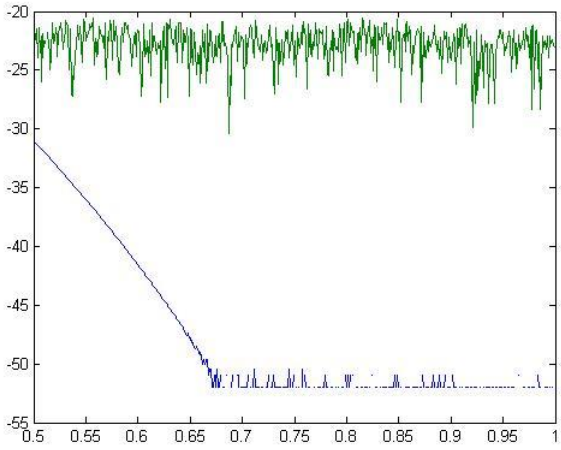

Figure 7 Division Functional Iteration

As in Figure 6 the *blue* and *green* lines have the same meaning. The thing to note is that even if the precision was 31 bits and the ideal error was smaller than the precision, the values of the error using finite precision for intermediate results are around -23. This means that implementation details can affect the value of the error.

A pseudo-code version of the actual algorithm is described in Appendix A. The cost of the algorithm in operations is: 16 + no of operations needed for normalization (see subsection about normalization).

The Newton-Raphson iteration proves to be a better choice for the software implementation as a better error value is obtained using roughly the same number of operations.

5.2 Square Root

Computation of square root can be done in several ways [18]: two different Newton-Raphson iterations or using pseudo-division algorithm. Parhami describes a method to compute square root using *additive and multiplicative normalization* [29]. Due to the space limitation for this report, this chapter will describe the most efficient of them, making considerations regarding the others.

Figure 8 describes the square root algorithm chosen. There are 3 stages: normalization to [0.5,1), computing the inverse of the square root and computing the final result.

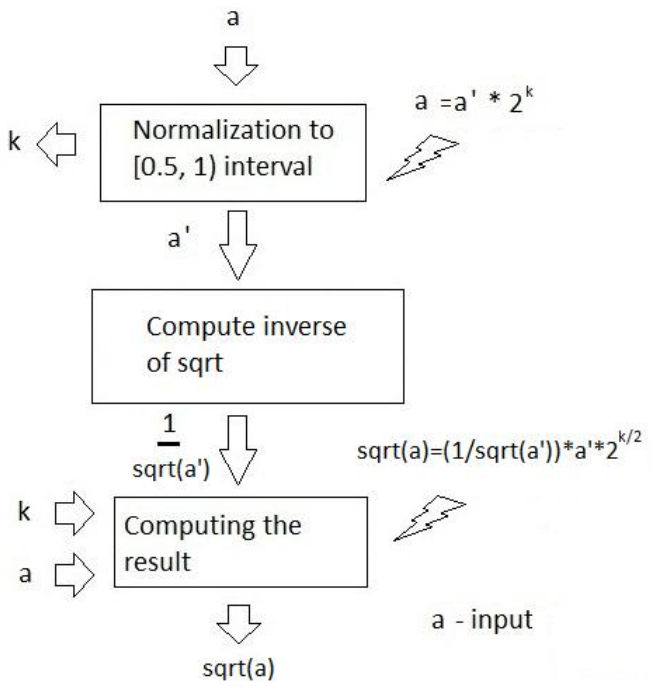

Figure 8 Square root algorithm description

The algorithm is to compute $\frac{1}{\sqrt{a}}$ using Newton-Rapshon iteration and then multiply the result by *a* to obtain the square root [18]. Because estimating an initial value is easier and better for numbers in [0.5, 1), a process of normalization will be done to reduce *a* to

this interval. Considering formula (18), if *k* is odd the final result is obtained by shifting the intermediary result with $[k/2]$ and multiplying by $\sqrt{2}$:

$$\sqrt{a} = \sqrt{a^*} * 2^{[\frac{k}{2}]} * p \text{ where } p = 1 \text{ when k even and } p = \sqrt{2} \text{ when k odd} \quad (22)$$

The set-up for the Newton-Raphson iteration is:

$$g(x) = \frac{1}{x^2} - a \quad (23)$$

$$g'(x) = -\frac{2}{x^3} \quad (24)$$

$$x_{i+1} = 0.5 * x_i * (3 - a * x_i^2) \quad (25)$$

Omondi uses the following approximation of the starting value [18]:

$$\frac{1}{\sqrt{a}} \cong x_0 = 1.78773 - 0.80999 * a \quad (26)$$

The following plot shows the error of computing $\sqrt{a}$ for *a* in [0.5, 1) using 3 iterations:

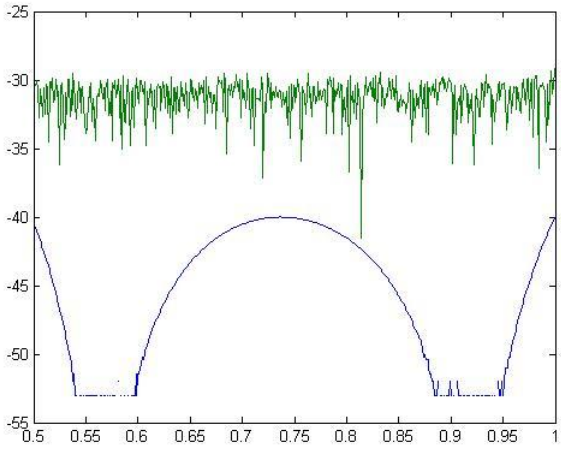

Figure 9 Square Root - Newton Raphson

As on the figures before the *blue* and *green* lines have the same meaning. One observation to make is that the precision used for the intermediate results was 2 bits for the integer part and 30 for the fractional part. These values can change at implementation time. The error obtained oscillates around the value of 31. Considering the shifting that should be done as in formula (22) the error for a number around 2^{30} will be around -31+30/2 = -16, the desired value.

A pseudo-code version of the algorithm is found in Appendix A. The cost of the algorithm in terms of operations is: 19 + cost of normalization.

The other variant of Newton-Raphson iteration uses division on the recurrence formula. The implementation introduces another recurrence formula to replace division by multiplication to reciprocal. The approach is not as fast as the one presented above, needing 6 iterations to obtain the same accuracy. The pseudo-division algorithm has a linear evolution of error while the De Lugish variant can be faster for higher radices (4 or 8) but the evolution is also linear.

5.3 Logarithm

For computing *logarithm* three techniques were investigated: *Taylor series approximation, Chebyshev polynomials expansion* and *De Lugish algorithm*. The first method is the most efficient and will be detailed, while the other 2 methods will be briefly presented.

Taylor series approximation uses the following formula (Maclaurin expansion) [18]:

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \cdots + (-1)^{n+1}\frac{x^n}{n} \quad (27)$$

The formula gives good result for x around 0. In order to accommodate the constraint, normalization of a to the [0.5,1) interval is done like the one in formula(18):

$$\ln(a) = \ln(a^*) + k * \ln(2) \text{ where } a^* \in [0.5,1) \text{ and } k \in \mathbb{Z} \quad (28)$$

Making the following replacement: $t = a^* - 1$ the formula (27) becomes:

$$\ln(a^*) = \ln(1+t) = t - \frac{t^2}{2} + \frac{t^3}{3} - \cdots + (-1)^{n+1}\frac{t^n}{n} \quad (29)$$

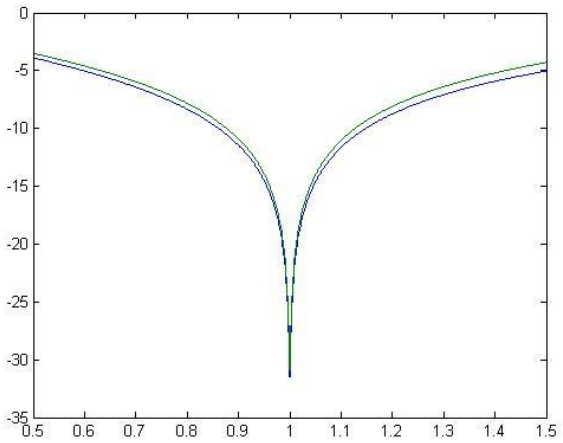

Figure 10 Logarithm Taylor series 2 and 3 terms

The plot in Figure 10 shows the value of the error when using the above approximation with 2 and 3 terms. The plot is for the interval [0.5,1.5] to show the relative symmetry of the error:

The *blue* line stands for the first 2 terms while the *green* line stands for the first 3 terms. An observation is that increasing the number of terms to 4 or 5 does not improve the approximation given by the first 2 terms. One thing to note is that using just 2 terms gives better results for error than using 3 terms. Another important observation is that for the most of the [0.5,1) interval the value of the error is unacceptable, making this version not useful.

An important observation to make is that the error is acceptable (lower than -16) for the interval [0.97,1.03]. A variant of the algorithm is developed using a function *f* with the following properties:

$$f(x) = a_i * x \in [0.97,1.03] \quad \text{when } x \in [s_i, s_{i+1}) \quad (30)$$

The values for s_i and a_i are pre-computed and stored in a lookup table. For lookup optimization s_i will have the following formula: $s_i = 0.5 + i * 2^{-q}$ where $q = 5$. The formula used for computation is:

$$\ln(x) = \ln(f(x)) - \ln(a_i) \quad \text{when } x \in [s_i, s_{i+1}) \quad (31)$$

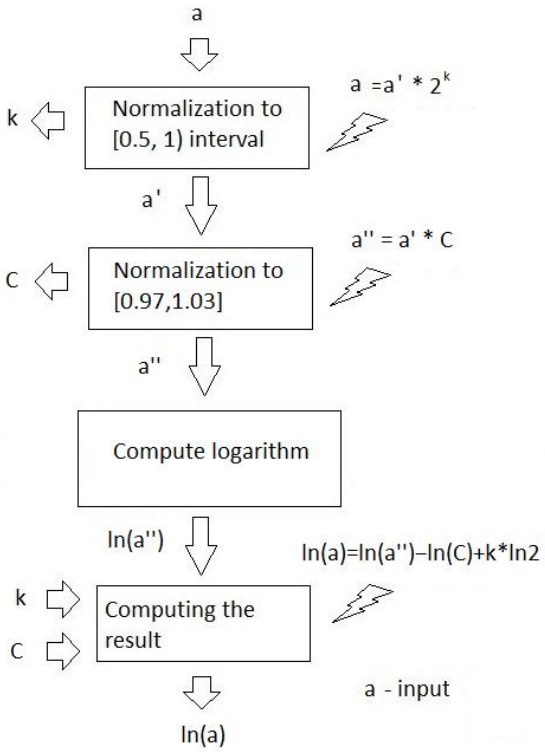

Figure 11 Logarithm algorithm description

Figure 11 describes the algorithm used. The first 2 stages represent normalization to a smaller interval; the 3rd stage is computing the logarithm while the last stage is constructing the result.

The following plot shows the value of the error obtained:

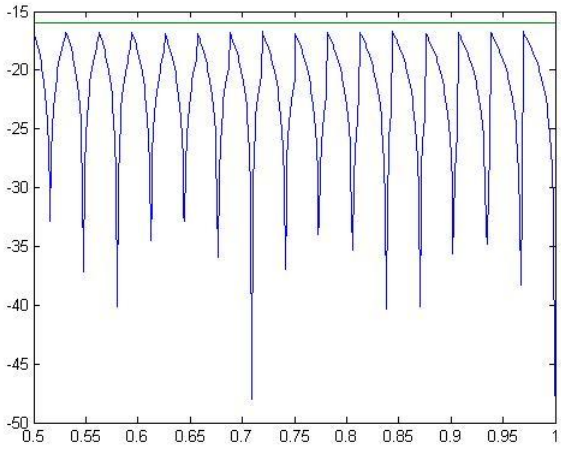

Figure 12 Logarithm Taylor series variant

The *blue* line shows the evolution of the error over the interval while the *green* line sets the limit of acceptable error (-16).

A pseudo-code version of the algorithm is described in Appendix A. The algorithm is very efficient from the number of operations point of view: 10 + normalization cost. A lookup table with 16 entries with 2 values each: a_i and $ln(a_i)$ will be needed.

The *Chebyshev polynomials expansion* method has been detailed in the previous chapter. In order to obtain a good enough error plot, the first 7 terms of the expansion have been used. The error was plotted between 0.5 and 1.5. The values obtained oscillate between -5 and -5.26, being almost constant for the [0.8,1.2] interval. One improvement brought was to use a similar function as the one described in (30) to move the interval [0.5,1] to [0.8,1.2] and subtract the value of the error (which on that interval was almost constant). The result obtained was acceptable with a maximum error of -15. The cost of the algorithm in terms of number of operations is: 37 + normalization cost, which is almost four times higher than the Taylor series method. The *De Lugish algorithm* [18] has a higher cost in operations than the other 2 methods because of its linear evolution of the error.

