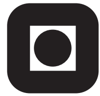

NTNU – Trondheim
Norwegian University of
Science and Technology

Implementation of the Epileptic Seizure Prediction Algorithm on the SHMAC Platform

Sunniva Nergaard Berg

Electronics System Design and Innovation
Submission date: June 2014
Supervisor: Per Gunnar Kjeldsberg, IET

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Problem Description

Candidate name: Sunniva Nergaard Berg

Thesis title: Implementation of the Epileptic Seizure Prediction Algorithm on the SHMAC Platform

Problem Description

The Single-ISA Heterogeneous MAny-core Computer (SHMAC) is an ongoing research project within the Energy Efficient Computing Systems (EECS) strategic research area at NTNU. SHMAC is planned to run in an FPGA and be an evaluation platform for research on heterogeneous multi-core systems. Due to battery limitations and the so called Dark silicon effect, future computing systems in all performance ranges are expected to be power limited. The goal of the SHMAC project is to propose software and hardware solutions for future power-limited heterogeneous systems.

An existing algorithm for epileptic seizure prediction is going to be mapped on the SHMAC platform. An important part of this algorithm is calculation of a Short-Term maximum Lyapunov exponent (STLmax). There currently exist several versions of this STLmax calculation, both in Matlab and in C, and at different levels of optimization, e.g., floating point and fixed point versions.

The main parts of this assignment are as follows:

- Study the epileptic seizure prediction algorithm and its STLmax calculation.
- Study different approaches found in the literature for mapping of algorithms on heterogeneous platforms consisting of general processing units and hardware accelerators.
- Adapt the STLmax calculation to the SHMAC single instruction set architecture and run it on a general SHMAC processing tile.
- Select one or more kernels in the STLmax calculation and adapt the general processing tile for its efficient execution.
- Evaluate performance and energy gains achieved as well as area overhead.

Supervisor: Per Gunnar Kjeldsberg

Abstract

This study concerns mapping of the Epileptic Seizure Prediction Algorithm to the SHMAC platform by analysing the algorithm and designing specialized hardware. The work is constituted through consideration of design approaches and variations within the different approaches and implementations of these designs. As the SHMAC platform uses the 32-bit floating point standard for number representations, the design approaches consists of either direct operations on floating point numbers or conversion from floating point to fixed point and operations with fixed point numbers within the module.

Two of the approaches in this study use the floating point and fixed point packages of VHDL and the *float* and *sfixed* types included in these packages, while the third uses standard VHDL and the *std_logic_vector* type. The approaches were analysed through simulation in *ISim* and synthesis using *Xilinx's XST*, and verified through a golden device-similar method.

The three approaches result respectively to each other as follows(these results assume utilization RAM resources in synthesis):

- Float package: Low design time, low speed, high area consumption.
- Fixed package: Low design time, medium speed, medium area consumption.
- Standard VHDL: High design time, high speed, low area consumption.

To conclude: standard VHDL appears to be the best choice due to the high speed and low area. It especially appears promising to keep to the standard VHDL as the *Xilinx* platform does not seem fully compatible with the float and fixed point packages. Therefore it seems more likely to avoid unexpected problems and incompatibility with the surrounding interface when the time comes for implementing the accelerator to the platform.

Sammendrag

Denne avhandlingen omhandler implementering av "the Epileptic Seizure Prediction Algorithm" til SHMAC plattformen ved å analysere algoritmen og designe spesialisert hardware. Arbeidet er gjennomført ved å vurdere ulike fremgangsmåter for design og variasjoner innad i de ulike fremgangsmåtene samt implementering av disse designene. Ettersom SHMAC plattformen bruker 32-bit floating point standarden for nummerrepresentasjon, består fremgangsmåtene enten av direkte behandling av floating point tall eller konvertering fra floating point til fixed point og deretter behandling av fixed point tall innad i modulen.

To av fremgangsmåtene i denne avhandlingen bruker floating point og fixed point pakkene i VHDL samt *float*- og *sfixed*-typene inkludert i disse pakkene, mens den tredje bruker standard VHDL og *std_logic_vector*-typen. Fremgangsmåtene ble analysert ved simulering i *ISim* og syntetisert ved bruk av *Xilinxs XST* og verifisert ved å bruke en golden device-lik metode.

De tre fremgangsmåtene resulterte i det følgende, respektivt hverandre(resultatene antar utnyttelse av RAM ressursene ved syntese):

- Float package: Lav designtid, lav hastighet, høyt arealforbruk.
- Fixed package: Lav designtid, medium hastighet, medium arealforbruk.
- Standard VHDL: Høy designtid, høy hastighet, lavt arealforbruk.

For å konkludere: standard VHDL virker å være det beste valget, grunnet høy hastighet og lavt areal. Det virker særlig lovende å holde seg til standard VHDL ettersom *Xilinx* plattformen ikke virker fullstendig kompatibel med float og fixed point pakkene. Derfor virker det mer sannsynlig å unngå uventede problemer og inkompatibilitet med det omsluttende grensesnittet ved bruk av standard VHDL, når tiden kommer for implementering av akseleratoren til plattformen.

Preface

This thesis constitutes the end of my Master of Science degree in Electronics at the Norwegian University of Technology and Science.

Throughout the work on this study, I have received a lot of great support from my supervisor Per Gunnar Kjeldsberg. I would like to thank him, saying that without his guidance, contributions and advice, this work would have been most challenging. I would also like to thank Håkon Wikene who helped me running of codes and profiling on the SHMAC platform, and the others working on SHMAC who have contributed expertise within their respective areas.

Sunniva Nergaard Berg

Contents

Abstract i

Sammendrag iii

Preface v

Contents vi

List of Figures ix

List of Tables xi

Abbreviations xiii

1 Introduction 1

2 Theory and Related Work 9
- 2.1 SHMAC . . . 9
- 2.2 Heterogeneous systems . . . 11
- 2.3 Accelerators . . . 12
- 2.4 Partitioning of Applications . . . 15
- 2.5 Number representations . . . 17
- 2.6 Lyapunov Exponent . . . 20

3 Analysis of the Lyapunov Exponent Calculation Algorithm 27
- 3.1 General profiling . . . 28
- 3.2 System specific profiling . . . 29

4 Implementation and Verification 35
- 4.1 Design . . . 35
  - 4.1.1 Approach expansion package, floating point . . . 37
  - 4.1.2 Approach expansion package, fixed point . . . 39
  - 4.1.3 Approach standard VHDL, fixed point . . . 39
- 4.2 Verification . . . 41

5 Results 45
- 5.1 Results from Simulation . . . 45

5.2 Results post Synthesis 48
5.2.1 Approach expansion package, floating point 49
5.2.2 Approach expansion package, fixed point 49
5.2.3 Approach standard VHDL, fixed point 50
5.2.4 Power estimates 51

6 Discussion 53

7 Conclusion and Future work 59

A Profiling of C-similar Matlab-code 61

B Timing of most time consuming calculations 63

C Deviations 69

D VHDL codes 75
D.1 Float approach 75
D.1.1 Design 1 75
D.1.2 Design 2 77
D.2 Fixed approach 79
D.2.1 Design 1 79
D.2.2 Design 2 81
D.3 Standard VHDL 82

E Verification scripts 93
E.1 C-script calculation differences 93
E.2 Matlab-script for sorting 94

Bibliography 95

List of Figures

1.1 Mean value(a) and standard deviation(b) of EEG over time from 5 patients[1]. Vertical lines denote seizure onset. 2
1.2 STL_{max}over time from 5 patients[1]. Vertical lines denote seizure onset. . 3
1.3 Control of seizures in a rat using the epileptic seizure prediction algorithm and different schemes of stimuli via electrodes[2]. 4
1.4 The tile-based architecture of the SHMAC platform[3]. 6

2.1 FPGA 10
2.2 Amber Tile 10
2.3 Process execution tree 13
2.4 Difference in execution on single and multi threaded CPUs (Figure inspired from [4, p. 361]) 13
2.5 Effect of unrolling 14
2.6 (a) Source-based and (b) binary-based partitioning[5]. 16
2.7 How floating point numbers are represented. 17
2.8 How fixed point numbers are represented. 18
2.9 Algorithm for addition/subtraction in FPU[6]. 19
2.10 (a)STL_{max} values between two seizures. (b) T-index for the three most optimal groups of sites between two seizures[1]. 22
2.11 Evolution of points along the fiducial trajectory for Lyapunov exponent calculation[7]. 23
2.12 Flowchart of algorithm for calculating Lyapunov exponents using fixed evolution time as described by Wolf *et al.*[7]. 24

3.1 Illustration of codes available for profiling. 27

4.1 Illustration of general module. 37
4.2 Illustration of the module using standard VHDL. 40
4.3 Illustration of the technique used for verification. 42

5.1 Waveform from simulation of a module without(left) and with(right) *Readout*-module. 47
5.2 7 cycles between outputs for design without *Readout*-module and no unrolling of *k*. 48

6.1 Graph showing area and speed(in estimated new total runtime, new sec) of different design choices. 54

6.2 Graph showing area and speed(in estimated new total runtime, new sec) of different design choices. 55
6.3 Graph showing area and speed(in estimated new total runtime, new sec) of the approach implementing fixed point from std_logic_vector. 56

List of Tables

1.1 Dennadrian versus Post-Dennardian scaling [8] 5

3.1 Profiling of Matlab code working on matrices. 28
3.2 Most run lines from Matlab profiling 29
3.3 Total execution time 30
3.4 Cycles for instructions in Fix1 code if rewritten for integer utilization. Addition and subtraction spend same amount of cycles. 30
3.5 System specific profiling, calculation times for different code versions in milliseconds. 31
3.6 System specific profiling, percentage of total runtime spent on calculations. 31

4.1 Area and performance for float extension package approach 38
4.2 Area and performance for fixed extension package approach 39
4.3 Area and performance for standard VHDL fixed point approach 40
4.4 Deviation for float expansion package approach 43
4.5 Deviation for fixed expansion package approach 44
4.6 Deviation for standard VHDL approach 44

5.1 Simulation time from float and fixed extension package approach, and the standard VHDL approach 46
5.2 New timing for float extension package approach 49
5.3 New timing for fixed extension package approach 50
5.4 New timing for standard VHDL approach 50
5.5 Table of power estimates. 51

6.1 Table which compares qualities of the different approaches. 57

A.1 Profiling of C-syntax like Matlab code 61

B.1 Timing of most time consuming calculations, algorithm float 1 63
B.2 Timing of most time consuming calculations, algorithm float 2 64
B.3 Timing of most time consuming calculations, algorithm fix 1 65
B.4 Timing of most time consuming calculations, algorithm fix 1 66
B.5 Timing of most time consuming calculations, algorithm no malloc 66
B.6 Timing of most time consuming calculations, algorithm FloatOpt from fixed point experiences. 68

C.1 Deviation for float extension package approach 70

C.2 Deviation for float extension package approach 72
C.3 Deviation for the approach using standard VHDL 74

Abbreviations

APB Advanced Peripheral Bus

BRAM Block Random Access Memory

CPU Central Processing Unit

CSL Configurable System Logic

EEG Electroencephalogram

FPGA Field-Programmable Gate Array

FPU Floating Point Unit

ISA Instruction Set Architecture

ISE Instruction Set Extention

ISEGEN ISE Generation

L_{max} Maximum Lyapunov exponent

LUT LookUp Table

RAM Random Access Memory

RISC Reduced Instruction Set Computer

SHMAC Single-ISA Heterogeneous MAny-core Computer

STL_{max} Short-term maximum Lyapunov exponent

STP System ThroughPut

VHDL VHSIC Hardware Description Language

$VHSIC$ Very High Speed Integrated Circuit

ZBT Zero Bus Turnaround

Chapter 1

Introduction

The Single-ISA Heterogeneous MAny-core Cumputer, SHMAC, project is an ongoing research project at NTNU. This is planned to be a platform for evaluation of heterogeneous systems. The platform is continuously changing and evolving, i.e. increasing the quality of the system. As for the beginning of this project, there is for instance no hardware support for floating point numbers, which will prove crucial for the scope of this thesis, to map an epileptic seizure prediction algorithm to the SHMAC platform.

Epilepsy is a neurological disorder characterized by epileptic seizures. These seizures are results of a temporary electrical disturbance of the brain[1]. The seizures are followed by postictal slowing and disorganization of the EEG rhythm, and can incapacitate the patient during as well as after the seizure[9]. Epileptic seizures can, if not controlled, have a severe impact on the patients social, vocal and educational activities, and thereby reduce the patient's quality of life[2]. 1% to 2% of the world's population, at all ages, are affected by this disorder. It was previously believed that seizures occurred abruptly and could not be anticipated[2]. This, however, proved to be incorrect.

The algorithm proposed by Iasemidis *et al.* [1][9] aims to be able to predict an oncoming seizure and thus enable the possibility of implantable devices for diagnostic or therapeutic purposes. The epileptic seizure prediction suggested by Iasemidis *et al.*[1][2][9], use the EEG recordings of the brain and utilize these recordings to predict an oncoming seizure. These recording, however, does not change consistently across patients: Figure 1.1 illustrates EEG recordings before, during and after a seizure in five patients. For example, the mean recording shows an upward trend in Patient 4, but no significant trend is visible in the other patients before the seizure.