5.4 Exponential

Computing the exponential, e^a, resembles computing the logarithm. First the following normalization takes place [18]:

$$a = a * \log_2(e) * \ln(2) \quad (32)$$

$$I = [a * \log_2 e] \text{ and } F = a * \log_2(e) - I \quad (33)$$

$$e^a = e^{\ln(2)*(I+F)} = e^{\ln(2)*I} * e^{\ln(2)*F} = 2^I * e^{\ln(2)*F} \quad (34)$$

The problem is reduced to computing e^y where y is between [0,1). The solutions available are: using Taylor series, using Chebyshev polynomial expansion and using De Lugish algorithm. After analysing the 3 methods, the Taylor series approximation proves to be the best choice. The following approximation is used [18]:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots \quad (35)$$

Analysing the graphs of errors proved that using 3 terms has the same accuracy as using more than 4. Also the analysis proved that the spread of error is not acceptable with the exception of a small interval around 0 (as was the case with logarithm in the previous subsection). Using the following formula the [0,1) interval was reduced to a smaller interval around 0:

$$y = (p * 2^{-k} + 2^{-k-1} + t) \; for \; y \epsilon \; [p * 2^{-k}, (p+1) * 2^{-k}) \; with \; t \; \epsilon \; [-2^{-k-1}, 2^{-k-1}) \quad (36)$$

$$e^y = e^{p*2^{-k}+2^{-k-1}} * e^t$$

The values for $p * 2^{-k}$ and $e^{p*2^{-k}+2^{-k-1}}$ are pre-computed and stored in a lookup table.

Description of the algorithm is provided in Figure 10. The first step is to pre-compute **I** and **a' = F*ln(2)**. The second step is to reduce the interval to a smaller one [0, 2^{-k}). The third step is to compute the logarithm, while the fourth and last step is to construct the result.

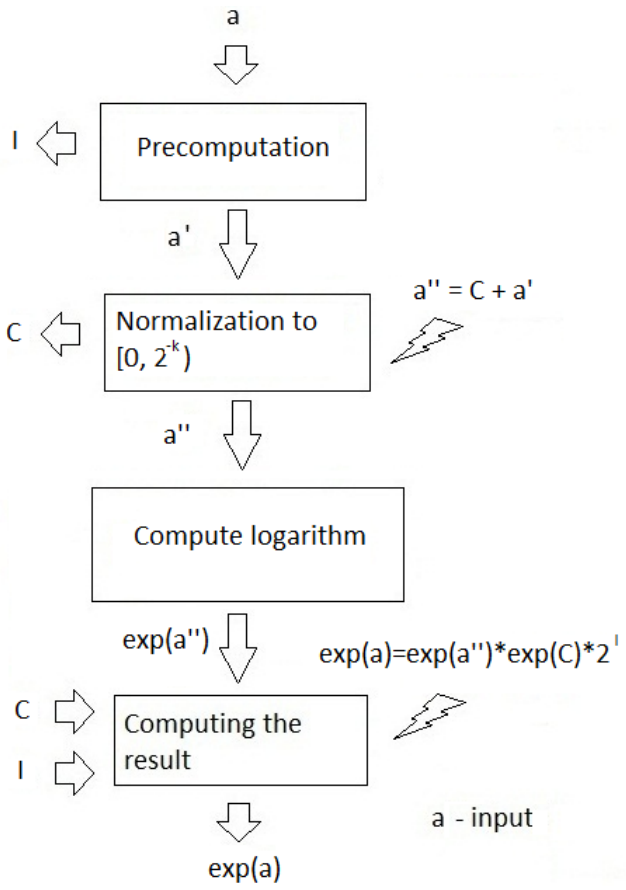

Figure 13 Exponential algorithm description

The implementation chosen used $k = 4$, obtaining the following plot:

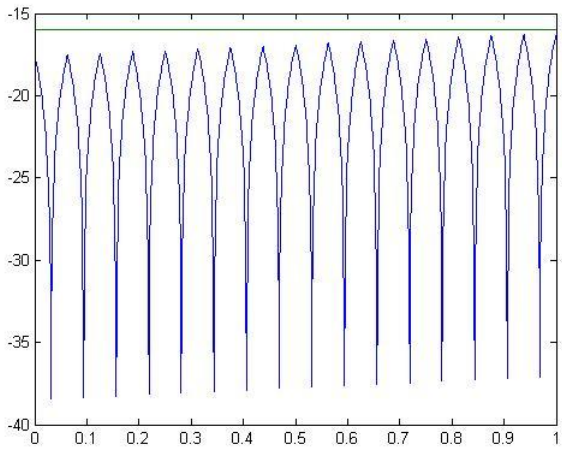

Figure 14 Exponential Taylor series variant

The *blue* line shows the evolution of error over the interval while the *green* line sets the limit for an acceptable error (-16). To note is that for the interval [0,1) the error is acceptable, but when considering formula (34) for higher numbers the error surpasses the acceptable error level. In the implementation stage further optimizations will be

implemented to decrease the error for higher numbers. The cost of the algorithm in number of operations is 11, making it very efficient. The lookup table has 16 entries with 2 values each.

As in the case of logarithm, the Chebyshev expansion method and De Lugish algorithm are more expensive in terms of operations and do not offer a better error performance.

5.5 Simple trigonometric functions – sine and cosine

Computing trigonometric functions is eased by the mathematical formulas that connect them. For example, computing cosine of a number being able to compute sine can be done in several ways. The most efficient method is illustrated by the following equation:

$$\cos x = \sin(\frac{\pi}{2} - x)$$

It shows that computing cosine is just one operation more expensive than computing sine. The focus of the next part will be on describing results obtained for computing sine.

In computing trigonometric functions, using mathematical formulas available can greatly improve the efficiency and the accuracy of the algorithm. It is important to use the formulas that can be computed in the context (i.e. formulas involving just additions and multiplications). Table 3 summarizes the formulas useful in this context together with an explanation of the value brought:

Formula	**Value**
$\sin(x + 2 * \pi) = \sin x$	Reduces the interval to a smaller one
$\sin\left(x + \frac{\pi}{2}\right) = \cos x$ $\sin(x + \pi) = -\sin x$ $\sin\left(x + \frac{3 * \pi}{2}\right) = -\cos x$	Reduces the interval to the first quadrant
$\cos x = \sin(\frac{\pi}{2} - x)$	Reduces the interval to the half of first quadrant

$\sin 2x = 2 * \sin x * \cos x$ $\cos 2x = 2 * \cos x * \cos x - 1$	Cuts the value to be computed in 2

Table 3 Useful trigonometric formulas

Table 3 shows that computing sine and cosine for the interval $[0, \frac{\pi}{4}]$ and using the formulas above, sine and cosine of any another value can be computed. Using the last 2 formulas in the table this interval can be made smaller. In order to illustrate that, if we consider using 3 times the formulas the interval becomes $[0, \frac{\pi}{32}]$. This would mean that computing sine or cosine of any value is equivalent to computing sine and cosine of a value in the interval $[0, \frac{\pi}{32}]$. One thing to note is that to maintain accuracy when using the last 2 formulas, the intermediary results have to have a higher accuracy than the one aimed for. Independent of the algorithm used, the observations above will greatly improve it because of the much smaller interval on which the algorithm has to run.

For computing sine, 3 different algorithms were tested and analysed:

1. **Taylor series** expansion
2. **Power series** expansion
3. **CORDIC** algorithm

The **Taylor series** expansion uses the following formula for approximating the sine and cosine:

$$\sin x = \sin a + (x-a) * \cos a - \frac{(x-a)^2}{2} * \sin a + \frac{(x-a)^3}{6} * \cos a$$

$$\cos x = \cos a - (x-a) * \sin a + \frac{(x-a)^2}{2} * \cos a - \frac{(x-a)^3}{6} * \sin a$$

In the formulas **a** represents the value around which the Taylor series expansion is computed. The formulas show the first 4 terms of the Taylor expansion. The values for sine and cosine should be stored in a lookup table.

The algorithm used has 3 stages. First, a corresponding value of the input is computed for a small interval. Second, sine and cosine of that value is computed. Third, sine of the input is obtained. A graphical representation of the algorithm is presented in Figure 15.

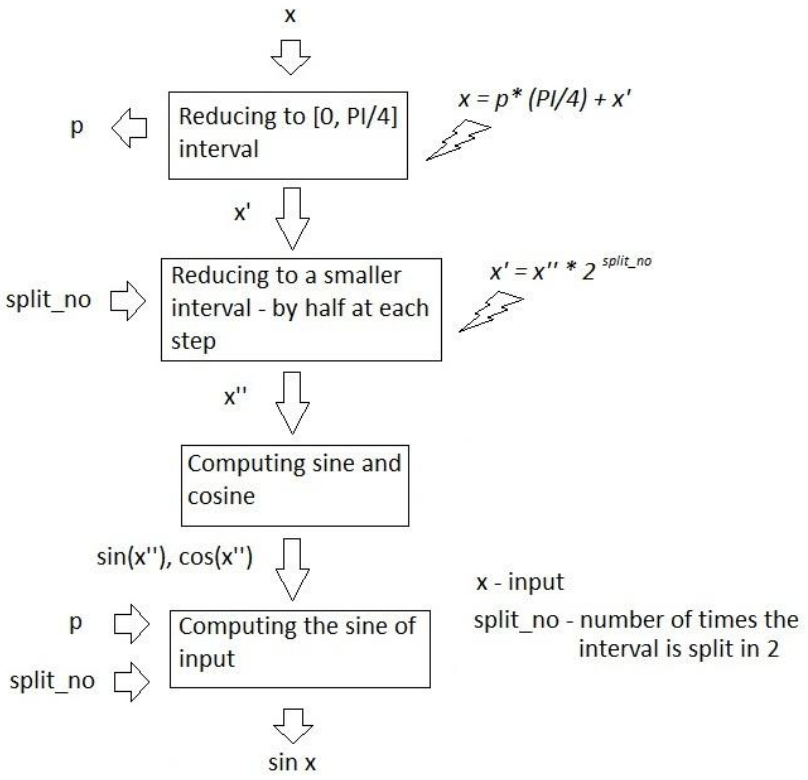

Figure 15 Graphical description for an algorithm computing sine

The parameters that influence the algorithm are: **split_no**, size of the lookup table and number of terms use in the approximation (first 3 or 4 terms). A synthesis of accuracy obtained by modifying the parameters above is shown in Table 4. The table contains the values for the maximum error obtained when using 3 terms (first value) and 4 terms (second value). The error is represented as described in chapter 3.

Split_no / **Lookup table size**	2	3	4
4	-16; -25	-18; -30	-20; -33
8	-20; -30	-21; -34	-23; -38

Table 4 Maximum error for Taylor series - sine algorithm

The values in the table were obtained by implementing the algorithm in Matlab and exploring the errors (as described in chapter 3). Considering that the result should be accurate to at least 24 bits after the binary point (8.24 format) and the overhead caused by a large split_no or a big lookup table, a good choice of parameters would be: split_no = 3; lookup table size = 4 and using the first 4 terms of the expansion. In this case the result should be accurate to 30 digits after the binary point. The number of instructions

is between 10 and 20 (first stage implementation cannot easily be estimated before the implementation step) and the size of the lookup table would be 32 bytes (8 entries of 4 bytes each – 0.32 format used).

The **Power series** expansion uses the following approximation formulas for sine and cosine:

$$\sin x = x - \frac{1}{6} * x^3 + \frac{1}{120} * x^5$$

$$\cos x = 1 - \frac{1}{2} * x^2 + \frac{1}{24} * x^4$$

The algorithm used with power series expansion is very similar to the one used for Taylor series expansion. The process is described in Figure 15. The difference consists of how the sine and cosine are estimated in the step "*Computing sine and cosine"*. While in the previous case Taylor series expansion was used, in this algorithm the formulas for power series expansion are used. This approach can be better suited to the context as it does not require the use of lookup tables. The parameters that influence the accuracy and efficiency of the method are: **split_no** and the number of terms used in power series expansion (2 or 3). Table 5 shows the value of maximum errors depending on the parameters:

Split_no / **Terms used**	2	3	4
2	-14	-18	-22
3	-23	-30	-36

Table 5 Maximum error for power series - sine algorithm

The values in the table were obtained by implementing the algorithm in Matlab and examining the errors obtained. Considering the requirements on precision a good choice of parameters would be: **split_no** = 3 and terms used = 3 giving a precision of roughly 30 bits after the decimal point.

Using the power series expansion presented several advantages over using the Taylor series expansion. Looking at the best choices of algorithm both provide roughly the same accuracy: 30 bits after binary point. Also, both used the same value for split_no. The difference is in the number of terms used in each expansion (3 in the power series versus 4 in the Taylor series). This has an impact on the number of operations needed

to compute the estimated value. Also, the power series method does not require the use of lookup tables.

In Figure 16, the plot of the algorithm chosen (power series expansion with best parameters) is shown. The green line sets the minimum accuracy needed while the blue line describes the evolution of the error.

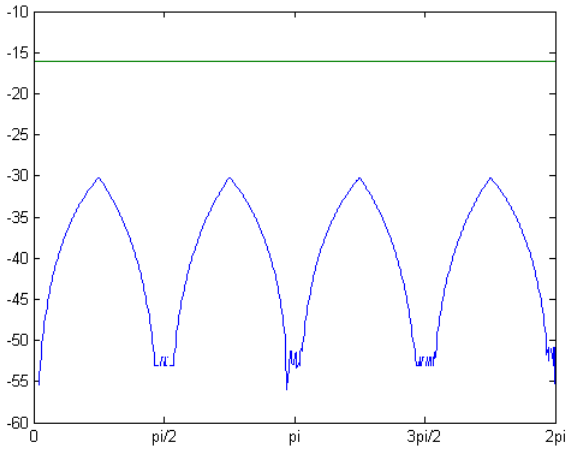

Figure 16 Error plot for sine computation

It was observed that by modifying the parameters of the algorithm the plot shape remains roughly the same, the difference being that it is translated vertically towards better results or worse.

The implementation of the CORDIC method does not make use of the reducing interval formulas describe above. The reason is that the method is a **digit-by-digit** one, meaning that one bit of accuracy is gained at each step. Using the split into half the interval will only increase the accuracy by one bit, making the effort useless as the same can be achieved by implementing an additional step in the process. Useful for this algorithm is the first step in the previous algorithm, "*Reducing to [0, PI/4] interval*". The implementation of the algorithm showed that the same accuracy as for the previous algorithms can be obtained but with a larger number of computations. Even if each step requires a small number of operations (roughly 3 to 5), the number of operations needed is big, proving the linear characteristic of the algorithm.

Number of iterations	10	15	20	25	30
Maximum error	-9	-14	-19	-24	-29

Table 6 Maximum error for CORDIC - sine algorithm

Table 6 shows the results of the implementation in Matlab. It reveals that at least 25 iterations are needed for the desired accuracy – meaning at least 75 operations. Considering this fact it can be stated that the CORDIC algorithm does not fit the requirements.

5.6 Summary

In this chapter a description of the work done in assessing the algorithms was presented. For each operation several potential algorithms were implemented and analysed using Matlab. The aim was to obtain a rough estimation of the cost (in terms of operations and size of data necessary) and one of the accuracy (the number of accurate bits after the binary point). For each operation the best method was detailed and a plot of the error was added to give a better sense of the results. Also, the results of assessing the other approaches were commented on.

Several learning outcomes were obtained in this phase of the project. First, estimating the number of operations needed for an algorithm can only give ranges on which the value can reside. An example would be the normalization to the [0.5,1) interval. This kind of task can prove to be easier or tougher on different architectures. Second, estimating the lookup table size can be very precise as the length is usually a parameter in the algorithm.

Some conclusions can be drawn from surveying the results obtained. The first thing to note is that not all the methods described in chapter 4 are useful. More precisely, the techniques that exhibit linear estimation (digit-by-digit) are proven to be very slow, hence using a huge number of operations. In this category resides: De Lugish algorithm and CORDIC method. On the other hand, Newton-Raphson iteration and Taylor series expansion proved to be a fast and efficient method for a good range of operations. Chebyshev polynomials proved to bring an overhead to Taylor series expansion therefore being rather inefficient (probably the value brought can be seen when accuracy should be very high – 64 bits). Another important observation is that preparatory mathematical work is needed for almost every operation, as it can be seen in exponential and sine computation.

Chapter 6 Implementation

In this chapter a description of the implementation stage will be provided. First a theoretical description of the testing approach is presented. Next, the description of the implementation of the testing framework is presented. A short presentation of the libraries implemented is followed by details regarding each of the implementations. This chapter presents the way the algorithms described earlier are implemented using C.

6.1 Testing approach

One important challenge to tackle in the implementation phase was the validation of the results. This refers to finding a strategy to verify that the code written implements the algorithm and, moreover, the results are the ones expected. In order to approach this challenge, 2 techniques were used:

- Unit testing
- Test-first approach

Unit testing consists in testing the smallest unit of functionality in the code. A definition is : "A *unit test* is an automated piece of code that invokes *a unit of work* in the system and then checks a simple assumption about the *behaviour* of that unit of work" [31]. Unit testing is commonly used in object-oriented programming, but it proves useful for procedural and functional programming too. In this context the smallest unit of code represents one function. The units tested were: the algorithms implemented and the other additional functions implemented. An important aspect of the definition is the **automated** characteristic of the test. A unit test framework usually enables running a series of automated tests. In the current context, the suite of unit tests built for the algorithms can also be used as regression tests. This is relevant as the implementation of the algorithm can change over time (to be optimized or enhanced), and a mechanism of ensuring that it keeps its behaviour is needed.

Test-first approach is a relatively new technique of handling testing in software development. It is advocated by the Agile movement [32] and known as Test-Driven Development. Test-first approach means building the tests before implementing any code. This technique is well suited to large, complex applications developed in a higher level programming language, but presents several advantages in the current context. One of them is that the content of the test is not biased towards passing the code. Another advantage would be that development is easier since the changes between

writing tests and subsequently writing code are minimal (write test, write code *versus* write code, write test, modify code to pass the test).

The structure of the unit tests used in the project has been tailored to suit the context. The main task of the suite is to ensure the correctness of the results of the algorithms implemented. Given the huge number of possible inputs (e.g. division for 16.16 has 264 possible input combinations), a way to have a large number of inputs tested is required. The solution thus chosen was creating a separate file containing the inputs and correct outputs, instead of writing code for each operation tested. The testing code then evaluates the function by supplying it with the inputs from the file, and comparing the result obtained with the output from the file. Figure 17 describes the process.

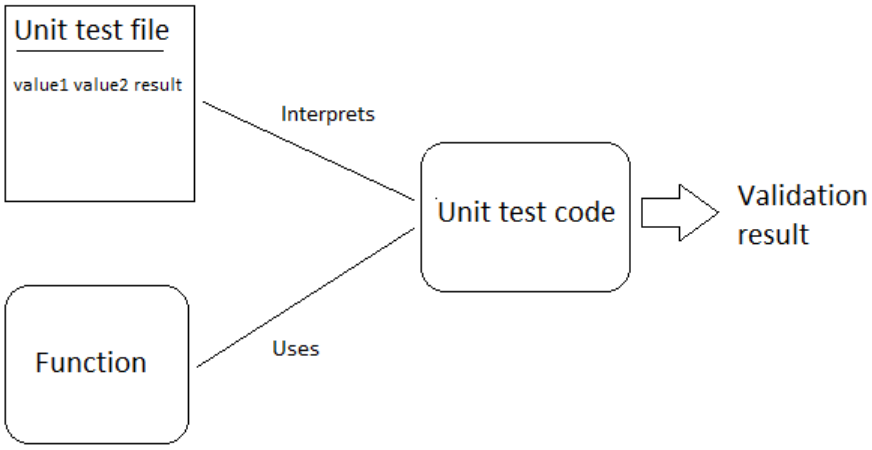

Figure 17 Unit test description

One important aspect of the unit test is the content of the files, which has to satisfy 2 important requirements:

- Having a large breadth of cases
- Handling edge cases

Each unit test is designed for the algorithm it has to test, but all of them share some common characteristics, to tackle the requirements above. In order to cover a large number of possible cases a number of random test-cases are generated. These tests should handle the most frequent inputs. Edge cases represent uncommon types of input that have the potential of being hard to compute, which may leave the system in a state of error. Edge cases exist for most operations. For example, an edge case for division

would be when the dividend is very high (close to the maximum possible) while the divisor is very close to 0 (zero). The edge cases are specific to the type of operation being performed. Details of the edge cases considered for each operation will be presented in the "Evaluation" chapter.

The number of tests generated has an impact on the size of the file and on the testing time. While at the end of the project the time needed to process the tests is not important, during the development it is an issue as running tests is done several times during implementation. This is the reason behind starting with a small number of test cases (100) replacing it with a higher number (1000) at the end.

Validation of the result of the function can be done in 2 different ways:

- Exact match
- With an absolute error

When using an **exact match** validation, a test is considered passed (also referred to as "green") when the output of the function is exactly the value expected. This kind of validation is used for basic operations (add, subtract, multiply, fractional part or integer part) and for auxiliary functions.

Validation can also be done with **an absolute error**. This means that a test is considered "green" when the output of the function is within a certain range from the expected value. This type of validation is used for functions implementing the algorithms. This approach is useful as a perfect method of computation is very hard to implement and would be computationally expensive. The 2 different types of validation are described by Table 7.

Exact match	output = expected_value
With an absolute error	|output – expected_value| <= max_error

Table 7 Types of validation used

A test is passed if the condition on the right is met. In the formulae, *output* is the output of the function tested, *expected_value* is the result expected and *max_error* defines the magnitude of the range within which the output should be.

The contents of a test file should be easy to understand by the users. There are 2 reasons behind this. First, when a test fails the developer may want to see the exact inputs and outputs that failed (for the cases where the validation code does not give

information about that). Second, a user may want to create, modify or delete a test file by hand in order to try some particular cases. The file should also be easy to generate programmatically. The structure of a test is described in Figure 18.

8_8/divide.test

Title: 8_8 divide#
Operands: 2#
Type of operands: FIXED POINT BINARY#
Type of result: FIXED POINT BINARY#
#0 11111000.10011100 01110011.11100010 11111111.11101111
#1 11011011.11110110 01111111.11111101 11111111.10110111
#2 10100000.01110110 10110011.11000001 00000001.01000000
#3 10111110.10100110 11101010.11000100 00000011.00010011
#4 00001001.10110101 10000000.00001111 11111111.11101100

Figure 18 Test file structure

Figure 18 shows the structure and content of the test file for non-saturated division of 8.8 format. It can be observed in the file description that the initial part describes the metadata that provides the information about the contents of the test file. The **title** describes the operation being tested (name of operation, fixed point format, saturated or non-saturated). **Operands** tag defines the number of operands of the operation (1 or 2). **Type of operands** tag is followed by the format, which can be one of *FIXED POINT BINARY, INTEGER* or *FLOATING POINT*. **Type of result** tag describes the data type of the result and has the same options as for **type of operands.** The metadata is followed by the test cases, where each new line represents a single test case. The format of the line is such that it starts with a "#" character followed by a number that uniquely identifies the test case. This is followed by the one or two operands and the expected output, each separated by single spaces. The inputs and the output are of the type defined in metadata.

6.2 Testing framework

Commercial off-the-shelf unit test frameworks exist for almost for every high-level programming language. In the case of C the offer is limited in both numbers and features. The testing approach described is also different from the common use-case because of the large number of assertions to be used, validation of results with error, as

well as interpreting a test file. This led to the decision to implement a testing framework to implement the testing approach detailed earlier.

The system would have 3 distinct parts, each with a specific purpose and functionality:

- Test file generator
- Test file interpreter
- Unit test implementation

Together, all these subsystems should work as an automated testing framework. This framework will generate tests, interpret the test files and give the result of the validation of the unit being tested. The way the parts work as a whole is described in Figure 19.

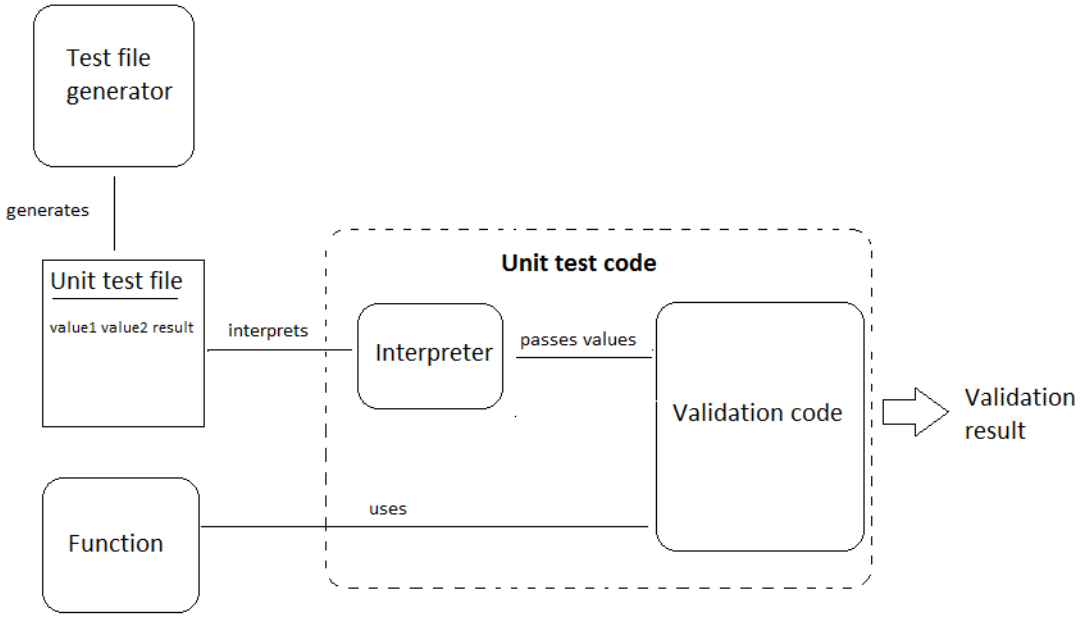

Figure 19 Testing framework components

Figure 19 describes how the sub-parts of the system interact. The test file generator creates the test file (specific for each function to be tested). The interpreter accesses the corresponding test file, parses the content and passes the parsed values to the unit testing implementation (validation code). The unit testing implementation uses the function to be tested by calling it with the corresponding values. After that, the testing framework validates the return value of the function according to the corresponding validation method.