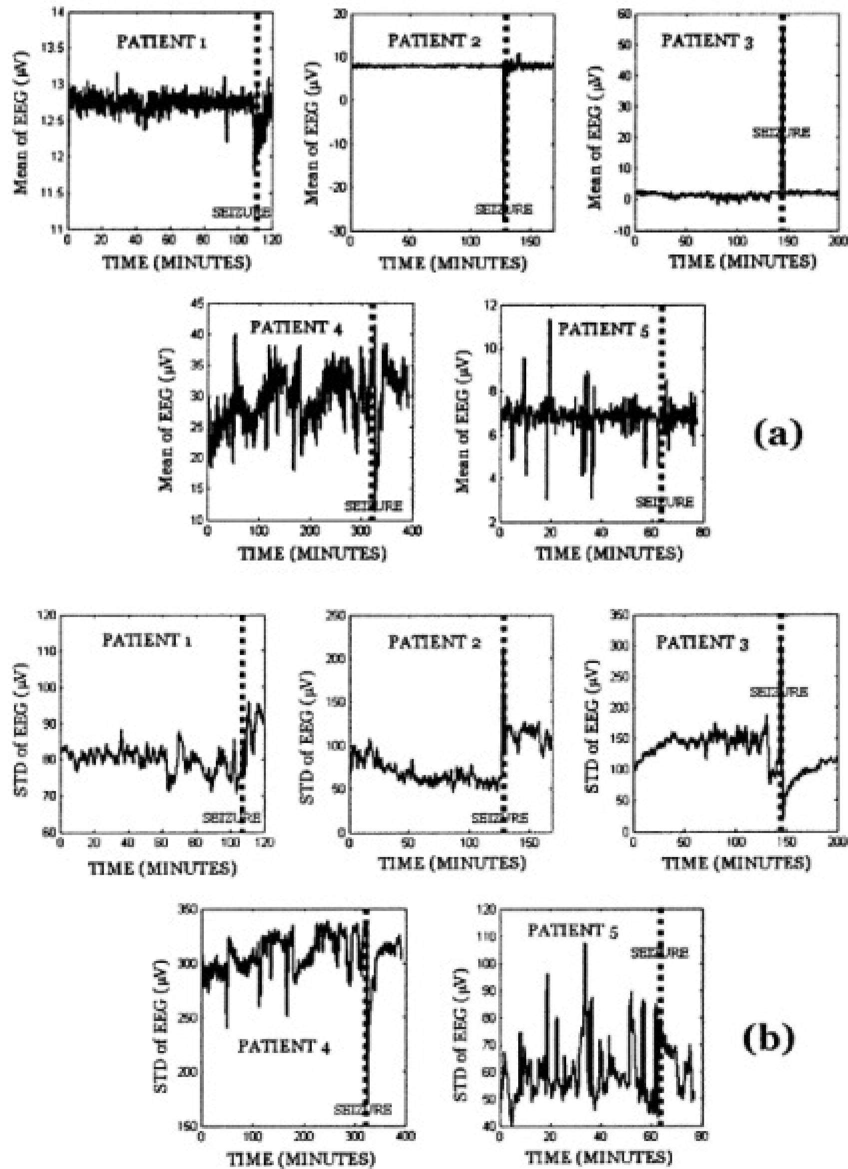

FIGURE 1.1: Mean value(a) and standard deviation(b) of EEG over time from 5 patients[1]. Vertical lines denote seizure onset.

This inconsistency applies to all cases, before, during and after the seizure, across all five patients. Because of this inconsistency, an other measure called Short-term maximum Lyapunov exponent, STL_{max}, is introduced. STL_{max} exponents quantify the rate of production/destruction of information within short periods of time (here 10.24s) of EEG measurements[1]. Looking at the STL_{max} values in Figure 1.2 for the same patients in the same timespan, it is obvious that STL_{max} values change consistently across patients. All the STL_{max} values are higher after than before the seizure, indicating that STL_{max} values can be used for seizure prediction.

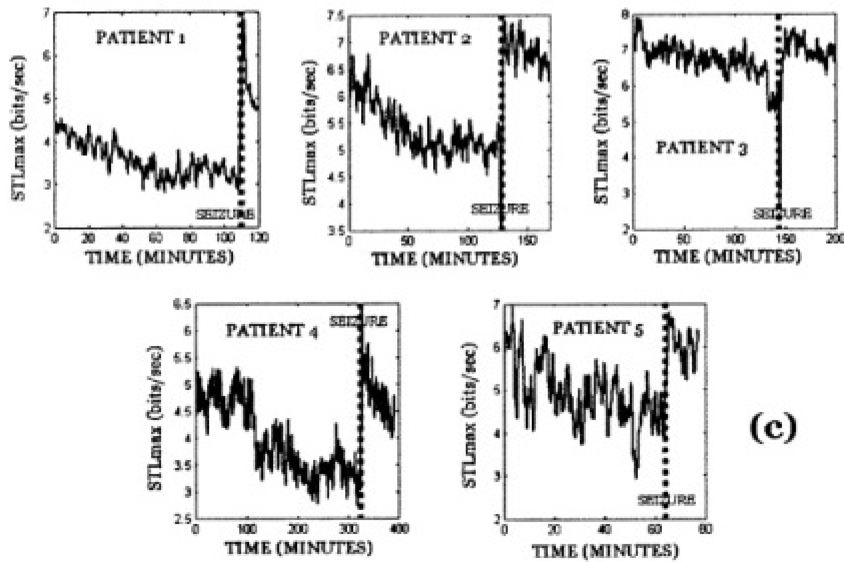

FIGURE 1.2: STL_{max}over time from 5 patients[1]. Vertical lines denote seizure onset.

A brain in a normal state is characterized by chaos, i.e. that measurements of production or destruction of information in different brain sites are incoherent. These measurements are carried out by calculating Lyapunov exponents of the system, in this case the nonlinear, dynamical system - the brain. Lyapunov exponents are amongst the most important measures of the dynamics of a linear or nonlinear system[2]. These exponents measure average information flow in bits per second. Positive exponents indicate generation of information while negative exponents indicate destruction.
For a dynamical system to be chaotic, at least one of the system's Lyapunov exponents needs to be positive[2]. Due to this, the calculation of Lyapunov exponents is crucial to determine whether or not the system is chaotic in the epileptic seizure prediction algorithm. The calculation will be further elaborated in Chapter 2.

The algorithm has also been tested on rats, using electrodes implanted in their brains and stimulating the rats brains according to warnings issued by the prediction algorithm. Using different schemes of stimuli, the combination of the seizure prediction algorithm and electrical pulses to stimulate the brain seems promising, as illustrated in Figure 1.3.

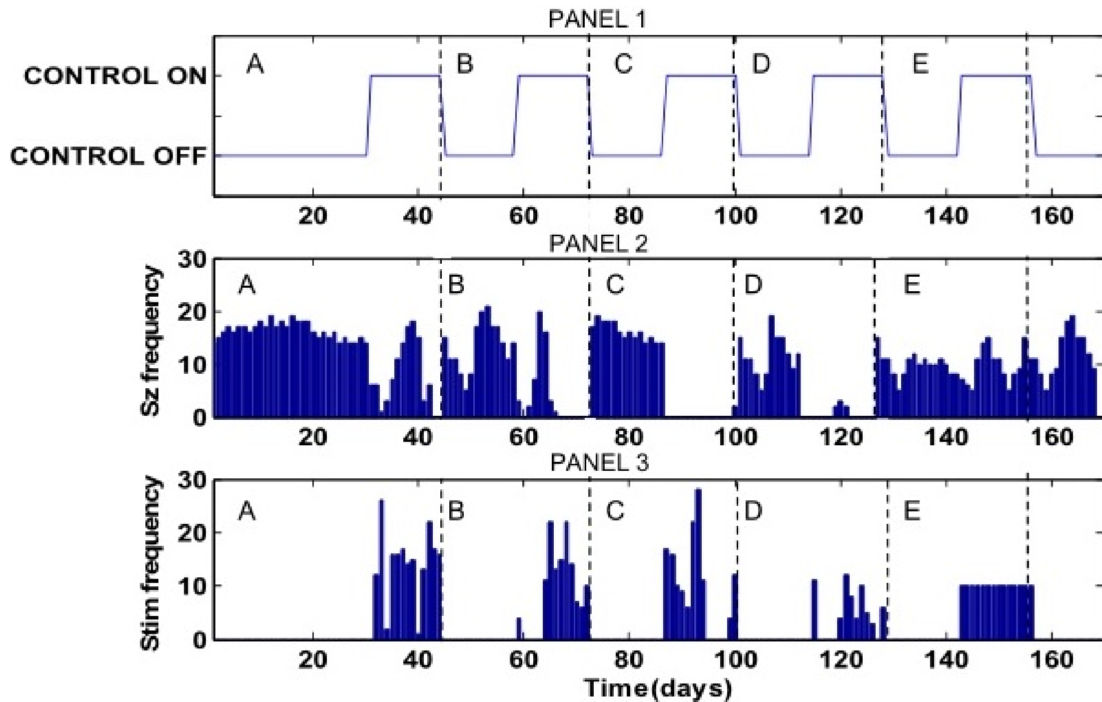

FIGURE 1.3: Control of seizures in a rat using the epileptic seizure prediction algorithm and different schemes of stimuli via electrodes[2].

Schemes C and D in Figure 1.3 seem to reduce the numbers of seizures to zero for days at a time. The epileptic seizure prediction thus appears to be a useful application, and could possibly be used in human implants in the future. It is, however, important that implants are as energy efficient as possible, as changing batteries or harvesting energy inside the body is a challenging task.

The scope of this thesis is to map the epileptic seizure algorithm to the SHMAC platform. The Single-ISA Heterogeneous MAny-core Computer(SHMAC) platform, is as previously stated, an ongoing research project within the research initiative Energy Efficient Computing Systems(EECS) at the Faculty for Information Technology, Mathematics and Electrical Engineering, NTNU. SHMAC is planned to be an evaluation platform for research on heterogeneous multi-core systems. Heterogeneous systems are required partly due to the Dark Silicon effect.

As an important goal of the SHMAC platform also is to be energy efficient, implementing this algorithm to the SHMAC platform could be of beneficial use both for the purpose of implementing the algorithm itself, and to explore the efficiency attainable on the SHMAC platform.

The future of electronic devices is expected to be power limited. This is both due to the limitation of batteries and the Dark silicon effect. The Dark Silicon effect comes from the breakdown of Dennardian scaling. Dennardian scaling used the concept of scaling the threshold voltage to improve energy efficiency. The formula for a chip's power consumption is shown in Table 1.1. As the power relates to the voltage squared, scaling the voltage will result in an improvement of S^2 in energy efficiency.

The relationship between Dennardian scaling and Post-Dennardian scaling is exposed in Table 1.1. The scaling factor S comes from the ratio between the feature size of two processes, for example the ratio S = 1.4× for two processes of 32 and 22 nm[8]. Post-Dennardian scaling is also called leakage-limited scaling, and in that regime, one cannot exploit the fact that reduction in threshold voltage gives a reduction in operating voltage resulting in a S^2 improvement of energy efficiency. This is what distinguishes Dennardian from post-Dennardian scaling, and is as shown in lines 4 through 6 in Table 1.1. This also causes the shortfall of S^2 for each process generation. The percentage of chips that needs to be powered off, or dark, will thus increase.

Scaling		
Transistor property	Dennardian	Post-Dennardian
$\Delta Quantity$	S^2	S^2
$\Delta Frequency$	S	S
$\Delta Capacitance$	$1/S$	$1/S$
ΔV_{dd}^2	$1/S^2$	1
$\Rightarrow \Delta Power = \Delta QFCV^2$	1	S^2
$\Rightarrow \Delta Utilization = 1/Power$	1	$1/S^2$

Table 1.1: Dennadrian versus Post-Dennardian scaling [8]

The SHMAC platform is a tile-based architecture, illustrated in Figure 1.4. The interconnections are in a mesh grid, meaning that all tiles have connections to their north, east, south and west, excluding tiles at the boundaries.

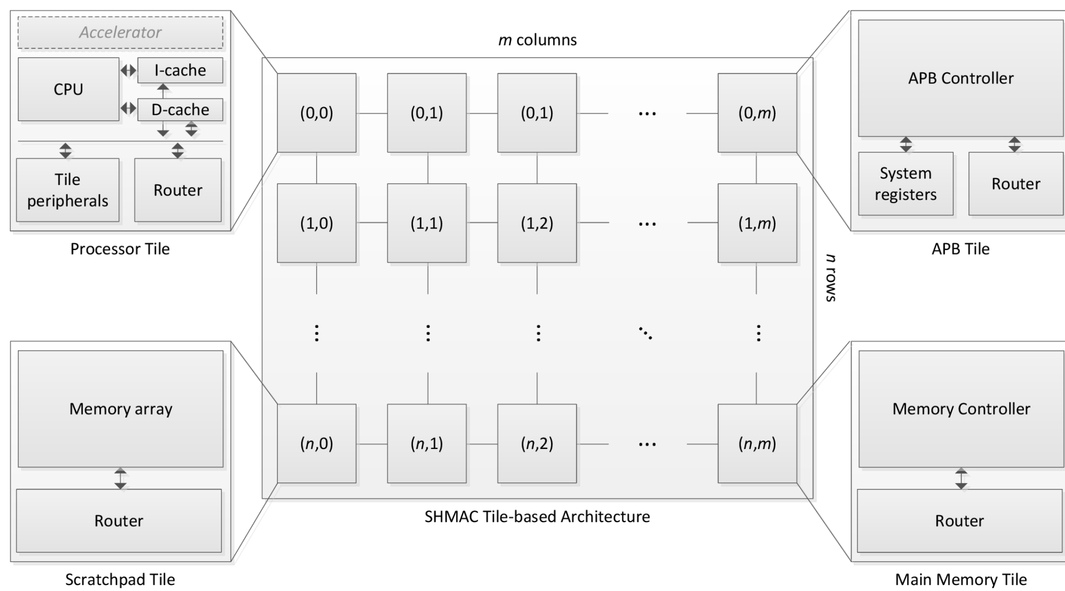

FIGURE 1.4: The tile-based architecture of the SHMAC platform[3].

The platform supports tiles of types[3]:

- Processor tile
- Scratchpad tile
- Main Memory tile
- APB Interface tile
- Dummy tile

For this project, the processor tile is the relevant tile. A tile of this type primarily consists of a processor, caches, peripherals and a router, but can also be equipped with accelerators. Accelerators are execution units optimized for their task.

The idea with the SHMAC-project, is to reduce the Dark Silicon effect by two main strategies, of which exploiting a heterogeneous architecture is one. A heterogeneous architecture is an architecture consisting of different resources, each included to achieve higher performance or a more energy efficient system[10].

By including cores both for high performance and for energy efficiency, one can dynamically schedule a system to have high performance or low energy consumption[11]. Heterogeneous systems will be further discussed in Chapter 2. Another way of addressing

the problem of the Dark Silicon effect, is to add application specific elements, accelerators, to the architecture that are very energy efficient at their task, and can be turned off if their task does not occur. This will actually increase the heterogeneity of the system, and is as such not a completely different strategy compared to the utilization of a heterogeneous architecture. Accelerators will be further elaborated in Chapter 2. The objective of this study is to map the Epileptic Seizure Prediction Algorithm on the SHMAC Platform by examining the algorithm and investigate parts of the code that could benefit from the implementation of specialized hardware.

The remaining of the thesis is organized as follows. Chapter 2 will go into details regarding the SHMAC platform and related theory, calculation of the Lyapunov exponent, partitioning of applications, and number representations. Analysis of the codes are presented out in Chapter 3 including both general and system specific profiling. Chapter 4 concerns design approaches, data from synthesis and verification of designs. Results from simulation in *ISim* and post synthesis estimations are placed in Chapter 5 and the results and how they are affected by the synthesis results are discussed in Chapter 6. Chapter 7 contains suggestions for future work and the conclusion.

Chapter 2

Theory and Related Work

As stated in Chapter 1, two main ideas have been suggested to address the Dark Silicon effect on the SHMAC platform. One is implementing a heterogeneous system and the other is adding accelerators to the architecture, increasing its heterogeneity. This chapter will go more into detail about what heterogeneous systems and accelerators are and different ways to represent numbers, as well as the calculation of the Lyapunov exponents used in the algorithm related to this work.

2.1 SHMAC

The SHMAC platform is, as previously stated, a research project at NTNU. The platform is planned to be an evaluation platform for research on heterogeneous multi-core systems. The platform's architecture is described in the following.

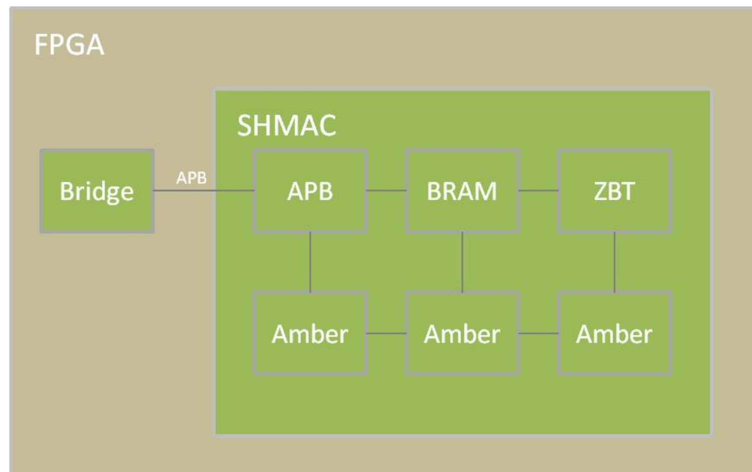

FIGURE 2.1: FPGA

As can be seen in Figure 2.1, the SHMAC platform consists of several tiles, here represented by APB, ZBT, BRAM and Amber tiles. The Amber tiles are equivalent to the processor tiles described in the project plan for the SHMAC project [3]. A lower level of abstraction than the Amber Tile is the Amber Wrapper. Illustrated in Figure 2.2, the wrapper excludes the router. Within the next level of abstraction from the Amber wrapper, one finds the Amber system, which amongst other, contains the Amber CPU [12].

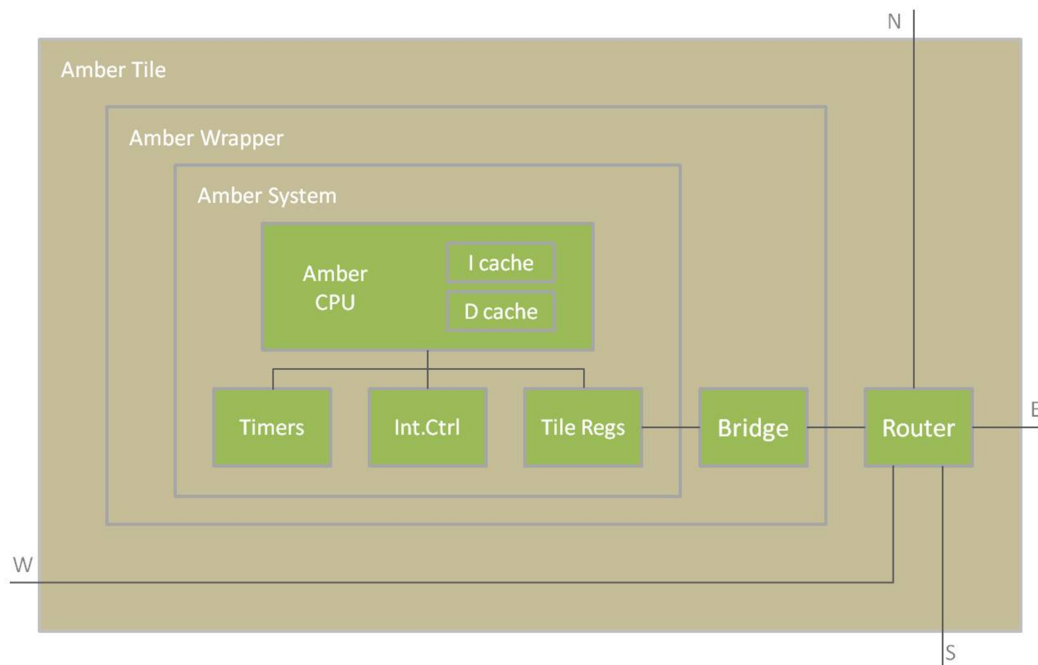

FIGURE 2.2: Amber Tile

The Amber processor core is a 32-bit RISC processor running at 60 MHz with a 5-stage pipeline, which is fully compatible with the ARM v2a ISA[12]. An important detail for this project is the delay of each instruction in the Amber core. As can be found in the reference manual for the Amber core[12], register based instructions are executed in one cycle, excluding instructions involving multiplication. Multiplication as well as multiply-accumulate operations take 34 clock cycles, making these operations obvious bottlenecks amongst the standard instructions. The multiplication on the Amber core uses the Booth algorithm[13], which is a small and slow implementation of a multiplier. The algorithm repeatedly adds one of two predetermined values to a product and then makes a rightward arithmetic shift on the product. The complete algorithm can be found in the Amber manual[12] or further reading in the article *"A signed binary multiplication technique"* by Booth[13].

2.2 Heterogeneous systems

A heterogeneous system is a system consisting of different resources. Which, relating to the SHMAC platform, would say that the different tiles in the SHMAC platform can have different characteristics due to additional units on the tile, i.e. an accelerator, additional memory or other units not featured on a standard tile.

Heterogeneity can be classified in three classes. The weakest form for heterogeneity, is systems having the same ISA and micro architecture across the cores, only varying in clock frequency. Then there is a stronger form where the cores have the same ISA, but have different micro architectures. The strongest form of heterogeneity is systems which have both different ISAs and micro architectures. The SHMAC is Single-ISA, meaning that the different cores all implement the same instruction set architecture.

K. V. Craeynest and L. Eeckhout [11] present some results regarding homogeneous versus heterogeneous designs. Compared to homogeneous designs, heterogeneity yields a better performance, up to 32% of a weighted speed-up, STP. This increase comes from heterogeneous systems' ability to map jobs to cores most appropriate. As heterogeneous systems are more specialized than homogeneous, they are more complex, as the cores in homogeneous systems all contain of the same tiles, while the tiles within a heterogeneous system varies. This presumably makes a homogeneous system easier to implement.

The complexity of heterogeneous systems can, however, often be accounted for by the benefits regarding energy consumption and performance. When executing a program on a heterogeneous multi-core system, the program is divided into smaller parts, each executed on the most suitable core[14, p. 14], leading to speed-ups and/or less energy consumption.

2.3 Accelerators

The fastest processors are considered to be very expensive, so splitting the application and dividing it to be performed on several smaller processors can be cheaper, even when considering the cost of assembling those components [4, p. 354]. In addition, some algorithms may not map well onto a CPU, and may as such be better executed on a specially designated processing element, such as an accelerator.

An accelerator is a processing element attached to the CPU bus [4, p. 357]. This is to be able to execute tasks quickly. The basic concern with accelerators is how much gain can be achieved. This depends in part on whether the system in which it is to be integrated is single threaded or multi threaded: does the CPU wait for the accelerator while it is operating, or does the CPU run with the accelerator running in the background? Figure 2.3 illustrates a tree of processes, where process A1 is executed on an accelerator whilst the other four processes are executed on the CPU. Figure 2.4 illustrates the difference in executing the sequence of processes from the process tree in Figure 2.3 on a single threaded and a multi threaded CPU, respectively.

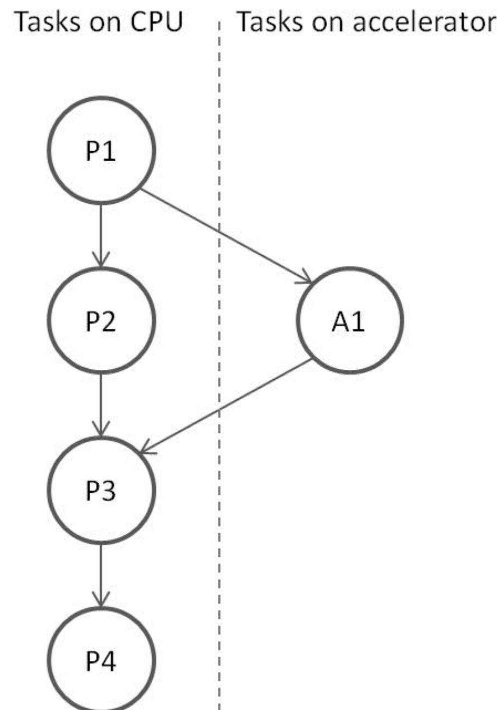

FIGURE 2.3: Process execution tree

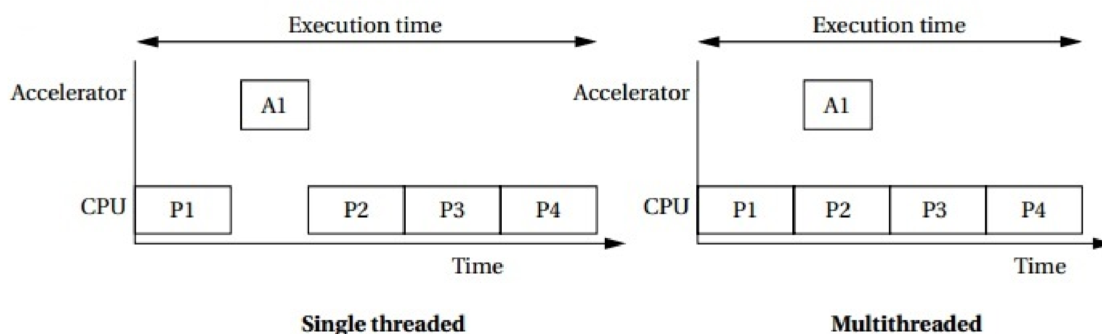

FIGURE 2.4: Difference in execution on single and multi threaded CPUs (Figure inspired from [4, p. 361])

A simple equation for the speed-up using an accelerator in a system is[4, p. 362]:

$$S = n(t_{CPU} - t_{accel}) = n[t_{CPU} - (t_{in} + t_x + t_{out})] \tag{2.1}$$

In this equation, n is the number of times the accelerated block is executed, t_{CPU} is the execution time of the function in software, t_{in} and t_{out} are the times spent reading and writing variables, and t_x is the execution time of the accelerator when all the data is available.