The **test file generator** is responsible for generating the test-cases for a test and writing them to file. Test file generation is separated from the actual unit-test

framework. Generation and testing are two different automated processes that are not synchronized.

The test file generator would generate numbers, manipulate strings and write to files. This kind of task requires a high level programming language with scripting capabilities. The choice was made for Ruby for the reasons described in the "Tools used" section of chapter 3. For each operation a different class was developed. Other information was parameterized: the fixed point format of the operands and whether the operation was saturated or not. The location of the file on the file system and the title of the test were also passed as parameters. In order to automate the process of generating tests, a *run script* that incorporates the calling code for all the functions was developed.

The **test file interpreter** is a part of the unit test code. The interpreter mediates the communication between the validation code and the file containing the test cases. The unit test code is automated and run as a block, which means that at runtime the validation code calls the interpreter for a certain test file. The interpreter makes the details of the test file available (metadata and test-cases) via its methods to the validation code.

The reason behind splitting the interpreting part from validation code is the rule of separating concerns in software development. The validation code should only get some inputs, the expected output and a function to test and as a result should return a validation result. Adding the task of interpreting a file would have increased the complexity beyond a reasonable limit (considering, mainly, the overhead of handling files and strings in C). Another reason behind this design choice was the modularity it brings to the testing framework. A possible use case example would be that it is possible to rapidly mock test cases using the validation code without the need of creating a test file.

Table 8 exemplifies the methods that the interpreter implements

Method signature	Explanation
void init_file(const char* file_name)	Loads the file with the specified name and parses the file's metadata
Metadata* get_metadata()	Returns the metadata of the current file as a structure - Metadata

void* get_result(int test_no)	Returns the result of the test case determined by the *test_no* number
void* get_operand(int test_no, int operand_no)	Returns the *operand_no* input value of the *test_no* test case

Table 8 Interpreter – description of methods

The **validation code** is responsible for testing the function and returning the validation result according to the corresponding method. For this subpart of the system, an open-source testing library built for C was used. The library called **MinUnit** was developed by Jera Design and is described in a tech note [33]. The author of this library released it with a note that "the code can be used, with the understanding that it comes with no warranty" [33].

The library is extremely simple and yet efficient. It consists of only one header file (*minuit.h*) consisting of 2 #define macros and 1 external variable. It handles the validation of a boolean expression, considering the test as failed if the expression is false. This makes the library suitable for exact matching validation described in the Testing Approach section above.

In order to be usable in the unit test code part of the system, several improvements were made to the library. The first was adding an index to identify the test case to the output. The index used was the test case number as found in the test case file. The second improvement was the addition of a macro that implements the validation with an absolute error method. The new feature was built to resemble the existing one in order to keep the consistency of the product. Table 9 summarizes the methods that the modified library consists of along with a description of each item.

Macro	**Description**	**Part of original library**
mu_assert(message, test_case)	Tests if expression *test_case* is true or not; if not, then prints *message.*	Yes
mu_run_test(test_func)	Runs *test_func* function which contains one or more assert statements	Yes

mu_assert_line(message, line, test_case)	Same as mu_assert, just adds the line number to the output	No
mu_assert_line_with_error (message, line, error, max_error)	Same as mu_assert_line, just replaces boolean *test* with integer *error* and uses validation with absolute error	No
mu_test_title(test_title)	Prints *test_title*	No

Table 9 MinUnit description

The output of the unit testing library (the validation result) is printed on the console. If a test fails, either due to the result not being an exact match or the error lying outside the range, the testing process stops, printing the line that contains the failing test case in the test file. The test file name can be obtained from the test title which is printed at the beginning of each test. If a test case passes with an error (considering validation with an absolute error), the library prints the line of the test case and the value of the error without stopping the testing process. If all the test cases in the test file pass, the library prints a message confirming the fact that all the test cases passed. At the end of the run, the library prints the number of tests that were executed.

The unit testing framework described above is used to ensure that the libraries built have the expected behaviour. The other purpose of the testing framework is to provide data for analysing the performance of the libraries built. The information about tests passed and, especially, the information about the value of the error are a very good measure of the accuracy. The way this data is interpreted and the results of interpretation are presented in the next chapter.

6.3 Libraries overview

The final product consists of 3 distinct libraries:

- Basic library – FixedPoint.h
- Math library – FixedPointMath.h
- Debugging library – Info.h

Basic library (FixedPoint.h) contains type definitions (for all the fixed point formats) and basic arithmetic functions (add, subtract, multiply, fractional part, integer part). The main purpose of this library is to set the stage for implementing the algorithms and to provide simple functions for the fixed point formats. These functions have to output the exact value (errors are not acceptable).

Math library (FixedPointMath.h) contains the implementations of all the operations needed such as division, square root, logarithm, exponential, sine. Saturated versions of the algorithms are also built. Some of the operations have two versions implemented (with two different algorithms).

Debugging library (Info.h) contains the data structures and the methods needed to update and retrieve the debugging information about errors such as overflow or misrepresentation. The methods can be used in both the basic and the math libraries, and can be enabled or disabled at compile time by a flag. The debugging library brings an overhead and thus, in order to achieve efficiency, the use of the methods has to be disabled.

	Basic library	**Math library**	**Debugging library**
Data structures	-type definitions for 8.8, 16.16, 8.24, 24.8		-Structures for error and additional information
Methods	-addition and subtraction -multiply -integer part and fractional part functions	-division -square root -logarithm and exponential -sine	-methods for initialization of structures -methods for setting/getting/ resetting data structures

Table 10 Libraries description

Table 10 summarizes the content of each library described above. Further details about each function will be provided in the next sections of the report.

6.4 Basic library

The basic library defines the data structures for fixed point formats and implements simple arithmetic functions. It was developed separately from the math library containing algorithm implementations for two reasons. Firstly, the size of the resulting library (basic and library defined in the same header) would have been too large to maintain. Secondly, there are possible use cases that need only the basic library. In these cases, loading the math library code would have increased the size of the code without any benefits.

The data structures of the basic library define the following fixed-point formats:

- 8.8 – int8_8
- 16.16 – int16_16
- 8.24 – int8_24
- 24.8 – int24_8

A fixed point format named A.B, where A and B are integers, defines a format that uses A bits for the integer part and B bits for the fractional part. The name convention used for defining types is: **intA_B** for format **A.B**.

Two approaches were experimented with for defining types. The first approach was defining a *struct* with 2 member bit fields with corresponding sizes for integer and fractional part. The type definition would look like:

```
struct __int8_8
{
    unsigned char int_part :8;
    unsigned char fract_part :8;
}
typedef struct __int8_8 int8_8;
```

The main advantage of this way of defining types is that the fractional and the integer parts are easily accessible (accessing a member of a structure). The first version of the basic library was built using this representation style and several downsides were revealed along the way. One important problem that appeared was the lack of control on the actual sizes of the structures used. Even though the size of the bit field was specified (*:8* - in the example above), the size of the structure had to be a multiple of the size of int (4 bytes) (if not, padding would be added). This is inefficient, especially in the case of formats that should occupy a number of bits that is not divisible by 4 bytes (e.g. 8.8 format - should occupy just 2 bytes). Another issue with this approach was the fact that the interpretation of bit fields is compiler dependent [34] and brings overhead in processing at run-time [34].

The second approach had to be simple and efficient at runtime and had to have control over the size of the types defined. First, in order to achieve better control over the size, instead of using standard types such as *char, short, int and long,* types defined in <inttypes.h> were used, which include *int16_t, int32_t, int64_t*. The size of the standard types depends on the compiler, unlike the types defined in <inttypes.h> that have the same size independent of the compiler.

Defining new types consists of renaming the integer types, as in the example:

```
typedef  int16_t  int8_8;
```

In the case of 16.16, 8.24 and 24.8 formats, the new types were created by redefining the same basic type:

```
typedef  int32_t  int16_16;
typedef  int32_t  int8_24;
typedef  int32_t  int24_8;
```

This means that an integer (32 bits long) is split such that the first A bits describe the integer part while the last B bits describe the fractional part. Figure 20 describes how the memory is utilized in each type defined

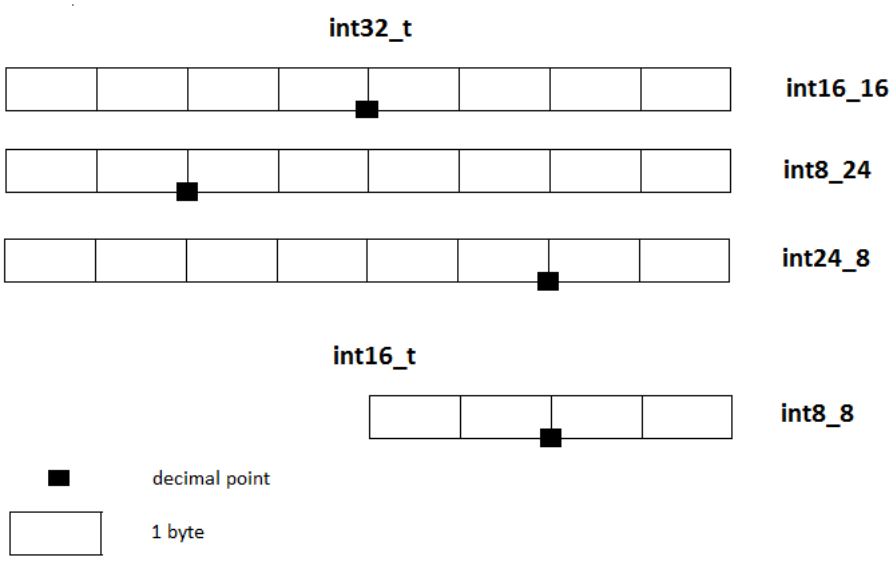

Figure 20 Type description

The difference between the new types that are represented by the same base type is made by the usage. For each type, there are specific functions defined for setting and retrieving integer and fractional parts. Also, specific operations are built for each type separately.

The advantage of this way of representing fixed point numbers is the simplicity that leads to efficient compiled code. The compiled code only uses integers and operations on it without having the overhead of handling a structure. This representation proved to be easier for developing the other functions during the implementation of the basic and the math library. Switching from the first way of representation was done when the basic library was finished. The transition required the modification of all the implemented methods. The suite of tests developed earlier helped ensure that the code maintained its behaviour.

Initialization of the fixed point numbers can be done in 2 different ways. One of them is to use functions (#define macros) to manually set the integer part or the fractional part. This means to know beforehand the bit representation of both parts. While for the integer part this is simple, for the fractional part this can be challenging. For example, considering the number 1.75 to be represented in 8.8 fixed-point format, the integer part would be 1 and the fractional part 0.75*256 = 192. Initialization for 1.75 would look like this:

```
int8_8 a;
set_int_part_8_8(a,1);
set_fract_part_8_8(a,192);
```

This way of developing is complex, mainly because of the need to compute the value of the fractional part. The functions for setting the integer part (*set_int_part_A_B*) and the fractional part (*set_fract_part_A_B*) are useful in other use cases.

In order to overcome the difficulties showed by the first approach, a macro for initialization was developed. It gets the value of the number to be represented as input and returns the fixed point format. The previous example can be modified to use the macro, as demonstrated below:

```
int8_8 a = new_int_8_8(1.75)
```

Using float numbers does not add an overhead as the macro performs a computation that is done at compile time (as long as the input is not a float variable). The code for the macro is:

```
#define new_int_8_8(X) ((int8_8) ((X)*256))
```

When X is replaced with a number, the computation will be done by the compiler and the result will be put instead of the macro. When X is replaced with a float variable the computation is done at run-time hugely increasing the computation time (because of the use of floating point computation).