From this, one can conclude that it is of greater value to accelerate a task which is executed a significant number of times. In a single threaded system, the total execution time is reduced by S, while it is more complex for multi threaded systems. In multi threaded systems, there are several execution paths. So, to find the total execution time, one have to find the longest path from the beginning to the end of the execution.

An example of targets that are executed a significant number of times, are the ones inside loops, especially nested loops. In this case loop unrolling is an interesting approach for an accelerator. In the case of double nested for-loops, loop unrolling can be done in more or less extensive ways, unrolling both the for-loops entirely, one entirely and the other partly, or both partly. Unrolling both loops to the fullest, will yield the highest performing hardware regarding speed, but also the most area consuming, and vice versa. It is therefore important to evaluate how the calculation speed and area of this specific hardware are influenced by different degrees of unrolling, to find a optimal solution.

Figure 2.5 illustrates how unrolling will affect the area and performance. By having no unrolling only one calculation module is needed. This means lower speed as the one module has to be run $(sizeloop1) \times (sizeloop2) = n$ times, but it also means lower area consumption. By completely unrolling both for-loops, we would have the exact opposite situation; n parallel modules, high speed and large area consumption.

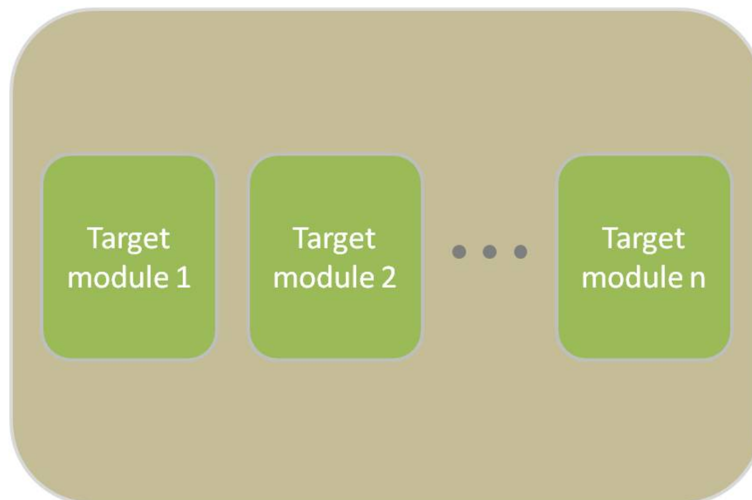

FIGURE 2.5: Effect of unrolling

Another aspect concerning accelerators is if they operate on a large amount of data. So, if this is the case, it can be beneficial to give it direct memory access. The accelerator

can thus read and write data directly, not having the CPU as a shuttle between the accelerator and the memory. In that case, the CPU and accelerator can communicate through a shared memory. To decide if an accelerator is needed, the specific system needs to be analysed. Other issues in the system, such as moving data, may be a bottleneck. In that case the inclusion of an accelerator may not be as beneficial as intended[4, p. 357].

2.4 Partitioning of Applications

Partitioning is a crucial aspect when one is to implement an accelerator to a system. The partitioning has a first order impact on the cost/performance of the final system[15] as it takes into account the gain and/or drawback of the resulting hardware or software blocks. The partitioning is, relating to this study, to map the Epileptic Seizure Prediction Algorithm partly onto an accelerator while the remaining code will run on the Amber CPU.

An approach to partitioning is the ISEGEN algorithm. This algorithm uses the Kernighan-Lin min-cut algorithm[16] for partitioning. The goal of Biswas *et al.*[17] is to make an iterative improvement technique to generate solutions close to the ones provided manually by expert designers. The algorithm starts with all nodes in either hardware or software and toggles each unmoved node from hardware to software or vice versa. For each toggled node, the gain function is computed to measure how the toggle affects the cost/performance of the system. The ISEGEN algorithm also takes into account how many outputs and inputs the resulting ISE will have. As an example, moving a node with one input and no output will increase the numbers of inputs and outputs of the ISE by one and zero, respectively.

A more straight forward approach is to examine the loop regions of a code and make the most time-consuming software into specialised hardware. Stitt and Vahid [5] applied such an approach by mapping critical loops of embedded applications to configurable system logic. They did, however, consider two different approaches to partitioning, source-based and binary-based. The two approaches are depicted in Figure 2.6.

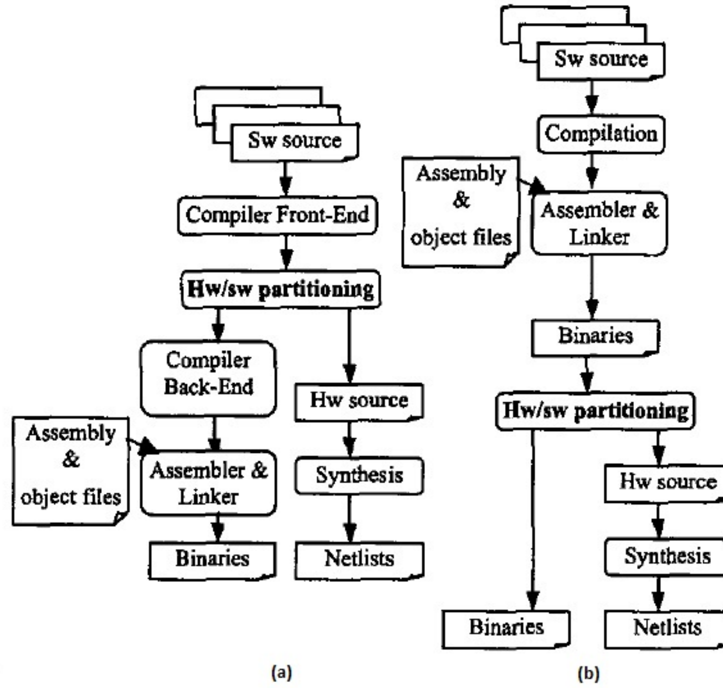

FIGURE 2.6: (a) Source-based and (b) binary-based partitioning[5].

For area and power characteristics they used Xilinx VirtexE. For estimation of total power consumption, Equation 2.2 was applied:

$$TotalPower = \%SW \times P_{SW} + \%CSL \times (P_{CSL} + 0.25 \times P_{SW}) + InterconnectPower + QuiecentPower \quad (2.2)$$

P_{CSL} is the power of the configurable system logic when it is active and P_{SW} is the power of the software when the microprocessor is active. It is also assumed that an inactive microprocessor uses $\frac{1}{4}$ of the power used while being active. Stitt and Vahids approach include implementing blocks and replacing the corresponding software region by a handshaking behavior[5]. Determination of potential improvements was done by converting C-code for the previously mentioned critical loops into VHDL-code.

Stitt and Vahid extracted the parallelism of the C-code and created appropriate hardware[5]. The authors also suggest to implement techniques such as loop unrolling and pipelining to achieve a greater speed-up. For partitioning at source-level, Stitt and Vahid achieved an average speed-up of 1.5 times and energy savings of 27% compared to running the benchmarks in pure software.

2.5 Number representations

Since the algorithm is written in C using float-numbers, the numbers being input to the accelerator will be represented in floating point. Floating point numbers have got this notation due to the fact that the decimal point is not set between a specific bit-pair, but is floating. The Amber core uses the 32-bit format of the IEEE Standard 754 for Floating-Point. According to the IEEE Standard for Floating-Point Arithmetic, the standard representation of 32-bit floating-point numbers is as follows[18]:

- Bit 31: sign
- Bits 30-23: exponent
- Bits 22-0: significand/mantissa

The floating point standard also requires that the most significant '1' in the number to be represented is removed and made implicit. Therefore, when converting to or from floating point, one always have to take this implicit 1 into account. This means that a number represented by floating point has implicitly 24 bits for the mantissa, resulting in an accuracy of $2^{-24} = 59.60 \times 10^{-9}$.

FIGURE 2.7: How floating point numbers are represented.

Floating point numbers can according to the IEEE Standard 754 also use bit widths such as 64 and 128 bit. The decimal value of the floating-point numbers are calculated as in Equation 2.3:

$$decimal_value = (-1)^{sign} * b^{exponent-exponent_bias} * (1 + significand) \tag{2.3}$$

The distinction between fixed and floating point numbers is that the fixed point has, as the name indicates, the decimal point at a fixed bit. To avoid erroneous calculations the user has to keep track of its location. The decimal point can be placed where desired, depending on the required size of the integer and accuracy of the fractional part. For example as illustrated in Figure 2.8: one bit is used for sign, 10 for the integer number and

21 for the fractional part. This means that the accuracy of numbers in this representation is $2^{-21} = 476.83 \times 10^{-9}$, and the integer part can be up to $2^{10} - 1 = 1023$.

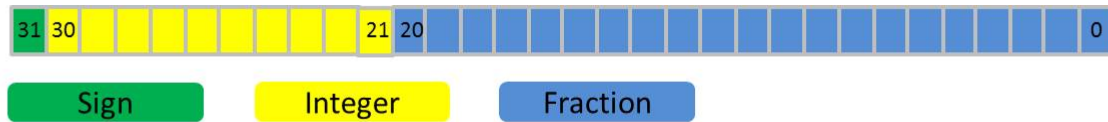

FIGURE 2.8: How fixed point numbers are represented.

Conversion from the fixed point representation as presented in Figure 2.8 to the 32 bit floating point standard can be carried out in the following way:

```
1. Sign bit: no conversion, directly transfer.
2. Find number of bits required to represent the integer of the fixed point ↩
representation.
3. Add exponent base to find correct exponent in the floating point standard.
4. Shift fraction equal to number of bits so that the most significant bit of the ↩
integer is at index 21.
5. Remove most significant bit of integer.
```

LISTING 2.1: Conversion strategy floating point to fixed point

Doing operations on floating-point numbers is not straight forward. Jidan Al-Eryan implemented an FPU and documented it in an accompanying paper. He there added the algorithms for addition, subtraction, multiplication and division[6]. The addition and subtraction algorithm is depicted in Figure 2.9.

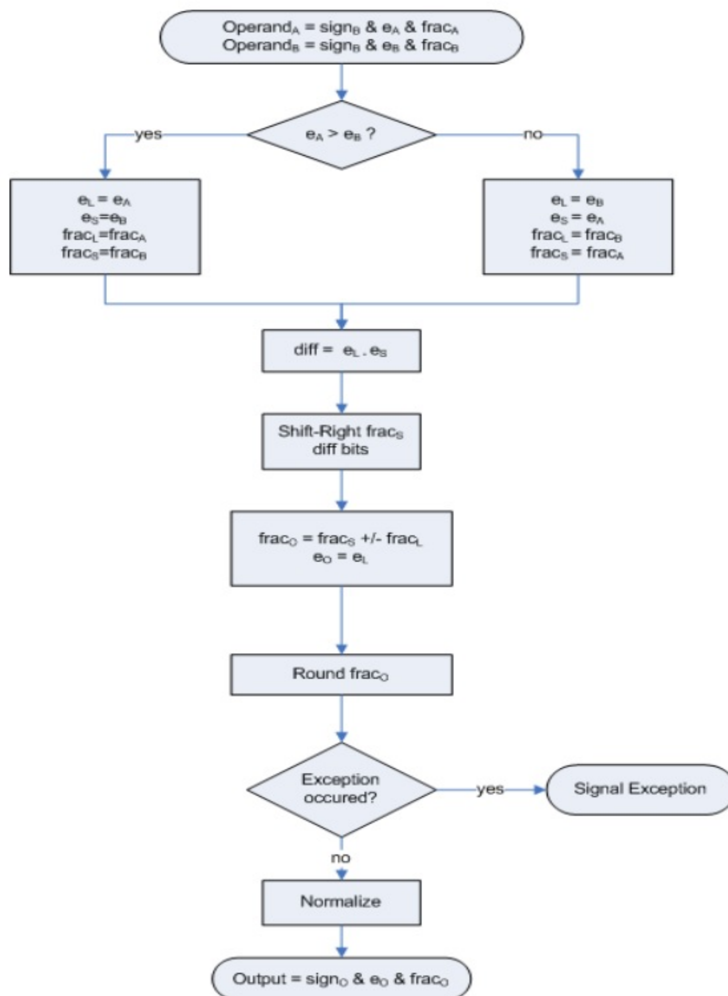

FIGURE 2.9: Algorithm for addition/subtraction in FPU[6].

Due to this, implementation of modules for addition/subtraction and multiplication would be necessary to directly work on the numbers sent from the CPU to the accelerator.

However, in VHDL there exists extension packages which include fixed point and floating point number-types, conversion from floating to fixed-point representations and vice versa[19][20][21]. These packages are:

- *float_pkg_c.vhdl*
- *fixed_pkg_c.vhdl*
- *fixed_float_types_c.vhdl*

David Bishop states in "Floating point package user's guide", that the hardware of the fixed point can be almost 3X less than that of floating point[20]. He also says in the

fixed point guide, that because the fixed point is a step between integer math and floating point, it is almost as fast as the numeric_std arithmetic of VHDL[21].

2.6 Lyapunov Exponent

Lyapunov exponents can be used to diagnose the dynamics in a chaotic system [7]. They quantify the average rate of separation for nearby trajectories[22]. For a system to be chaotic, it needs to contain at least one positive Lyapunov exponent. Lyapunov exponents measure the rate of creation or destruction of information in system processes, and are expressed in bits/s or bits/orbit[7].

The seizure algorithm uses short-term maximum Lyapunov exponents(STL_{max}) which are more accurate than the traditional maximum Lyapunov exponents(L_{max}). STL_{max} profiles are created from recordings of EEG signals from different sites for 10 seconds, each recording site resulting in one STL_{max} estimation[2]. Progressive convergence of STL_{max} profiles is called dynamical entrainment [1][2][9]. The brain sites involved in measuring this dynamical entrainment are called critical sites, while the corresponding pairs of sites interacting are called critical pairs[2].

The selection of sites is however not straight forward, as the sites participating in the pre-ictal transition can vary from seizure to seizure even within the same patient. Iasemidis *et al.* [9] resolved this by selecting k critical sites after the occurrence of the first seizure. Their selection procedure leads to the selection of the k sites that were most entrained 10 minutes prior to the seizure as well as being disentrained postically. This procedure is then carried out after each seizure, and new critical sites are updated automatically for optimal prediction[9].

When it comes to determining entrainment amongst the detected critical electrode sites, the authors use a period of time in which the difference of the mean STL_{max} values of two sites are statistically estimated. They use 10 minute periods[1], which include approximately 60 STL_{max} values, and do the test at significance level $\alpha = 0.01$[9]. As the entrainment is calculated by comparing two sites, the pair T-statistic is used. The T-index for electrode sites i and j in the epileptic seizure prediction algorithm is calculated as in Equations 2.4 to 2.8[1]:

[1]denoted $w(t)$

$$L_i^t = \left\{ STL_{max}{}_i^t, STL_{max}{}_i^{t+1}, ..., STL_{max}{}_i^{t+59} \right\} \tag{2.4}$$

$$L_j^t = \left\{ STL_{max}{}_j^t, STL_{max}{}_j^{t+1}, ..., STL_{max}{}_j^{t+59} \right\} \tag{2.5}$$

$$D_{ij}^t = L_i^t - L_j^t = \left\{ d_{ij}^t, d_{ij}^{t+1}, ..., d_{ij}^{t+59} \right\} \tag{2.6}$$

$$d_{ij}^{t'} = \left\{ STL_{max}{}_i^{t'} - STL_{max}{}_j^{t'} \right\} \tag{2.7}$$

L_i and L_j are vectors of the 60 STL_{max} values for the 10 minute period, and D is the distance between the electrode sites i and$j$ at a time t. This results in Equation 2.8, the pair T-statistic:

$$T_{ij}^t = \frac{|\overline{D}_{ij}^t|}{\frac{\hat{\sigma}_d}{\sqrt{60}}} \tag{2.8}$$

The pair-T statistic shown in Equation 2.8 use $\overline{D}^t{}_{ij}$ and $\hat{\sigma_d}$ which are the sample average and standard deviation respectively. Significance level 0.01 means that the threshold value for disentrainment is $T_{i,j} > T_{\frac{\alpha}{2},59} = T_{0.005,59} = 2.662$. The choice of significance level is explained in Iasemidis *et al.* [1, p. 619].

The threshold value of $T = 2.622$ indicates, as mentioned, disentrainment of the sites. A second threshold has been introduced at significance level $\alpha = 0.00001$, this corresponds to a T-value of 5.000. Thus, there are two threshold values, $T_1 = 5.000$ and $T_2 = 2.662$. A dynamical transition toward a seizure is recognized when the T-value transits from a value $T > T_1$ at time t_1 to a value $T < T_2$ at a time $t_2 > t_1$. If this transition results from the initially selected sites, a warning of an impending seizure will be issued[9].

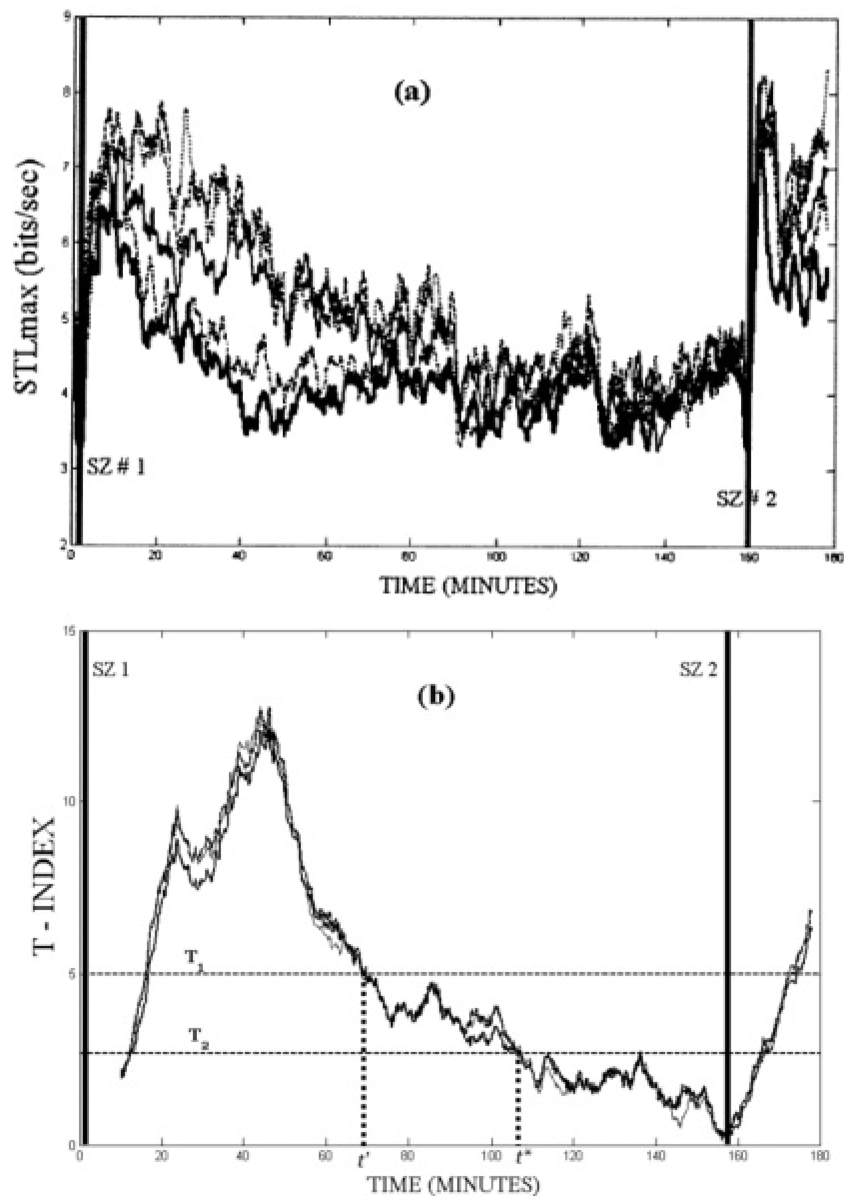

FIGURE 2.10: (a)STL_{max} values between two seizures. (b) T-index for the three most optimal groups of sites between two seizures[1].

Figure 2.10 shows the T-index and STL_{max} between two seizures. The two solid vertical lines are seizure onset while the dashed lines indicate where the T-index crosses the two threshold values T_1 and T_2. As Figure 2.10 depicts, the warning of an impending seizure will for these electrode sites and this seizure be issued about 55 minutes prior to the seizure.

The calculation of Lyapunov exponents in the Epileptic Seizure Prediction algorithm is carried out by expanding the EEG-signals into a m-dimensional[2] phase portrait with

[2] In this case 7-dimensional.

delay coordinates[7]:

$$\{x(t), x(t+T), ..., x(t+[m-1]T)\} \tag{2.9}$$

where T is the delay time. At a time t_0, the nearest neighbour[3] to the initial point is located, and the distance $x(t_0) - x(t)$ is denoted $L(t_0)$. After a constant propagation time, at time t_1, this distance will have evolved to $L'(t_1)$. At this point in time, a point to replace the evolved point is searched for to better meet the following criteria[7]:

- The distance between the new chosen point and the evolved fiducial point is small
- The angular separation between them is small

This procedure is then executed throughout the provided data, and at last the Lyapunov exponent is estimated by Equation 2.10[7]:

$$\lambda_1 = \frac{1}{t_M - t_0} \sum_{k=1}^{M} log_2 \frac{L'(t_k)}{L(t_{k-1})} \tag{2.10}$$

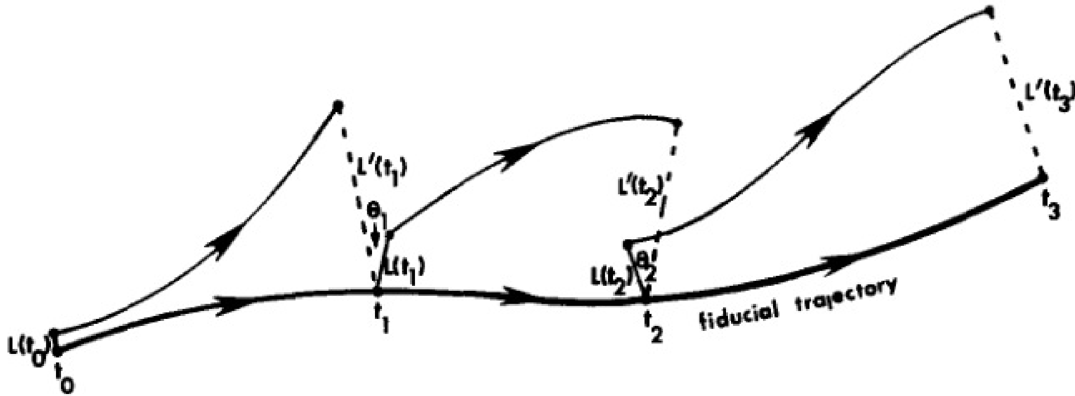

FIGURE 2.11: Evolution of points along the fiducial trajectory for Lyapunov exponent calculation[7].

Figure 2.11 illustrates how the procedure of estimating Lyapunov exponents works, traversing the fiducial trajectory and choosing new points nearer the trajectory if the absolute value of the vector between two points becomes too large. It is then essential that the replacement point minimize both the distance between the new point and the fiducial trajectory as well as the angular separation, denoted θ in Figure 2.11.

[3]Euclidean

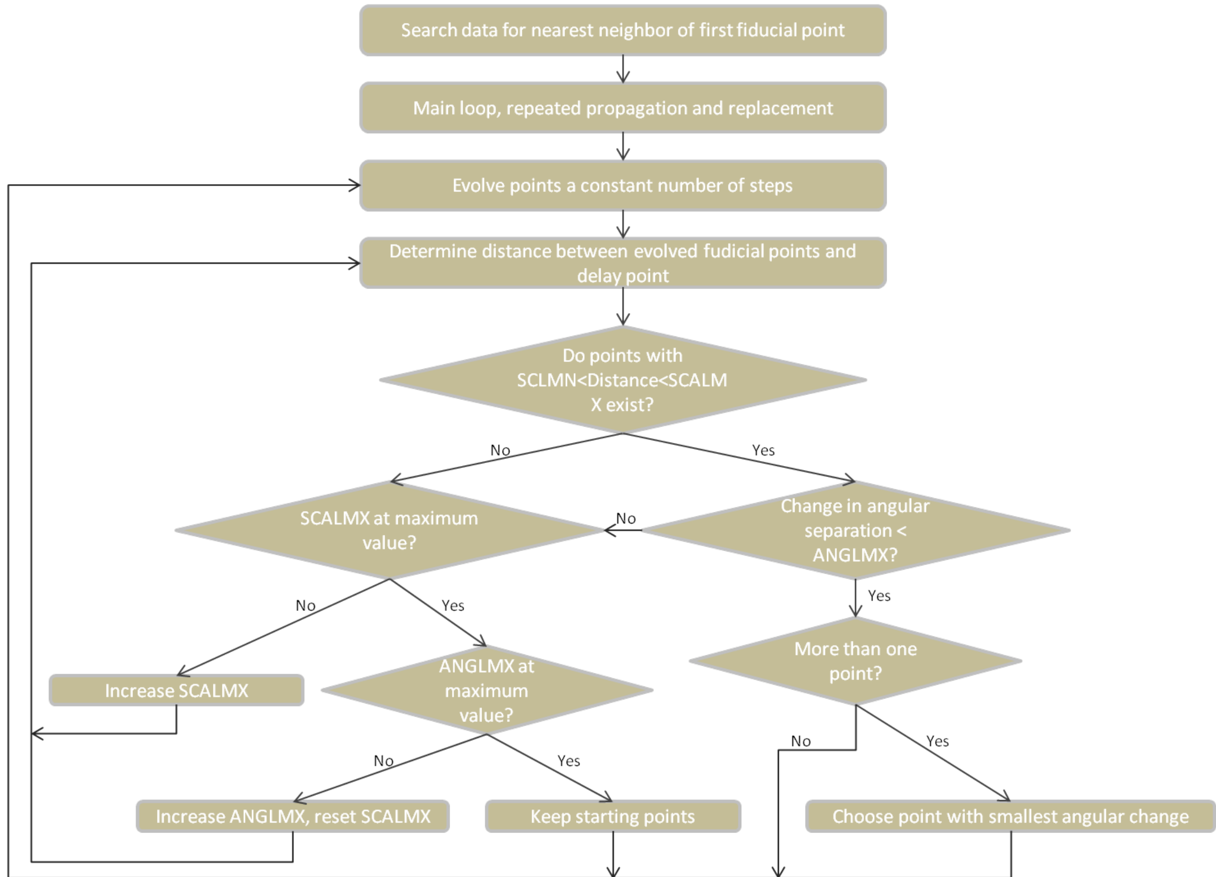

FIGURE 2.12: Flowchart of algorithm for calculating Lyapunov exponents using fixed evolution time as described by Wolf *et al.*[7].