The basic library implements simple functions to retrieve the integer part and fractional part of a number as fixed point numbers. Following are the functions that return the fixed point numbers:

```
#define floor8_8(X) ((int8_8)((X) & 0xFF00))
#define fract8_8(X) ((int8_8)((X) & 0x00FF))
```

To obtain the integer representation of the integer and fractional part, the functions: *get_int_part_A_B* and *get_fract_part_A_B* are developed.

For addition and subtraction operations the basic library has general functions (macros) that can be used by any type. The restriction is that the operands and result have to be of the same type. For example:

```
int8_8 a = new_int_8_8(0.3);
int8_8 b = new_int_8_8(0.4);
int8_8 c = add(a,b);
```

The code behind the add function is:

```
#define add(X,Y) ((X)+(Y))
```

In order to add 2 numbers of different fixed-point formats, a *cast* has to be performed. The cast can easily be done with a shift. The subtract function is defined in a similar way to addition.

Multiplication is also defined as a macro, but each type has its own definition, as detailed below:

```
#define mul8_8(X,Y)  ((int8_8) (((X)*(Y))>>8))
#define mul16_16(X,Y)  ((int16_16) (((X)*(Y))>>16))
#define mul8_24(X,Y)  ((int8_24) (((X)*(Y))>>24))
#define mul24_8(X,Y)  ((int24_8) (((X)*(Y))>>8))
```

The use of a multiplication macro is the same as the use of addition and subtraction.

Saturated versions for addition, subtraction and multiplication are implemented separately as functions because the corresponding saturated versions are too complex to be put in a #define macro statement. The saturated versions return the result of the operation if it is within the range, or one of the limits of the range if the result is outside the range. The naming convention used is: **s**{*name of operation*}{*format*}. For example, the following functions are implemented for a few saturated operations:

```
sadd8_8 – saturated addition for 8.8 format
smul16_16 – saturated multiplication for 16.16 format
ssub8_24 – saturated subtract for 8.24 format
sadd24_8 – saturated add for 24.8 format
```

Comparisons between variables of the same type can be done using the default mechanism for comparing integers in C (i.e. with operators <,>, ==).

An example of an addition between an 8.24 variable and a 16.16 variable is presented below:

```
int16_16 a = new_int_16_16(1.2);
int8_24 b = new_int_8_8(1.5);
int16_16 b_cast = b>>8;
int16_16 c = add(a, b>>8);
```

The basic library provides the tools for declaration and initialization of the fixed point numbers. It also defines basic functions for addition, subtraction and multiplication, together with corresponding saturated versions.

6.5 Math library

The math library (FixedPointMath.h) contains the implementation of the algorithms analyzed in the chapter "Algorithm assessment". Algorithms implemented also have a saturated variant where needed. As most of the functions in the basic library, the functions in the math library have operands and result of the same type. Consequently, a separate version of the function is built for each type (e.g. **div8_8**, **sqrt16_16**, **exp8_24** and **log24_8**). If the operation to be used takes two arguments (e.g. division) and the arguments are of different types, then a cast has to be done to the type desired for the result.

The saturated versions implement the same algorithm as the non-saturated versions, the only difference being the step before return. The algorithms implemented return the result interpreted as a 32.32 number. The last stage of the non-saturated version is to cast this result to the corresponding type. The step added to the saturated versions checks if the result interpreted as a 32.32 number is within the range of the type and, if not, it updates the value of the result with the corresponding limit of the range. The naming convention used for saturated versions is to add an **s** before the name of the function (e.g. **sdiv8_8**, **slog16_16**).

The algorithms described and analysed in the "Algorithms assessment" chapter are implemented in C code. Small changes were made to optimize the implementation to either use smaller lookup tables or to use a smaller number of instructions. Another kind of modification done during the implementation phase was to alter the parameters of the algorithm (number of terms used from expansions, number of iterations) so that the results meet the expectations.

All the algorithms were implemented using intermediary results interpreted as 32.32 numbers. This would normally require the use of a 64-bit integer, however, for the ARM architecture, using 64-bit integers has an overhead in terms of number of operations. This is due to the fact that registers in the ARM architecture have a size of 32 bits. An ARM instruction (umull [35]) handles multiplying two 32-bit integers and returns the result as a 64-bit integer. Such an instruction does not exist for multiplying

64-bit integers, and making use of such multiplications in C code would highly increase the number of ARM instructions. Taking this into consideration, during the implementation, 32-bit numbers were used to represent the integer and the fractional part of the intermediary results when such multiplications were needed. This optimization made the code harder to read than when using plain 64 bit numbers but the tradeoff paid off in terms of efficiency.

Every algorithm shares a common pre-computation step performed before starting the actual algorithm. One aim of this step is to ensure that the values of the input operands are in the correct ranges for the operation implemented. For example, for division, the denominator is checked not to be zero.

Some algorithms share a step where the input value is normalized to [0.5, 1).This is done by computing the position of the most significant bit with value 1. Two different functions were built: one for types that have a total size of 16 bits (8.8) – **norm16bits** and one for the types that have a total size of 32 bits (16.16,8.24,24.8) – **norm32bits**. Using the output of this function, each algorithm shifts by the correct amount needed to normalize so that the value of the first bit is 1. An optimized version of this function can be built for the ARM v5 architecture in order to take advantage of the CLZ instruction.

For division, two different algorithms were implemented: Newton-Raphson iteration and functional iteration. The pre-computation step is common for both implementations. In this stage the validity of the denominator (different from 0) is checked and the values of the inputs are transformed into positive numbers (if needed). The change is done by changing the sign of the number and computing the sign of the result. The rest of the algorithm will compute the absolute value of the result and a step is added before the return to change the sign of the result if needed.

In order to improve the algorithm, another step was added that increases the efficiency for a particular case. If the denominator is a power of 2, then the result is obtained by doing a corresponding shift on the numerator. This step is done after normalization to the [0.5, 1) interval. The code for **div8_8** is presented below where **m** represents the denominator, **n** represents the numerator and **sign** determines the sign of the result:

```
short k =norm16bits(m);
uint32_t  b = m << (32-k);
```

```
uint32_t result;
if (b == 0x80000000)
{   result = (k>16 ? n<<(k-17) : n>>(17-k));
    if (sign==-1) result = ~result;
   return result;
}
```

In the code example presented, variable **b** holds the normalized value of the denominator. The value is interpreted as a 0.32 number.

The next step in the algorithm is to implement the iteration. This is the main part of the algorithm and this is also where the Newton-Raphson implementation and the functional iteration implementation differ. As an example the Newton-Raphson algorithm is described next. The recurrence formula and the starting value of the algorithm are presented below (see chapter "Algorithms Assessment" for more details):

$$x_0 = -\frac{32}{17} * b^* + \frac{48}{17}$$

$$x_{i+1} = x_i * (2 - b * x_i)$$

In the formulas above **b*** is the normalized value of the input. The formula for the starting value can be modified in order to be computed more easily. For instance, 32/17 can be approximated to 2 and 48/17 can be approximated to 3. This means that the formula for the x_0 would be:

$$x_0 = -2 * b^* + 3$$

One important observation that can be made is that the integer part of x_i is always 1. Given that b* is in [0.5, 1) means that x_0 is in (1, 2]. Considering that the case when b* = 0.5 (and x_0 = 2) is treated separately (when b* = 0.5 implies that the input value is a power of 2), we can conclude that x_0 is in (1, 2). The series x_i converges towards 1/b*, which is in (1, 2) and thus implying that x_i has the integer part equal to 1 for every i. This observation hugely improves the efficiency of the implementation as only the fractional part should be known during the iteration.

The C code for computing the starting value is:

```
uint32_t   x = -(b<<1);
```

Variable **x** will hold the fractional part of the approximated value at each step and is interpreted as a 0.32 number.

The following piece of code shows how an iteration of the Newton-Raphson algorithm is implemented in C:

```
uint64_t  p,q;
p = ((int64_t) x)<<1;
q = ((int64_t) b*x ) >>32;
q = (int64_t)((int32_t) q)*x;
x = (q>>32) + (((int64_t) b*x ) >> 31) + b;
```

p and **q** are variables that hold intermediary values in computation. These values are interpreted as 32.32 numbers. The result of multiplying 2 integers interpreted as an A.B number (A – bits for the integer part and B – bits for the fractional part) should be interpreted as a C.D number, where C = 2*A and D = 2*B. For the piece of code above, **b** and **x** are interpreted as 0.32 numbers, so the result will of 0.64 format. Variable **q** is interpreted as 32.32, which means that a shift to right of 64-32 =32 bits is needed (as it can be noted from the code). At the end of all the iterations, the variable **x** will hold the fractional part of the inverse.

The next step is to compute the product between the inverse of the denominator and the numerator. The C code for this is:

```
uint32_t  r = (((int64_t) x*n ) >> 32 ) + n;
```

Variable **r** is interpreted as a 24.8 number.

The next step is to scale the result back from the normalized value. The piece of code for doing that is:

```
r  = ( k>8 ? r>>(k-8): r<<(8-k) );
```

For the saturated versions, a check is done to verify if **r** is outside the range of the format (8.8 in this case). Considering that **r** represents a positive number, it should be verified if it is larger than 128*256 (7 bits for the integer part and 8 for the fractional part). The code for that is:

```
if  (r & 0xFFFF8000 )  r = 0x7FFF;
```

The last step before return is to actualize the sign of the result:

```
if  (sign == -1)  r = ~r;
```

When returned, the value of **r** should be cast to the corresponding type, int8_8 in this case:

```
return (int8_8) r;
```

The implementation of the functional iteration algorithm is very similar to the implementation for the Newton-Raphson method. The main difference is how the

iteration step is implemented. The rest of the steps, namely multiplying with numerator, reconstructing the result, verifying the sign are all the same.

For the other types (16.16, 8.24, 24.8) the algorithms are identical, the difference being at the normalization step and at the step for reconstructing the result. For saturated versions, the step that verifies whether the result is in the range is also different.

For each of the other operations implemented (square root, logarithm, exponential, sine) only a brief description will be provided for reasons of space and readability of the dissertation. The details presented will focus on the changes made to the original algorithms to make it more efficient in the C implementation. A code example for each operation is presented in the Appendix.

The algorithm for square root computation is similar to the Newton-Rapshon variant for division. The reason is that the algorithm is based on the Newton-Raphson iteration. The algorithm is almost identical for all types, the only difference being in normalization and constructing the result stages. A key difference from the division implementation is that there is no saturated version. All the results are positive and smaller than the inputs therefore they cannot get out of the range.

In the pre-computation step, it is verified that the input is positive and then it is normalized. The next step is to compute the starting value x_0. For this purpose, two constants were defined: **SQRT_C1** and **SQRT_C2**. The formula is: x_0 = SQRT_C1 – SQRT_C2*a; where **a** is the normalized value to [0.5, 1) interval. The values of the constants are determined by the formula:

$$\frac{1}{\sqrt{a}} \cong x_0 = 1.78773 - 0.80999 * a$$

For a small set of values (**a** in vicinity of 1) $\mathbf{x_0}$ is lower than 1. For those values, 1 is a better approximation of the inverse of the square root, thus, we can start from $\mathbf{x_0} = 1$. This means that at each step the integer part of the approximated value is 1. As for the division algorithm, only the fractional part is computed at each step. In order to reduce the amount of space used, the constants were stored as 32 bits variables interpreted as 0.32 numbers. In the case of **SQRT_C1,** only the fractional part was stored.

A supplementary step is needed before returning. If in normalization to [0.5, 1), the power of 2 used is odd, then a multiplication by the root square of 2 is needed. In order

to increase the efficiency of the implementation, only the fractional part of the square root of 2 was stored as a 32-bit integer interpreted as a 0.32 number.

The algorithm for logarithm differs greatly from the algorithms for division and square root. This is because it implements the Taylor series approximation and makes use of lookup tables. The implementation uses two 16-entries lookup tables. One table is for the constants $\mathbf{A_i}$ and the other is for $\mathbf{log(A_i)}$ (see the corresponding section in the "Algorithm assessment" chapter). Similar to the square root algorithm, a saturated version is not needed for the logarithm algorithm. The values of the result lies between $\log(2^{-32}) = -22.18$ and $\log(2^{32}) = 22.18$, which does not exceed any range.

The first step in the algorithm is to check if the input is positive and to normalize the value of the input to the [0.5, 1) interval. The second step is to find the number of the entry in the lookup tables corresponding to the input value. The variable **key** holds the number of the entry (**x** is the normalized value 0.32 format):

```
unit16_t  key = (x>>27) – 16;
```

The next step is to compute the corresponding value of the input in the small interval around 1, by multiplying **x** with the value of the constants in the lookup table for the entry **key**. All the entries in the lookup table have the integer part equal to 1, therefore, only the fractional part was stored. The next step in the algorithm is to compute the Taylor approximation for the corresponding value of the input. The last step before the returning the result is computing the final result by subtracting the logarithm of the constant used earlier (stored in the second lookup table) and adding a multiple of log(2) (stored as a constant). The code for 8.8 format that implements this step is:

```
app = app  -  __LNA[key];
app = app + (k-8)*LN_2;
```

__LNA is the lookup table for the logarithm of constant; **k** is the output of **norm16bits** functions and **LN_2** is the constant storing log(2).