Figure 2.12 describes the algorithm for calculating Lyapunov exponents using the fixed evolution time program proposed by Wolf *et al.* [7]. As can be seen in Figure 2.12, the program contains a main loop where all the data is traversed. Inside the main loop, the distances between the evolved points are calculated. The goal is to find points as little separated as possible down to a lower threshold SCALMN[4]. The upper threshold for separation of points is SCALMX, and for angular separation, ANGLMX. All distances within the interval $\langle SCALMN, SCALMX \rangle$ are examined and the new point is determined based on the angular separation. If no points in the interval exist, or the points within the interval have a too large angular separation, the upper threshold SCALMX is increased. This can be repeated a predetermined number of times. When SCALMX has been increased the maximum amount of times, and no points have been found to fit the criteria, ANGLMX is increased and SCALMX is reset to its initial value.

[4]Lower threshold SCALMN is introduced due to noise considerations.

This procedure is repeated until ANGLMX and SCALMX are both at their maximum values. If no points satisfy the criteria, the initial evolved point of this time step[5] is kept. The exponent is then updated by adding the new points in the sum in Equation 2.10.

[5]e.g. t_1 in Figure 2.11

Chapter 3

Analysis of the Lyapunov Exponent Calculation Algorithm

There are several versions of the Lyapunov Exponent Calculation Algorithm. The initial ones are written in Matlab. Using Matlab is beneficial as matrices and matrix operations are heavily featured in the algorithm. From these initial versions, others have been developed by translating the codes to C-language. This evolution is illustrated in Figure 3.1.

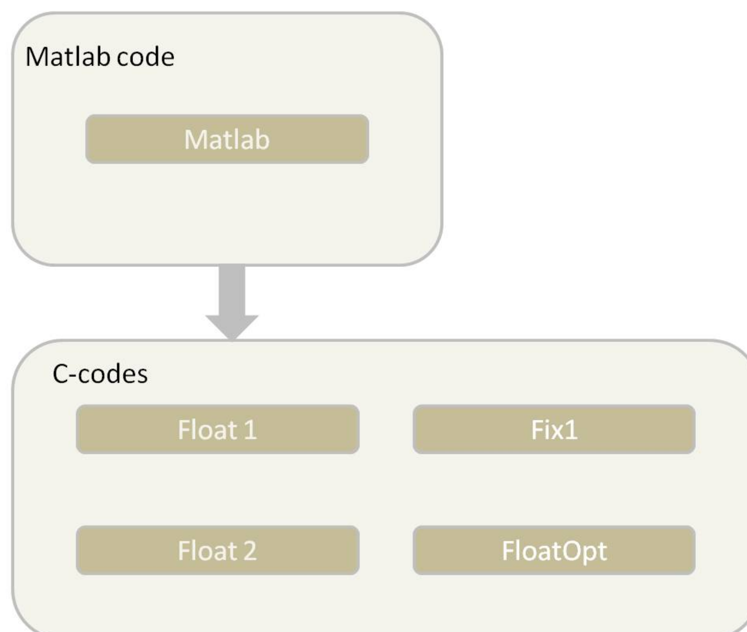

FIGURE 3.1: Illustration of codes available for profiling.

To determine which blocks of the code that are the most computational demanding and thus good targets for acceleration, the codes need to be profiled. As the objective of this study is to map the algorithm to the SHMAC platform, the profiling is divided in two. First the Matlab code is profiled using the built-in tools of Matlab version R2013b. The data attained from the general profiling are then used as a pointer for the system specific profiling. This is done due to a necessity of narrowing down blocks to record time consumption of, when run on the SHMAC platform.

3.1 General profiling

The run time from the profiling in Matlab is shown in Table 3.1. This code uses a C-similar syntax instead of exploiting the matrix operations available in Matlab. The datum in the first row of Table 3.1 is for calculating 3516 Lyapunov exponents. To calculate one exponent it thus takes an average of $\frac{4975{,}14}{3516} = 1.42$ seconds, when run on a computer with 8GB RAM and 3.20GHz processor.

Profiling in Matlab	
Code	Time
For calculating 3516 exponents	4975.142 seconds
For calculating one exponent	1.42 seconds

TABLE 3.1: Profiling of Matlab code working on matrices.

While looking into the profiling data, six lines stood out as the absolutely most run lines. This is illustrated in Table 3.2, where more time is spent on the top six lines than on the seventh. To make this profiling data more comparable with the system specific profiling, the data in Table 3.2 have been divided by 3516 to estimate run times per exponent. This is because the system specific profiling was run for calculation of one exponent. The raw data can be found in Appendix A. The information in Table 3.2 indicate that the vast majority of the execution time is spent on calculating $d1partial$, which corresponds to the code in Listing 3.1.

```
for k=1:7
    for n=1:2008
        d1partial=curr_xV(k)-xV(n+(k-1)*4);
        d1partial=d1partial*d1partial;
        d1(1,n) = d1(1,n)+d1partial;
    end
end
```

Listing 3.1: Snippet of C-similar Matlab code

Most run lines from Matlab profiling			
Line number	Code	Time in msec	% of total time
156 157 158	$d1partial = curr_xV(k) - xV(n + (k-1) * tau);$ $d1partial = d1partial * d1partial;$ $d1(1,n) = d1(1,n) + d1partial;$	742.19	52.35
334 335	$select = cv(n) + (k-1) * tau;$ $dot2partial(n) = dot2partial(n) + (diffpoint(k) * xV(select));$	167.55	11.81
342	$dot2(i) = (dot2(i) * dot2(i))/(d1(cv(i)) * df);$	33.51	2.36
361	$sub = fudpoint - xV(ind : tau : ind + xVexpanse)';$	5.55	0.39

Table 3.2: Most run lines from Matlab profiling

The general profiling indicates that over 50% of the total execution time is spent in the code blocks described by row one in Table 3.2. As seen in Listing 3.1, these three code lines are inside a double nested for-loop. This indicates that is it possible to use the method of Stitt and Vahid (ref. Chapter 2 Section 2.4) and extract the parallelism of the C-code in the possible implementation of an accelerator.

3.2 System specific profiling

For the system specific profiling, four different versions of the Lyapunov Exponent Calculation algorithm were profiled. They were all based on the initial Matlab code as illustrated in Figure 3.1. The codes listed in the tables in this section are as follows:

- Float1: First translation from Matlab to C. Uses floating point.
- Float2: First revision of Float1. Uses floating point.
- Fix1: First translation to fixed point from floating point.

- FloatOpt: Floating point version with optimizations based on experiences from fixed point version, includes loop unrolling.

To determine which version of the algorithm to use for further reference, the execution times while running on the SHMAC platform were measured. This is important data for comparison with timing data after the implementation of an accelerator, as a goal is to run the algorithm within a time schedule of having 10.24 seconds to calculate 32 Lyapunov exponents, i.e. $\frac{10.24}{32} = 0.32$ seconds per exponent. The total execution times per exponent are listed in Table 3.3.

Comparison of execution time on the SHMAC platform				
Code	Float1	Float2	Fix1	FloatOpt
Time in seconds	71.66	69.57	7935.48	52.46

TABLE 3.3: Total execution time

As can be seen from Table 3.3, the fixed point version of the algorithm spends a tremendous amount of time compared to the floating point versions. The algorithm for fixed point is more complicated than the floating point algorithms, as displayed in Table B.4. This is because the Fix1 version emulates fixed point by using floating point, which again is emulated in software on the SHMAC platform. However, if the Fix1 code is rewritten to utilize the integer compatible hardware on the SHMAC platform, the emulation of fixed point via floating point would be not be necessary. Håkon Wikene at the Department of Computer and Information Science, highlighted that the amount of cycles required for integer instructions are a lot lower than those required for floating point instructions. The differences in cycles are presented in Table 3.4.

Change in cycles [float/integer]			
	Add/sub	Multiplication	Division
Emulating cycles/integer cycles	1000/17	3300/17	2500/17

TABLE 3.4: Cycles for instructions in Fix1 code if rewritten for integer utilization. Addition and subtraction spend same amount of cycles.

This would presumably lead to a reduction in run time for the Fix1 code on the SHMAC platform.

Comparison of calculation times				
Code	Float1	Float2	Fix1	FloatOpt
d1partial1	223.27	216.18	37,870.46	156.91
d1partial2	37,686.96	36,488.21	6,393,424.96	26,464.31
dot2partial	14,535.65	14,057.00	918,770.47	14,475.42
dot2	2777.12	2777.06	504,950.10	20.63

TABLE 3.5: System specific profiling, calculation times for different code versions in milliseconds.

Table 3.5 compares the time it takes to run the calculations that stood out in the general profiling on the SHMAC platform. Based on the numbers in Table 3.3 and Table 3.5, the percentage of the total execution time spent in each of the calculations can be found. This further helps to illustrate where most of the execution time is spent, as shown in Table 3.6.

Comparison of percentage of runtimes				
Code	Float1	Float2	Fix1	FloatOpt
d1partial1	0.31%	0.31%	0.48%	0.30%
d1partial2	52.59%	52.45%	80.57%	50.45%
dot2partial	20.28%	20.21%	11.58%	27.60%
dot2	3.88%	3.99%	6.36%	0.04%

TABLE 3.6: System specific profiling, percentage of total runtime spent on calculations.

The calculations of *d1partial* in codes Float1 and Float2 are carried out as in Listing 3.2.

```
for (k=0; k < m; k++)
{
    for (i=0; i < d1size; i++)
    {
        d1partial=curr_xV[k]-xV[i+k*tau];
        d1partial=d1partial*d1partial;
        d1[i] = d1[i]+d1partial;
    }
}
```

LISTING 3.2: Snippet of C code calculation of *d1partial*

After analysis of the percentage of total time spent on the different blocks (ref. Table 3.6), two of the calculations stood out: $d1partial2$ and $dot2partial$, $d1partial2$ is more than twice as time consuming as $dot2partial$. Further, the codes for $d1partial1$ and $d1partial2$ are basically the same, only differing in starting value in their for-loops[1]. $d1partial1$ is the calculation of the first point, while $d1partial2$ is of the remaining 167. The time spent on calculating $d1partial2$ is therefore about 167 times greater than the time spent on calculating $d1partial1$.

Based on the data evolved from the analysis in this chapter and the aim of making the calculation of Lyapunov exponents faster than the previously mentioned time frame on the SHMAC platform, the timing results from the FloatOpt version is chosen as a reference for results after implementation of an accelerator. The FloatOpt version spends 16.8 seconds less time than the second fastest code. Without altering the SHMAC platform it takes minimum 52.46 seconds to calculate one Lyapunov exponent. It is necessary to decrease the run time for it to be applicable in the seizure prediction.

Due to the massive percentage of total execution time spent on calculating $d1partial$, this will be the first target for acceleration in the effort to speed up the code, with $dot2partial$ as a possible extension.

The calculation of $d1partial$ in FloatOpt differs from the others in that instead of having a double-nested for-loop, the outer loop, running from $k = 0 : 7$, is unrolled. This loop unrolling is shown in Listing 3.3. Loop unrolling has also been exploited in the calculation of $dot2partial$ in FloatOpt, illustrated in Listing 3.4. The two code excerpts in Listings 3.3 and 3.4 do look somewhat similar, enabling the opportunity to exploit this similarity to speed up the code even further than with just looking at $d1partial$. The calculation of $dot2partial$ is, however, in a for-loop which varies in size, making it more challenging to implement.

[1]Iterating from $i = th : d1size$ and $i = 0 : d1size$, respectively.

```
for (i = 0; i < d1size; i++ )
    {
        float d1temp = 0;
        //Complete loop unroll of k-loop
        d1partial=curr_xV[0]-xV[i+0*tau];
        d1temp += d1partial*d1partial;
        d1partial=curr_xV[1]-xV[i+1*tau];
        d1temp += d1partial*d1partial;
        d1partial=curr_xV[2]-xV[i+2*tau];
        d1temp += d1partial*d1partial;
        d1partial=curr_xV[3]-xV[i+3*tau];
        d1temp += d1partial*d1partial;
        d1partial=curr_xV[4]-xV[i+4*tau];
        d1temp += d1partial*d1partial;
        d1partial=curr_xV[5]-xV[i+5*tau];
        d1temp += d1partial*d1partial;
        d1partial=curr_xV[6]-xV[i+6*tau];
        d1temp += d1partial*d1partial;
        d1[i] = d1temp;
    }
```

LISTING 3.3: Snippet of calculation of *d1partial2* in FloatOpt

```
select=cv[i];
dot2partial=dot2partial+(diffpoint[0]*xV[select]);
select=cv[i]+tau;
dot2partial=dot2partial+(diffpoint[1]*xV[select]);
select=cv[i]+2*tau;
dot2partial=dot2partial+(diffpoint[2]*xV[select]);
select=cv[i]+3*tau;
dot2partial=dot2partial+(diffpoint[3]*xV[select]);
select=cv[i]+4*tau;
dot2partial=dot2partial+(diffpoint[4]*xV[select]);
select=cv[i]+5*tau;
dot2partial=dot2partial+(diffpoint[5]*xV[select]);
select=cv[i]+6*tau;
dot2partial=dot2partial+(diffpoint[6]*xV[select]);
```

LISTING 3.4: Snippet of calculation of *dot2partial2* in FloatOpt

This loop unrolling, extracting the parallelism of the C-code, is somewhat the same idea Stitt and Vahid exploited in their *Hardware/software partitioning of software binaries*. The replacement of software with hardware has, however, not yet been carried out in this case, and is therefore a natural next step in the process of speeding up the calculation.

Chapter 4

Implementation and Verification

The first step in discovering the benefits of implementing *d1partial* is to design the hardware module. One way to implement the *d1partial* is to make a multiply-accumulate module which is likely to enhance the run time of this calculation, especially if one utilizes pre-fetching of the data by having an own module for address calculation. However, due to the nature of the intended target, another approach is to unroll the nested for-loops. This would probably, contrary to a pure multiply-accumulate acceleration, lead to less cycles required to run the accelerator. An example of unrolling of the *d1partial*-calculation is shown in the FloatOpt code in Listing 3.3. The target for implementation takes, as displayed in Table 3.6, 50.45% of the execution time of the FloatOpt code. As stated in Chapter 3, a Lyapunov point is based on 32 simultaneous Lyapunov exponents, and the Lyapunov exponent is calculated based on a 10.24 second time window. This, together with the percentage of total execution time, means that the accelerator should spend less than $0.50 \times \frac{10.24}{32} = 0.16$ seconds.

4.1 Design

Initially, when looking at the target, two options for the design were clear. The C-codes which are going to work with the accelerator, operates on numbers represented in the floating point standard. Therefore, making an accelerator which takes in floating point numbers and operates on these was an option. Another option was conversion to fixed point and then operate directly on these. However, two other options appeared while learning about the fixed point package[19, p.283] and floating point package[19,

p.284] of VHDL. This means that there are four different approaches to implementing the accelerator:

- Including separate modules for addition/subtraction and multiplication in the design using standard VHDL, floating point.
- Converting floating point input to std_logic_vectors and make logic from standard VHDL to assure the location of the decimal point, fixed point.
- Use functions and type included in the floating point package, floating point.
- Use functions and type included in the fixed point package. This package includes conversion from floating point to fixed. Fixed point.

The first option concerns the implementation of modules for doing floating point operations and directly operating on the data from the CPU using standard VHDL. This approach will however not be implemented, as it basically is like adding an FPU with little functionality to the design, and a full function FPU is currently being worked on by others on the SHMAC project. The second is about conversion from floating point to fixed point and uses the standard package of VHDL to implement the code and the necessary logic to keep track of the location of the decimal point. The third and fourth are somewhat similar in the description, as they both use expansion packages to VHDL, which includes types *float*, *sfixed* and *ufixed* and arithmetic operators for these types. Hence, the third and fourth approaches are to use these packages to make a floating-point and fixed-point implementation of the $d1partial$ calculation, respectively.

The general structure of the design will however be somewhat equal for all approaches, and is illustrated in Figure 4.1. The actual implementations of all approaches are attached in Appendix D.

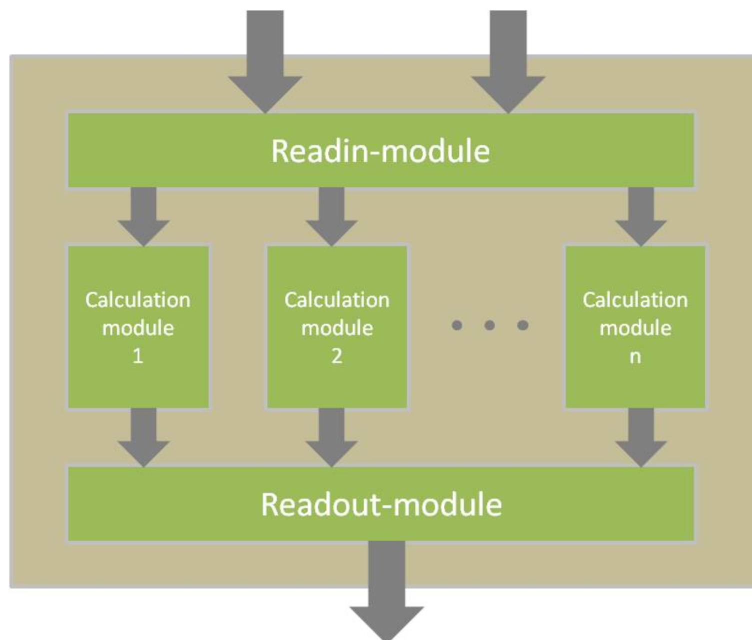

FIGURE 4.1: Illustration of general module.

4.1.1 Approach expansion package, floating point

This approach uses the expansion package for floating point numbers and operators, such as $+$, $-$, $*$ and $/$. It has two 32-bit floating point input ports and a 32-bit floating point output port. The first design is a straight forward implementation of the C-code without any performance enhancements. After this, the technique used for increasing the performance is loop unrolling, as discussed in Chapter 2 Section 2.3. The unroll factors are for the k-loop either not unrolled or completely unrolled, due to the fact that it iterates over 7 numbers, and 7 is a prime number. Therefore, unrolling factor of 7 or none are the two available options whilst avoiding additional control for other unroll factors. For the i-loop, the factors are 2,4 or 8. Next possible unroll factor for the i-loop without making additional control logic is 251, as the number 2008 can be factorized to $2 * 2 * 2 * 251$ and 251 is a prime number.

Table 4.1 displays the synthesis data for the different designs. As exposed by the data, the designs utilize different unroll factors for k and i. Designs 1 through 8 in Tables 4.1 and 4.2 are as illustrated in Figure 4.1, with a *Readin*-module, one or several *Calculation*-module(s) and a *Readout*-module. Designs 9 and 10 are without inclusion of a *Readout*-module, and values are continuously read out as they are calculated.

Designs: 207,360 Slice LUTs available on Virtex5 model				
Design number	Unroll factor k,i	#Slice LUTs	% of Slice LUTs	Maximum Frequency
1	None,0	9505	4%	15.565MHz
2	None,2	144,056	69%	11.872MHz
3	None,4	268,887	129%	9.682MHz
4	None,8	470,221	226%	9.495MHz
5	Full,0	40,558	19%	5.706MHz
6	Full,2	210,065	101%	5.061MHz
7	Full,4	400,558	193%	4.595MHz
8	Full,8	727,126	350%	4.556MHz
9	None,0 w/o RO	6498	3%	15.617MHz
10	Full,0 w/o RO	36,400	17%	6.341MHz

TABLE 4.1: Area and performance for float extension package approach

Design 1 is not at all unrolled, each of the $d1[i]$-s are calculated at separate clock cycles. This results in the hardware having just one calculation-module. The designs of Table 4.1 iterates first over i and then increments k and repeats. This makes the *Readout* module necessary; one needs to iterate through all ks, as the $d1[i]$ calculation is dependent on each of the ks from 0 to 6. To save hardware area, this can be implemented differently, iterating firstly over k and then increment i. This means, that every seventh clock cycle, a value of the $d1$-vector is ready to be output. These can then be read into a vector in the C-code or to memory, and eliminate the necessity of saving the $d1$-vector until all 2008 calculations are done, which saves both area and cycles. Removing the *Readout* module does, however, eliminate the opportunity to have other unroll factors for i than 0, without implementing multiple output ports. There are therefore only two designs without *Readout* module, no unrolling, and full unrolling of k. Full unrolling of k will result in a $d1$-value on the output every clock cycle, while no unrolling will result in a value every seventh clock cycle.

The maximum frequencies of the floating point designs are low, indicating long critical paths. This can be explained by David Bishop's statement: "the hardware of the floating point can be up to 3× more than that of the fixed point package"[20].

4.1.2 Approach expansion package, fixed point

The fixed point approach uses the expansion package for fixed point numbers and operators, such as +, −, ∗ and /. It has two 32-bit floating point input ports and a 32-bit floating point output port, which makes it necessary to convert from floating point to fixed point and vice versa. This has also been included in the package, thus conversion to/from float/fixed is not a design issue. The designs 1 through 10 are otherwise carried out in the same way as those of the floating point approaches.

Designs: 207,360 Slice LUTs available on Virtex5 model				
Design number	Unroll factor k,i	#Slice LUTs	% of Slice LUTs	Maximum Frequency
1	None,0	5450	2%	29.485MHz
2	None,2	141,143	68%	28.091MHz
3	None,4	263,119	126%	25.815MHz
4	None,8	452,204	218%	25.308MHz
5	Full,0	15,920	7%	21.261MHz
6	Full,2	163,578	78%	20.183MHz
7	Full,4	309,559	149%	19.096MHz
8	Full,8	553,642	266%	18.653MHz
9	None,0 w/o RO	3035	1%	29.296MHz
10	Full,0 w/o RO	13,420	6%	22.407MHz

Table 4.2: Area and performance for fixed extension package approach

As can be seen in Table 4.2, the maximum frequencies of the fixed point implementations are significantly higher than that of the floating point implementations. It is also clear that the areas of the fixed point designs are smaller, in line with David Bishops statement, even though these need extra logic for conversion to/from floating point and resizing of vectors within the module. This is needed because operators included in the package result in longer result vectors than the operand vectors.

4.1.3 Approach standard VHDL, fixed point

The approach using standard VHDL has a more intricate calculation-module than those using the predefined packages. VHDL does not support operators such as +, −, ∗ and / for *std_logic_vector*-types, and not using packages with fixed point types means that one

must implement the fixed point type and keep track of the decimal point manually. The Amber core uses 32 bit floating point numbers as described in Chapter 2 Section 2.5. Conversion to and from floating point is due to this needed on the input and output of the accelerator. This is carried out by using the conversion as described in Listing 2.1, Section 2.5.

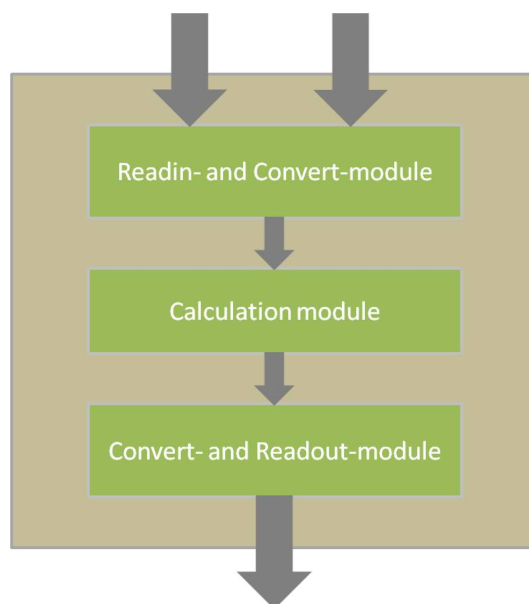

FIGURE 4.2: Illustration of the module using standard VHDL.

The additional logic for correct calculation using the fixed point representation and conversion on inputs and outputs increases the area consumption of the module compared to the equivalent designs using the floating and fixed point packages.

Designs: 207,360 Slice LUTs available on Virtex5 model				
Design number	Unroll factor k,i	#Slice LUTs	% of Slice LUTs	Maximum Frequency
1	None,0	112,778	54%	44.139MHz

TABLE 4.3: Area and performance for standard VHDL fixed point approach

An advantage of using the standard VHDL is that you have more control of the design. Using the packages may include additions which may consume additional space and make critical paths longer than necessary. The design shown in Table 4.3 is synthesised without utilizing the available RAM resources. This results in the synthesiser generating one FlipFlop for each bit in the arrays, which is very inefficient compared to synthesising arrays as RAM, and causes tremendous area consumption. In this case, a total of

$32 \times 7 = 224$ FlipFlops are generated for saving the curr_xV values, and $32 \times 2048 = 65,536$ FlipFlops for the xV values. Still, the frequency of this design is almost 50% higher than the design with the highest frequency amongst the package utilizing designs, indicating that using standard VHDL results in more efficient hardware and shorter critical paths.

4.2 Verification

For the standard VHDL-approach, the conversion modules needed local verification. To do this, several numbers with different qualities were applied to the converters and compared to the number's corresponding floating or fixed point representation. The numbers were different cases such as:

- Between 0 and -1
- Between 0 and 1
- Zero
- Larger numbers requiring several bits for integer part

This was, however, done progressively during the design time, and no established test bench was used.