The implementation for 8.24 format was enhanced so that it would provide good error patterns. The initial algorithm (same as for the other formats) would produce the results with large errors (errors of 6-7 bits were noticed). To get better results, an additional step was implemented. Instead of using the Taylor series expansion around 1 for numbers in the small interval, the interval was split in 8 equal intervals and the Taylor expansion for the middle of each interval was used. The algorithm is more

efficient as the interval on which the approximation is made is much smaller. In order to achieve this, two additional lookup tables of eight entries each are needed.

The algorithm implemented for the exponential operation resembles the one used for the logarithm. The first step represents the pre-computation described by the algorithm (see "Algorithm assessment" chapter). The second step is to compute the position of the corresponding entry in lookup tables and subtract the constant stored in the table. The third step is to compute the Taylor series approximation and the last step is to construct the result.

This algorithm gives good error results for 8.8 format but not for other formats. To achieve better results, a variation of the algorithm was implemented. A step similar to the one added to the logarithm algorithm was introduced. The small interval around 0 was split in 8 equal intervals. The Taylor series expansion around the middle of the interval where the corresponding value lays was used. This reduces the difference between the value where the function should be approximated and the value around which the Taylor series is built by 8, increasing the error performance.

The algorithm that implements the sine function is different from the other two types of algorithms presented above. It uses power series approximation and takes advantage of the various formulae in trigonometry. The description of the algorithm implemented can be found in the "Algorithm assessment" chapter. Computing the cosine can be achieved by only modifying the last step of the algorithm, and therefore, the performance and efficiency should be similar.

The first step of the algorithm is to reduce the input to the [0, $\pi/2$) interval. To achieve that, the input **a** should be written as: **a = p*(π/2) + q;** where **p** and **q** are integers (**q** being positive). This is done by multiplying the input by the inverse of $\pi/2$ and separating the integer part **p** and the fractional part **q'**. The only information needed about **p** is its result of **p % 4**, which indicates in what quarter of the unit circle does the input value lie. In order to obtain **q** from **q'**, the following equality is used: **q = q' * (π/2);**

The next steps of the algorithm handle computing **sin(q)** and **cos(q)**. The first step is to compute **q/8.** In the next step is computing the powers of **(q/8)** needed (2 to 5). This is followed by using the power series expansions described by the algorithm to compute **sin(q/8)** and **cos(q/8)**. The fourth step involves computing **sin(q)** and **cos(q)** using trigonometric equalities.

The last step of the algorithm is to construct **sin(a)** from **sin(q)** and **cos(q).** In order to output the cosine only this step of the implementation should be changed using the trigonometric equalities corresponding to the cosine.

The implementations of the algorithms share some common characteristics that were noted during the description (use of 32-bit accuracy intermediary results, normalization to [0.5, 1). This ensures that the library is consistent and new algorithms can be built by taking the code already written as an example. Some modifications were made to the algorithm presented in the "Algorithm assessment" chapter. The main reason for that was the need for optimizing the code in terms of efficiency or accuracy.

6.6 Debugging library

The purpose of the debugging library is to provide storage and accessing information about errors that appear during computation. Example of errors to be considered: overflow, division by zero, logarithm of negative number or square of negative number or underrepresentation. While the other types of errors are self-explanatory, underrepresentation refers to the state where a non-zero number is represented as a zero (0). This happens when the number is between zero and the smallest positive number that can be represented. These errors do not stop the run of the program; they should be recorded and accessible.

The implementation of the library was not finished due to lack of time. The data structures for storing the latest error along with additional information and the methods to initialize the use of the library were implemented. Support for one type of error means implementing corresponding functions for setting/getting/resetting the value and a function for checking an input for the error. Currently, these methods are implemented just for overflow. Developing support for new types of errors or debugging information is easily done taking the functions built for overflow as model.

The structure for storing the error has an integer field that should be interpreted as an array of bits. Each bit corresponds to one type of error. If the corresponding bit is 1 then this implies that the error has occurred, otherwise, if the bit is 0 then the corresponding error has not occurred. A variable (**CURRENT_ERR**) is declared to hold the last state of the error. The structure is defined as illustrated below:

```
struct __error
{
    int code;
```

```
}
typedef struct __error _Error;
```

For each error a mask is defined, which is an integer that has a bit representation containing only 0s, except on the position corresponding to the type of error, where the bit is 1. This mask is used in the set/get/reset macros defined for each error. The code example for overflow is provided below:

```
#define MASK_OVERFLOW 0x00000001

#define set_overflow(X) ((X)->code |= MASK_OVERFLOW)
#define reset_overflow(X) ((X)->code &= ~MASK_OVERFLOW)
#define get_overflow(X) ((X)->code & MASK_OVERFLOW)
```

A function to verify if an input is within the range of its type is needed. Two different functions were built: one for 8.8 type – **verify_overflow_16_bits** and one for 16.16, 8.24, 24.8 types – **verify_overflow_32_bits**. The signatures of the functions are:

```
void verify_overflow_16_bits( int64_t x);
void verify_overflow_32_bits( int32_t x);
```

The value held by the variable **x** is compared to the limits of the corresponding range. Because 16.16, 8.24, 24.8 formats have the same number of bits in total (32), the range of **x** is the same: between $2^{31} - 1$ and -2^{31}.

The use of debugging library is triggered by defining a flag denoted as **DEBUG_FLAG**. If it is defined, the code corresponding to the debugging library is added to the functions in the math and basic library at compile time. Conditional compiling is done using the #**ifdef** preprocessor directive. An example of the use of this directive is described below - from **div8_8** function:

```
#ifdef DEBUG_FLAG
  verify_overflow_16_bits(out);
#endif
```

As mentioned earlier, the implementation of the debugging library is not fully finished. It can be easily extended as the data structures are defined and the overflow implementation can be used as a guideline.

6.7 Summary

This chapter described the implementation effort done in this project. In order to keep a good flow of the dissertation, adding pieces of code was avoided as much as possible.

A description of the testing framework was added for two main reasons. First of them is that a custom system was built in order to achieve the requirements needed for it. The second reason is that the testing framework plays an important role in the project. It validates the code written and provides the information about the accuracy of the implemented algorithms.

The next chapter will present the results of the evaluation of the libraries. References to performance were made in this chapter to argue the need of some optimization but a detailed description is presented in the "Evaluation" chapter.

Chapter 7 Evaluation

In this chapter an evaluation of the libraries developed as a part of this project is presented. It will focus on the initial requirements and the degree to which the implementation fulfils them. This chapter will reflect the results of the efforts described in "Algorithm assessment" and "Implementation" chapters.

7.1 Evaluation criteria

The success of the project can be measured from different points of view. The pragmatic aim of the project is to deliver a set of libraries, the use of which improves the way fixed point computations are performed for SpiNNaker. One measure of success would be the rate of adoption among the developers who are contributing to SpiNNaker.

In this chapter the focus will be on the measurable aspects of the results. From the initial requirements, three different axes can be distinguished as criteria for the evaluation of the project:

- **Software usability**
- **Accuracy**
- **Efficiency**

Software usability refers to how easy or natural a piece of software is to learn and use. In the current context, software usability refers to how easy to learn and to use the libraries have been without encountering errors.

Accuracy refers to how well the operations implemented approximates the intended value. In this area exact measurements were done, which will be presented later in this chapter. Preliminary results about the accuracy of the algorithms implemented were detailed in the "Algorithm assessment" chapter. The results obtained after the implementation are compared with the expected ones and the differences discovered will be explained. The main resource of the data about accuracy represents the validation data obtained when running the unit tests.

Efficiency refers to how many machine operations are needed to compute the desired result. The way efficiency is measured will be defined in the section dedicated for it. As mentioned about the accuracy, preliminary results for efficiency are described in the "Algorithm assessment" chapter. Measurement of the efficiency was mainly done on an ARM emulator. Small changes were added to the code to optimize it for the ARM

architecture. At the end of optimizations, the test suite was run again to ensure that the code has the same behaviour.

In the following sections of the chapter each of the evaluation axes will be treated separately. The methods of assessment will be described and then the results will be showed. At the end of each section, conclusions about how well the libraries performed in that area will be discussed.

7.2 Software usability

Software usability of a certain product refers to the interaction with it. Due to the lack of time and resources, a proficient analysis and measurement of the usability of the libraries could not be performed. Instead, a brief analysis was conducted and the results are presented.

Jakob Nielsen gives a definition of the basic principles of usability in a famous post [36]. Even though, in the post Nielsen is discussing mainly about user-facing interface usability, its definition is valid for a general software product.

Nielsen states in [36] that "usability is a quality attribute that assesses how easy user-interfaces are to use". In the current context, the user is the developer trying to make use of the libraries built, while the user-interface represents the method signatures and data-structures accessible to him. In the post, the author identifies 5 quality attributes [36]:

- Learnability
- Efficiency
- Memorability
- Errors
- Satisfaction

A brief presentation of the interface of the libraries is done while referring to the 5 qualities enumerated above.

One of the tasks the user can do while using the library is **declaring a variable of a fixed-point type**. The syntax for that is simple and straightforward:

```
intA_B x;
```

Where **A_B** stands for the one of the formats: 8_8, 16_16, 8_24, 24_8. This task is easy to learn and, because of its simplicity, is error free.

Another task a user can do is **initializing a variable of a fixed-point type**. The correct way of using the libraries to achieve that is the following (an example):

```
intA_B  x = new_int_A_B(1.4);
```

The syntax is simple and easy to learn, especially because of the use of the word *new* in the name of the macro. It resembles the way a constructor for an object is called in higher level languages. While the use of this method is easy to learn, a mistake that can appear could be one the following:

```
intA_B  x = 1.4;
intA_B x = 1;
```

The user tries to initialize the value of x with the fixed point representation of 1.4, but only succeeds to assign the value 1 to the integer x (which is the representation of the smallest positive number that can be represented with that format). The main issue is that the above code compiles without triggering any warnings. We can conclude that while the task of initializing a variable of a fixed-point type is easy to learn and use, some cases of misuse exist.

Obtaining the integer part of one number is another task the user can require. The macro built for it is floorA_B and the use is:

```
intA_B  f = floorA_B(x);
```

The macro has a self-descriptive name (as all the function names in the libraries) therefore easy to remember. The signature of the macro is also straightforward. The only possible confusion would be with the macro **get_int_part_A_B** which returns the integer representation of the integer part, the correct use is:

```
int  n =  get_int_part_A_B(x);
```

If **A_B = 8_8** and **x = new_int_8_8(1.5)** then **floor8_8(x) = 0x0100 = 256** while **get_int_part_8_8(x) = 1** . This shows that the two functions have, as intended, different meanings. A user can use one instead of the other by mistake, and the compiler would not flag any warning leading to a source of mistakes that is hard to detect.

Using the basic arithmetic (add, subtract) is as simple and as natural as for other basic types (int, float). There are two ways of doing the operations. One is by using the native operators (+,-) or by using the general macros provided. The only condition is for the operands to be of the same type. This approach is simple and easy to learn.

Using the functions defined for the operations (division, square root, logarithm, exponential) is also simple and straightforward. The names of the functions are self describing: *nameA_B* ; where name stands for the corresponding name of the operation while A_B stands for the corresponding format. Table 11 shows the connection between the name of the operation and the name of the function. As it can be seen, most of the names are borrowed from **math.h** library

Operation	Function name
Addition	add
Subtract	sub
Multiply	mul
Division	div
Square root	sqrt
Logarithm	log
Exponential	exp
Sine	sin

Table 11 Names used for functions

One potential pitfall that can appear is that the user will try to send two arguments of different fixed-point formats to one function. One example would be to use the **mul8_8** function for an 8.8 number and a 16.16 number. At compile time no warning flag will be triggered, while at run-time the function will treat the 16.16 as a 8.8 number (more precisely, the last 16 digits of the 16.16 number). This behaviour is not desirable, and therefore leads to errors.

It can be concluded that overall the libraries are easy to use and learn. However, there are some areas where pitfalls can appear. The main disadvantage is that these pitfalls are not flagged at compile time, leading to potential bugs at run-time. These potential errors of usage would be avoided after the user became familiar with the libraries.

7.3 Accuracy

In the article mentioned earlier [36], the author contrasts usability with utility. While usability refers to the interface of the product, utility refers to its functionality. In the

current context, utility is a function of accuracy and efficiency (number of instructions executed). For an algorithm accuracy and efficiency are related, affecting each other. Usually, an increase accuracy leads to a decrease efficiency (more instructions are needed). This is the reason why between accuracy and efficiency a tradeoff has to take place.

The information about the accuracy was obtained from the validation results of the unit-testing framework. The program was run on the development machine: standard PC with x86 architecture. From the behaviour point of view, compiling for x86, ARM or any other compatible architecture (that has the instructions needed) is of no importance. The C programming language portability ensures that the algorithm compiled run on an x86 PC has the same results as the same algorithm run on an ARM device. Running the unit-testing framework on the development machine had the main advantage of not needing to port testing framework to an ARM device.

The validation results contain the value of the error (the expected result minus the actual result). When computing the error, both the expected result and the actual result are interpreted as integers. The errors are interpreted in number of bits. For example an error of -1 or 1 is an error of 1 bit, while an error of -2, -3 , 2 or 3 is an error of 2 bits. The interpretation helps better understand how well the result approximates the value. An error of x bits means that the result approximates the expected value to the last x bits.

For each operation the structure of the test file is different but all share some characteristics. One characteristic is that the first 1000 test cases are randomly chosen. Another characteristic is that the for each edge case situation 100 test cases are generated. When presenting the results, the edge situations are described.

For the division 4 edge cases were identified and implemented in the test generator and are part of the test file structure. Naming **n** the numerator, **m** denominator and **max** the maximum absolute value that can be represented in the format, the description of the edge cases is:

- **n** positive large (between **max/2** and **max**) and **m** around 0 (between -1 and 1) – test case **A**
- **n** negative large (between **-max/2** and **-max**) and **m** around 0 (between -1 and 1) – test case **B**
- **n** around 0 (between -1 and 1) and **m** positive large (between **max/2** and **max**) – test case **C**

- **n** around 0 (between -1 and 1) and **m** negative large (between **– max/2** and **max**) – test case **D**

The Table 12 summarizes the results for functional iteration algorithm random cases. The values are out of a total of 1000 test cases.

Function	Accurate	1 bit errors	2 bits errors	>3 bits errors
div8_8	995	3	1	1
div8_8_v2	996	2	1	1
div16_16	1000	0	0	0
div16_16_v2	1000	0	0	0
div8_24	962	35	2	1
div8_24_v2	990	10	0	0
div24_8	1000	0	0	0
div24_8_v2	1000	0	0	0

Table 12 Division evaluation results - Random case

Table 12 shows that for the general case the division functions perform quite well: over 99% of the cases are accurate (with the exception of div8_24). Also the Newton-Rapshon iteration method (v2) seems to produce slightly better errors. The saturated versions have virtually the same kind of error rate which is explained by the fact the added step does not disrupt the algorithm.

For C and D edge cases (defined above) the error rate is virtually 0%. For A and B edge cases the error pattern of the non-saturated versions is very bad. The number of accurate test cases is <50% and errors of 6 or 7 bits appear. The reason is that almost all the test cases have as result a huge number that overpasses the range of the fixed point format. In this case the non-saturated versions will return the last bytes of the result. The saturated versions have almost 100% accuracy. The reason is that if the result is out of the range the corresponding is returned.

For square root 2 edge cases were analysed and their results were computed- where **a** is the input:

- **a** is large (between **max/2** and **max**) – edge case **A**

- **a** is small (between 0 and 2) –edge case **B**

The Table 13 is showing the errors for random case (1000 entries)

Function	Accurate	1 bit errors	2 bits errors	>3 bits errors
sqrt8_8	999	1	0	0
sqrt16_16	994	6	0	0
sqrt8_24	927	73	0	0
sqrt24_8	999	1	0	0

Table 13 Square root evaluation results - Random case

For edge case **B** all the functions are 100% accurate, while for edge case **A** all the functions except 8.24 have 100% accuracy while 8.24 has 93% accurate results and 7% 1 bit errors. These results show that the algorithms implemented for square root are very accurate. Only for the 8_24 format the number of accurate outputs is less than 99% , but the value is good ~93%. It can be concluded that the square root functions are reliable.

For logarithm the same 2 edge cases as for square root were implemented. The Table 14 shows the error for the random case (100 entries)

Function	Accurate	1 bit errors	2 bits errors	>3 bits errors
log8_8	1000	0	0	0
log16_16	885	115	0	0
log16_16_v2	999	1	0	0
log8_24_v2	925	75	0	0
log24_8	998	2	0	0

Table 14 Logarithm evaluation results - Random case

v2 stands for the optimized version of the algorithm. The error pattern for log8_24 is very bad: less than 20% accurate and error as large as 7 bits. The results obtained for the edge cases reflect the values obtained for the random case. It can be concluded that the logarithm implementation was a good accuracy.

For exponential no edge cases were defined. The random test case was constrained to be smaller than 32 so the result is not very large. Table 15 shows the results of the evaluation.

Function	Accurate	1 bit errors	2 bits errors	>3 bits errors
exp8_8	999	1	0	0
exp16_16_v2	988	12	0	0
exp8_24_v2	917	39	34	10
exp24_8_v2	875	116	9	0

Table 15 Exponential evaluation results

Functions exp16_16, exp8_24 and exp24_8 have very bad error results. Functions with names ending in "v2" are optimized versions of the originals. Analysing the results we can conclude that although exponential performs worse than the previous operation, the error pattern is good enough to make the exponential usable.

For sine no extra edge cases were analysed. The results of the evaluation are presented in Table 16.

Function	Accurate	1 bit errors	2 bits errors	>3 bits errors
sin8_8	999	1	0	0

Table 16 Sin evaluation results

Looking at the table it can be concluded that the sine computation is very accurate.

Analysing the results of all the operations some conclusions can be drawn. One is that operations for 8.8, 16.16, 24.8 have good or very good error patterns (>95 % accuracy) while the operations for 8.8 have an accuracy between 87% and 95% (which are acceptable).

7.4 Efficiency

Efficiency represents the number of machine instructions used during one algorithm. This number has an impact on the time needed for computation and on the energy consumed. The aim for the libraries is to compute the results using as few machine instructions as possible.

Evaluating the number of instructions can be done in several ways. One way to approximate is to count the multiply, add, subtract and branch instruction found the pseudo-code representation. Such estimation can only give a rough idea about the efficiency of the algorithm as the actual implementation can modify the instructions. In the "Algorithms assessment" chapter such estimations were presented. Another way of approximating the number of machine instructions used is to analyse the C code implemented. By using this kind of estimation a better result will be available but the approximation is still rough as the ARM compiler can produce a variable number of instructions. The most accurate way of evaluating the efficiency is to count the ARM instructions needed for the algorithm when it is run on an ARM device.

In order to evaluate the efficiency, a code sample using one function of the library at a time was compiled for ARM and executable file was loaded to Komodo - an ARM emulator and debugger. The compile step was performed using *-01* optimization flag. For the ARM emulator the latest available ARM architecture in Komodo was used: ARMv4. This architecture is older than the one used in SpiNNaker and the one for which the code was aimed (ARMv5).

One difference between these architectures consists in the content of the instruction set. ARMv5 adds a couple of new instructions to the previous set. The functions were compiled for ARMv5 and could be run on the emulator. This was possible because of the backwards compatibility and because no new instructions were generated. The only exception was the sine function, which in its assembly code made use of an instruction introduced in the ARMv5 (smlabb) set. In this case the code was compiled for ARMv4 architecture. One important note to make is that the CLZ instruction (which is needed to highly improve the efficiency of normalization functions) is introduced in the ARMv5 instruction set. This means that the instruction could not be used in the emulator. In order to not to prevent the measurement the CLZ instruction was replaced with NOP (no operation). This was possible because at this step only a measurement of the efficiency was wanted, not one of accuracy.

One thing to note is that norm functions from the math library (**norm16bits** and **norm32bits**) were optimized by changing the assembly code. The aim was to use the CLZ instruction reducing the impact on number of arm instructions from 20 to 2. Because the emulator lacked the possibility to run ARMv5 code, CLZ instruction was replaced as described above.

The method of measuring the efficiency of a function was to count the instructions performed by the processor between its call and its return. Komodo shows the total count of instruction performed from the start of the program. This number is obtained by observing the count at function's call and at its return.

For each operation, its default C implementation from *math.h* was measured in terms of number of ARM instruction. Also, all the functions in the math library were measured in the same way. In order to compare the results and analyse the results, the same set of inputs was used for all the functions associated with an operation. The values of the inputs were chosen randomly within the range of the format types used.

The code sample used for exponential function is shown below:

```
#include <math.h>;
int main
{
    float res = exp(5.28);
    return 0;
}
```

The code sample used for the fixed-point version of the exponential is described below:

```
#include <FixedPointMath.h>;
int main
{
    int8_8 res = exp8_8(new_int_8_8(5.28));
    return 0;
}
```

For exponential six functions were compared: exp, exp8_8, exp16_16, exp16_16_v2, exp8_24_v2 and exp24_8_v2. Table 17 summarizes the output of the efficiency measurement.

Function	**Input 1**	**Input 2**	**Input 3**	**Input 4**	**Input 5**	**Average**
Exp	1308	1326	947	1304	1322	1241.4
Exp8_8	97	98	97	97	97	97.2
Exp16_16	106	106	106	106	106	106
Exp16_16_v2	141	144	144	144	144	143.6
Exp8_24_v2	141	144	144	144	144	143.6

Exp24_8_v2	141	144	144	144	144	143.6

Table 17 Efficiency results – exponential

The results show that the algorithms implemented for exponential are between 9 to 12 times more efficient than the basic implementation from *nath.h*. The improved version is 35% more costly in terms of ARM instructions (for 16.16). Algorithms for the formats using 32 bits in size are less efficient. The difference is not significant, around 10%.

Table 18 describes the results obtained for logarithm.

Function	**Input 1**	**Input 2**	**Input 3**	**Input 4**	**Input 5**	**Average**
Log	1555	1259	1461	1425	1433	1426.6
Log8_8	75	75	75	75	75	75
Log16_16	73	73	73	73	73	73
Log16_16_v2	107	107	107	107	107	107
Log8_24_v2	107	107	107	107	107	107
Log24_8_v2	107	107	107	107	107	107

Table 18 Efficiency evalutation – logarithm

The results show that on average the algorithms implemented in the math library for logarithm are 13 to 19 times more efficient than the default implementation from *math.h*. The improved variant ("v2") is 46% more costly than the first form (as seen for 16.16 format). The number of operations is within acceptable values (around 100 instructions).

Function	**Input 1**	**Input 2**	**Input 3**	**Input 4**	**Input 5**	**Average**
Sqrt	560	565	540	562	545	544.4
Sqrt8_8	196	208	208	196	208	203.2
Sqrt16_16	196	208	208	196	208	203.2
Sqrt8_24	231	232	244	233	244	236.8
Sqrt24_8	194	206	206	194	206	201.2

Table 19 Efficiency evaluation - square root

Table 19 describes the evaluation results for square root operation. The evaluation shows that the functions implemented are between 130% and 170% more efficient than the default square root implementation. The variant for 8.24 format is more expensive in computations because it needs one more iteration in the algorithm to achieve good accuracy results. An iteration in the square root algorithm is equivalent with 33 operations which accounts for the difference between the values for sqrt8_24 and the rest.

For sinus operation only the 8.8 format was implemented and evaluated. Table 20 shows the measurements of efficiency.

Function	Input 1	Input 2	Input 3	Input 4	Input 5	Average
Sin	761	1332	1387	1415	1322	1243.4
Sin8_8	275	278	275	281	281	278

Table 20 Efficiency evaluation - sinus

The results show that the implementation of sinus operation is 347% more efficient than the default implementation of in the math.h library. The number of computations required for the fixed-point implementation is higher than the value obtained for the other functions.

The implementations for logarithm, exponential, square root and sinus proved to bring a big improvement in terms of number of operations, comparing to default *math.h* library. An observation is that the number of operations needed does not differ too much from input to input. In the default implementation, the values of the inputs have an impact on the efficiency, which is not true for the implementations in fixed point.

Table 21 summarizes the results of the evaluation for division.

Function	Input 1	Input 2	Input 3	Input 4	Input 5	Average
Div	55	55	55	55	55	55
Div8_8	114	114	112	114	114	113.6
Div16_16	155	155	155	155	155	155
Div8_24	155	155	155	155	155	155
Div24_8	155	155	155	155	155	155

Table 21 Efficiency evaluation – division

The measurements show that the use of fixed-point implementation of the division is less efficient that using the default implementation from *math.h*. The float default implementation is 105% to 181% more efficient. This shows that the division operation needs further optimization so it can improve the figures for number of instructions needed. The values shown for the float implementation do not include the amount of computation of casting integer to float or float to integer. These operations would be required when numbers are used as integers.

7.5 Summary

In this chapter a brief evaluation of the software product developed was presented. Three different areas were tackled to provide a well-rounded view of the libraries performance. Firstly, it has been shown that the libraries are easy to learn and use. Some possible misuse cases have been highlighted but those are easily avoidable after the user gets used to the libraries. Secondly, the accuracy of the operations was measured and presented. The values obtained show that the errors are of at most 1 bit in size with a few outliers of larger size. The large percentage of accuracy (mostly over 95% and at least 87%) shows that the libraries can be used reliably. Thirdly, the efficiency was evaluated by counting the ARM instructions needed for each operation. The results for all the operations except division showed a larger increase in efficiency. For the division the values show that the default implementation is more efficient than the ones developed.

Chapter 8 Conclusions and future work

This dissertation had two main purposes. One of them was to provide a theoretical basis for fixed-point computations for energy efficient devices. The second purpose was to build a working piece of software that would solve the real number computation problem the SpiNNaker developement encountered. Each of the aims was at least partly achieved.

The first part of the dissertation aimed to describe the mathematical background and to provide an evaluation of the algorithms and methods to be used. Valuable information was obtained by reading the existing literature and by implementing versions of algorithms to check their performance. Implementing particular algorithms was beneficial as it provided results about how good one method (or algorithm) is in the current context. The context is determined by elements such as requirements on error values, available operations and available memory size. The first part produced a valuable amount of information used in the implementation part. During the development phase some inconsistencies were observed. One of them was that the requirement for accuracy was set at 16 bits while a type defined had 24 bits (8.24). This meant that most of the algorithms had to be further optimized to reach 24 bits accuracy during the development phase.

The purpose of the second part of the dissertation was to build a working version of libraries to be used by SpiNNaker developers. The piece of software built had to implement three broad features: defining fixed-point formats, implementing operations and developing a method of gathering debug and error information. For each of these features a different C library was built. This was beneficial as the design proved to be modular and adapt efficiently to possible use-cases.

The basic library aimed to provide the basic functions and data structures needed for the use of the fixed-point types. This objective was partially fulfilled. For example, only four fixed-point formats were implemented while more could have been useful. From the computation efficiency point of view, most of the functions defined in the library are efficient. The saturated version of addition, subtraction and multiplication bring an overhead as they are defined as functions (that are not inline) and add a few instructions to their non-saturated variants. A pitfall in the development process was a wrong first approach to defining types by using structures. The downsides of this approach become obvious only late in the implementation process causing a waste of time as the code had

to be re-written. The new approach was beneficial as it simplified the structure and the use of the fixed-point types making them almost as natural and efficient to use as default data-types.

The math library has as main objective implementation of efficient algorithms for the set of operations wanted. For logarithm, exponential, square root and sinus the three areas of evaluation (usability, accuracy and efficiency) showed good and very good results. These results can be interpreted as a validation of the use for these functions. Division's implementation failed to prove itself more efficient than the default float implementation. An argument for the fixed-point division would be that using float division implies using float representation for numbers. This is not desirable, as every other operation on floats is many times less efficient than its fixed-point correspondent. A pitfall of the math library implementation is the size of the code and of the memory used. While an exact measure was not performed, an informal assessment in terms of lines of code and size of lookup table shows values that can represent a problem for the SpiNNaker memory system. The implementation phase lacked a dedicated ARM optimization stage, as firstly intended. Small optimizations were made in the C code to be more efficient for ARM, but a lot of space remains for improving.

The debugging library aims to provide the tools for recording and accessing debugging and error information. The data-structures and general functions were developed. The implementation of the functions for a particular type of error was done for one of them. A pitfall was the lack of unit-testing for the functions defined in this library (only manual testing was done).

The work done for this dissertation can be continued on several directions in future work. In the mathematical background part more methods can be studied and more algorithms can be implemented and analysed. Another promising direction would be studying combinations of different methods in one algorithm to achieve a better performance.

For the basic library several improvements can be done. First of them would be adding support for more fixed-point types. Second would be improving performance for basic library's non-saturated versions.

For the math library future work would mainly be focused on increasing the computational efficiency of the algorithms without disrupting their accuracy. Another possible improvement would be optimizing the division operation to be more efficient

in terms of instructions needed. Regarding debugging library, implementing support for new types of errors or debugging information can be part of future work.

The work done for this dissertation revealed several insights. First, efficient algorithms can be developed for almost every operation. It implies work on the theoretical background part and also on the implementation part. Good results for accuracy and efficiency cannot be achieved by improving just one of the parts; both of them have to be optimal. The second insight was that design simplicity is important to achieve both ease of use and computational efficiency. The best example to support the previous statement is the way used for defining new fixed-point types. The third insight concerns the development process. Auxiliary tools that are needed may be required to be built alongside with actual development. As an example is the testing framework. It required to be built and it consumed time and resources that were not planned from the beginning.

The work performed for this dissertation can be viewed as a starting point in the matter of fixed-point computation for energy efficient architecture. More work can be done to improve the areas showed earlier but the work done is a reliable point of start. The software product resulted can used in the SpiNNaker development process, at least with prototyping purposes. Considering this, it can be concluded that the project achieved its main purpose of exploring the fixed-point computations in the context of the SpiNNaker project.

Appendix A

In this appendix a list of algorithms described in pseudo-code is presented. The function *normalization* is not defined, it is just called. It returns two values: *the normalized value* and *the power of 2 used* (see equation (18)). For each lookup table, a lookup function exists that returns the value corresponding a certain entry.

Newton – Raphson Division

Input: a, x

Output: y = a/x

```
z ,k := normalization(x)
y :=  -32/17 * z + 48/17
y := y ∗ (2 − z ∗ y)
y := y ∗ (2 − z ∗ y)
y := y ∗ (2 − z ∗ y)
y := y * a
y >>= k
return y
```

Number of operations: 13 + normalization cost

Functional Iteration Division

Input: a, x

Output: y = a/x

```
z ,k := normalization(x)
t ≔ z − 1
y := 2 – z
t := t*t
y := y*(1-t)
t := t*t
y := y*(1-t)
t := t*t
y := y*(1-t)
t := t*t
y := y*(1-t)
y := y*a
y >>= k
return  y
```

Number of operations: 16 + normalization cost

Square Root Newton-Raphson

Input: x

Output: y

```
z, k := normalization(x)
y := 1.78773 – z * 0.80999
y := 0.5 * y * (3 – z * y * y)
y := 0.5 * y * (3 – z * y * y)
y := 0.5 * y * (3 – z * y * y)
y := z * y
if k is uneven
    y := y * sqrt(2)
end
y >>= [k/2]
return y
```

Number of operations: 19 + normalization cost

Logarithm Taylor series

Input: x

Output: y

Lookup tables:

table_a – values for a_i

table_lna – values for $\ln(a_i)$

```
z, k := normalization(x)
y  := z * table_a(z)
y := y – 1
y := y – 0.5 * t * t
y := y – table_lna(z)
y := y + k * ln(2)
return y
```

Number of operations: 10 + normalization cost

Bibliography

[1] APT Group, School of Computer Science, University of Manchester, "SpiNNaker Home," [Online]. Available: http://apt.cs.man.ac.uk/projects/SpiNNaker/project/. [Accessed 26 April 2012].

[2] L. A. Plana, D. Clark, S. Davidson, S. Furber, J. Garside, E. Painkras, J. Pepper, S. Temple and J. Bainbridge, "SpiNNaker: Design and Implementation of a GALS Multi-Core System-on-Chip," *ACM Journal on Emerging Technologies in Computing Systems Vol. 7, No. 4, Article 17,* pp. 17:1 - 17:18, Dec. 2011.

[3] ARM, "ARM968 Processor," [Online]. Available: http://www.arm.com/products/processors/classic/arm9/arm968.php. [Accessed 26 April 2012].

[4] APT Group, School of Computer Science, University of Manchester, "SpiNNaker Project - Architectural Overview," [Online]. Available: http://apt.cs.man.ac.uk/projects/SpiNNaker/architecture/. [Accessed 26 April 2012].

[5] Xin Jin, Francesco Galluppi, Cameron Patterson, Alexander Rast, Sergio Davies, Steve Temple and Steve Furber, "Algorithm and Software for Simulation of Spiking Neural Networks on the Multi-Chip SpiNNaker System," in *2010 International Joint Conference on Neural Networks (IJCNN 2010)*, Barcelona, Spain, 18-23 July, 2010.

[6] APT Group, School of Computer Science, University of Manchester, "SpiNNaker Project - The SpiNNaker Chip," [Online]. Available: http://apt.cs.man.ac.uk/projects/SpiNNaker/SpiNNchip/. [Accessed 26 April 2012].

[7] A. Group, "SpiNNaker web page," [Online]. Available: http://apt.cs.man.ac.uk/projects/SpiNNaker/architecture/. [Accessed 16 August 2012].

[8] Xin Jin, S.B.Furber, J.V.Woods , "Efficient Modelling of Spiking Neural

Networks on a Scalable Chip Multiprocessor," in *Proceedings 2008 International Joint Conference on Neural Networks, IJCNN* , June 2008 .

[9] A. Limited, ARM Architecture Reference Manual.

[10] ISO, "ISO website," [Online]. Available: http://www.iso.org. [Accessed 1 May 2012].

[11] ISO/IEC, "Technical Report TR 18037," 2006.

[12] "libfixmath - Cross Platform Fixed Point Math Library," Google, [Online]. Available: http://code.google.com/p/libfixmath. [Accessed 1 May 2012].

[13] MathWorks, Matlab Primer R2012a, 2012.

[14] Eclipse Foundation, "Eclipse IDE for C/C++ Developers," [Online]. Available: http://www.eclipse.org/downloads/moreinfo/c.php. [Accessed 3 May 2012].

[15] C. Brej, "KMD - Komodo," [Online]. Available: http://brej.org/kmd/. [Accessed August 15 2012].

[16] G. B. Thomas and R. L. Finney, Calculus and Analytic Geometry, Addison Wesley, 1996.

[17] M. Wolfram, "Cauchy Remainder," Wolfram Research, [Online]. Available: http://mathworld.wolfram.com/CauchyRemainder.html. [Accessed 3 May 2012].

[18] A. Omondi, Computer Arithmetic Systems, Prentice Hall, 1994.

[19] T. Rivlin, Chebyshev polynomials: from approximation theory to algebra and number theory, John Wiley and Sons, 1990.

[20] I. S. Milton Abramovitz, Handbook of Mathematical Functions, New York: Dover Publications, 1964.

[21] C. Tung and A. Avizienis, "Combinational arithmetic systems for the approximation of functions," in *Proceedings of the May 5-7, 1970, spring joint computer conference*, Atlantic City, New Jersey, 1970.

[22] F. Acton, Numerical Methods that work, Washington DC: The Mathematical Association of America, 1990.

[23] [Online]. Available: http://darwins-god.blogspot.co.uk/2010/11/adaptive-robots-yet-more-evidence-for.html. [Accessed 15 August 2012].

[24] A. Ralston, A first course in numerical analysis, 1965: McGraw - Hill.

[25] J. Volder, "The CORDIC Trigonometric Computing Technique," *IRE Transactions on Electronic Computers ,* pp. 330-334, 1959.

[26] J. Walter, "A unified algorithm for elementary functions," in *Proc. of Spring Joint Computer Conference*, 1971.

[27] Wikipedia, "Wikipedia," [Online]. Available: http://en.wikipedia.org/wiki/File:CORDIC-illustration.png. [Accessed 16 August 2012].

[28] Wikipedia, "CORDIC," [Online]. Available: http://en.wikipedia.org/wiki/CORDIC. [Accessed 10 August 2012].

[29] B. Parhami, Computer Architecture: Algorithms and Hardware Design, Oxford University Press.

[30] S. E. Anderson, "Bit Twiddling Hacks," CS Stanford, [Online]. Available: http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetTable. [Accessed 4 May 2012].

[31] R. Osherove, The Art Of Unit Testing, Manning, 2009.

[32] Wikipedia, "Agile software development," [Online]. Available: http://en.wikipedia.org/wiki/Agile_software_development. [Accessed 14 August 2012].

[33] J. Design, "JTN002 - MinUnit -- a minimal unit testing framework for C," [Online]. Available: http://www.jera.com/techinfo/jtns/jtn002.html. [Accessed 15 August 2012].

[34] D. B. a. M. D. Mike Banahan, "The C Book - online version," [Online]. Available: http://publications.gbdirect.co.uk/c_book/chapter6/bitfields.html.

[Accessed 17 August 2012].

[35] ARM, "ARM Instruction Reference," ARM, [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0068b/CIHBJEHG.html. [Accessed 2012 August 22].

[36] J. Nielsen, "Usability 101: Intorduction to Usability," [Online]. Available: http://www.useit.com/alertbox/20030825.html. [Accessed 24 August 2012].

[37] G. N. W. E. T. Whittaker, A course in modern analysis, Cambridge University Press.