The plan for verification of the entire system is to use the existing EEG recordings attached to the codes, and generate numbers using the software. These numbers will then be used in a golden device-like strategy. To check this only for the code that is to be implemented in hardware, all the intermediate $d1$ values are written to a file. With the attached EEG signals, there is an opportunity to calculate a total of $32 \times 3516 = 112,512$ Lyapunov exponents, as there are EEG signals from 32 channels. This means that it is possible to test the hardware against the software $32 \times 3516 \times 2008 = 225,924,096$ times for each $d1[i]$-value calculated, or the entire $d1$ vector $32 \times 3516 = 112,512$ times. As this would be done manually, running all 21 designs 112,512 times would be very time consuming. Therefore 28 of these 112,512 $d1$-calculations have been chosen from the first seven channels. Within the seven channels, data at four different intervals of the EEG-signals have been chosen, for four calculations within each channel. This spreading is done to hopefully get different characteristics of the EEG signals in the test.

An important part of the Lyapunov exponent calculation is accuracy, as increased accuracy in calculating exponents will contribute to the accuracy in issued seizure warnings. Therefore, an important part of the verification is looking at deviations between values produced by the software code and the hardware module. This will also indicate correctness of the code if the deviations are small, for example less than 0.1. Exactly how accurate the calculation needs to be is not possible to decide without running the entire prediction algorithm with the numbers presented via this hardware and evaluate correctness of issued seizure warnings. Unfortunately the entire code is not accessible.

The deviation arises, as mentioned, due to the fact that implementing a software code in hardware will probably lead to rounding errors due to the limitation in number of bits used to represent the number, as discussed in Section 2.5. Due to this, a direct comparison on whether or not the output from the C-code and the output of the implemented hardware are equal, appears not to be a good solution. To make a good comparison, a code calculating the difference between the different $d1[i]s$ from software and hardware has been written. This technique is illustrated in Figure 4.3. Excerpts of the results from this comparisons as presented in Tables 4.4, 4.5 and 4.6 show the deviations from seven file pairs from each channel. Which exponent the deviation is calculated for is decided by choosing file pairs which are $\frac{3516}{7} \simeq 502$ file pairs apart for even distribution.

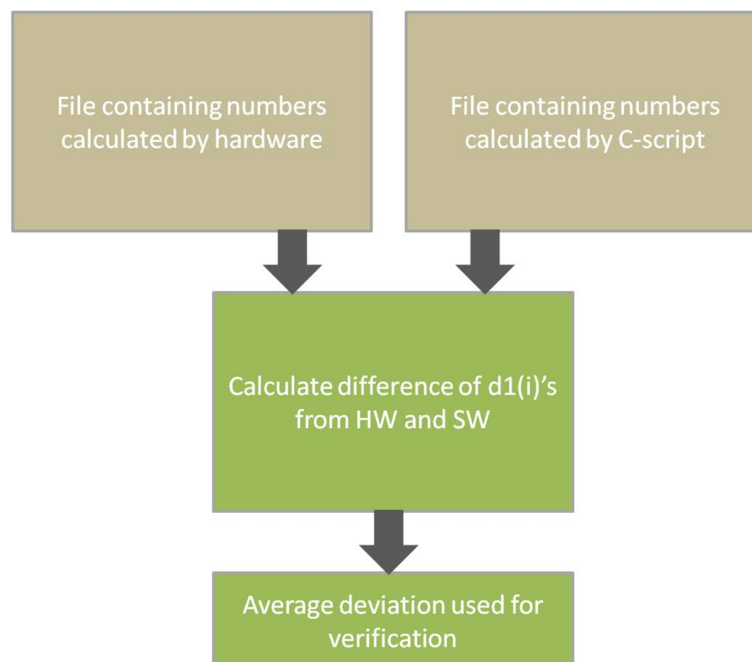

FIGURE 4.3: Illustration of the technique used for verification.

The actions described in the green boxes of Figure 4.3 are carried out by scripts written in *C++* and *Matlab*. The source codes are attached in Appendix E.

Verification of module utilizing the floating point expansion package

Table 4.4 is an excerpt of absolute mean deviations. The average mean value at the bottom of Table 4.4 is the average value after calculating mean deviations of 28 randomly selected file pairs. Each file pair contain values for calculating one total $d1$ array of 2008 elements.

Deviation	
Channel,exponent	Absolute mean deviation
1,1	0.000077
2,503	0.000019
3,1005	0.000015
4,1507	0.000066
5,2009	0.000012
6,2511	0.000012
7,3516	0.000014
Average mean value:	0.000022

Table 4.4: Deviation for float expansion package approach

Verification of module utilizing the fixed point expansion package

The average mean deviation after calculating deviations from 28 different datasets, resulting in $28 \times 2008 = 56,224$ $d1$-values, is 0.050554, as displayed in Appendix C Table C.2. This rather large average deviation does, however, not give the full picture of the design's quality. As seen in Appendix C, Tables C.2, two file-pairs, from Channel 4 exponent 3516 and Channel 7 exponent 1, increase the average mean deviation significantly compared to the contribution of the other 26 pairs. Even though this relatively huge deviation originates from only 2 out of 28 file-pairs, it still questions the quality of the design. To try to eliminate this deviation, the calculation of these values was debugged. During that process, everything seemed to work fine, until an addition went wrong and resulted in a larger number than expected. As it was just a simple addition which apparently works fine for the other file pairs, I am not sure how to address this issue.

Deviation	
Channel,exponent	Absolute mean deviation
1,1	0.000075
2,503	0.000019
3,1055	0.000015
4,1507	0.000066
5,2009	0.000012
6,2511	0.000012
7,3516	0.000015
Average mean value:	0.050554

TABLE 4.5: Deviation for fixed expansion package approach

Verification of module using standard VHDL

The verification plan of calculating average deviation for the $d1$-values was also used for the module using the standard VHDL. As for the two other approaches, a total of 28 file-pairs were used for verification. The results of this process is presented in Appendix C Table 4.6. The average mean deviation as seen in the last row of Table 4.6 implies that using the standard VHDL approach and using fixed point numbers in the calculation is on average accurate to 10^{-4}.

Deviation	
Channel,exponent	Absolute mean deviation
1,1	0.000091
2,503	0.000024
3,1005	0.000021
4,1507	0.000073
5,2009	0.000013
6,2511	0.000016
7,3516	0.000016
Average mean value:	0.000027

TABLE 4.6: Deviation for standard VHDL approach

Chapter 5

Results

5.1 Results from Simulation

This section presents results from simulation using *Xilinx's ISim* and synthesis using *Xilinx's XST*. Firstly the general results from the pre-synthesis simulations without delays are presented, and then the specific results which take into account the physical limitations post synthesis. The unroll factor for k is either "none" or "full" as described in Section 4.1.1. The fixed point package designs, floating point package designs, and the standard VHDL design with same unrolling factor, spent exactly the same amount of time calculating the $d1$-vector during simulation. This is due to an ideal environment, i.e. no delays, equal clock frequency and that the designs spend the same amount of cycles for calculation with the same unroll factors. The timing data from the pre-synthesis simulations at 60 MHz are shown in Table 5.1. The columns 4 and 5 present the amount of time spent calculating the $d1$-values and the total run time from the start of the read in until the last value is read out. Please note, however, that there only exists one design, which has unroll factor *None,0*, for the standard VHDL approach.

Timing[us]				
Unroll factor k,i	Read in data	Read out data	Calculation	Total
None,0	34.131968	33.465328	234.257296	301.854592
None,2	34.131968	33.465328	117.128648	184.7259442
None,4	34.131968	33.465328	58.564324	126.161620
None,8	34.131968	33.465328	29.282162	96.879458
Full,0	34.131968	33.465328	33.465328	101.062624
Full,2	34.131968	33.465328	16.732664	84.329960
Full,4	34.131968	33.465328	8.366332	75.963628
Full,8	34.131968	33.465328	4.183166	71.780462
None,0, w/o RO	34.131968	N/A	234.257296	268.389264
Full,0, w/o RO	34.131968	N/A	33.465328	67.597296

TABLE 5.1: Simulation time from float and fixed extension package approach, and the standard VHDL approach

To illustrate the difference between the designs with a *Readout*-module and those without, some waveforms from the simulations are added. The waveforms, as seen in Figure 5.1, compare the total runtime of an accelerator without(left) and with(right) a *Readout*-module. This figure illustrates how the $d1$ values are not stored intermediately in the accelerator, but are read out continuously as they are ready. As shown in Figure 5.1, this happens at the first marker, at time 34.140301 us, in the left-hand waveform.

FIGURE 5.1: Waveform from simulation of a module without(left) and with(right) *Readout*-module.

Figure 5.2 illustrates how a $d1$-value is output every seventh clock cycle when simulating an accelerator without a *Readout*-module and no unrolling of k as described in Chapter 4.

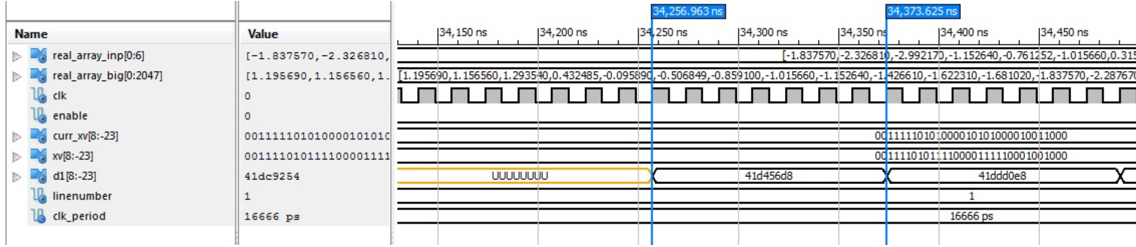

FIGURE 5.2: 7 cycles between outputs for design without *Readout*-module and no unrolling of k.

5.2 Results post Synthesis

Results after the synthesis provide a picture of what limitations the implemented design will have. The actual frequencies the accelerators are able to run on will affect the run times from the simulation without delays. Sections 5.2.1, 5.2.2 and 5.2.3 present estimations of run times and calculation times when using the new maximum frequencies. The new timing was estimated by multiplying the pre-synthesis timing with the factor of reduction in frequency after synthesis, as in Equation 5.1

$$New_time = Old_time \times \frac{60MHz}{New_freq} \tag{5.1}$$

This method has been chosen due to unresolved difficulties with post-synthesis simulation using the float and fixed packages. Due to the fact that the float ports[1] were synthesised to a type *std_logic_vector1* which handles negative indices, these difficulties made post-synthesis simulation unavailable. It lead to a mismatch with the test-bench, which is not synthesised, and still works with the float-type. Within the time-limits of this study it was not time to study the *std_logic_vector1*-type to possibly create working a test-bench with this type.

[1]Both approaches use float-type as ports

5.2.1 Approach expansion package, floating point

The synthesis results for the floating point expansion package approach were mainly presented in Chapter 4 Table 4.1. Table 5.2 presents new timing data estimations from the post-synthesis maximum frequencies.

Estimated timing with new frequencies			
Design	Frequency post synthesis	Calculation time[us]	Total time[us]
1	15.565MHz	903.015597	1163.58982
2	11.872MHz	591.967453	933.587992
3	9.682MHz	362.927023	781.831977
4	9.495MHz	185.037359	612.192468
5	5.706MHz	351.896193	1062.69846
6	5.061MHz	198.371832	999.762418
7	4.595MHz	109.244814	991.908091
8	4.556MHz	55.0899824	945.308982
9	15.617MHz	900.008821	1031.14272
10	6.341MHz	316.656628	639.621158

Table 5.2: New timing for float extension package approach

The first column enumerates the designs from 1 to 10. This is to make the reading more easy than using their unroll factors. The designs are ordered as in Table 5.1. The second column presents the maximum clock frequency of the designs post-synthesis. These are together with the simulation timing data used to estimate new times with the new frequencies, in accordance to Equation 5.1. The resulting new calculation times and run times are presented in Columns 3 and 4.

5.2.2 Approach expansion package, fixed point

Corresponding data for estimated time consumed post-synthesis for the fixed point implementations are presented in Table 5.3. These results suggest that the hardware created using the fixed point package is more efficient than the floating point. Even though the fixed point approach has the resizing and conversion logic mentioned in Chapter 4, the critical paths are smaller than that of the floating point hardware. This results in higher frequencies and significantly lower calculation and total run time.

Estimated timing with new frequencies			
Design	Frequency post synthesis	Calculation time[us]	Total time[us]
1	29.485MHz	476.697906	614.253876
2	28.091MHz	250.176885	394.558992
3	25.815MHz	136.116965	293.228635
4	25.308MHz	69.4219109	229.681029
5	21.261MHz	94.4414505	285.205655
6	20.183MHz	49.742845	250.696011
7	19.096MHz	26.2871764	238.679183
8	18.653MHz	13.4557422	230.891959
9	29.296MHz	479.773271	549.677630
10	22.407MHz	89.6112679	181.007621

TABLE 5.3: New timing for fixed extension package approach

The data in Table 5.3 corresponds to those in Table 5.2 for the fixed point package approach, and they are explained in Section 5.2.1.

5.2.3 Approach standard VHDL, fixed point

The approach using standard VHDL resulted in the largest post-synthesis frequency. As seen in Table 5.4, the total run time increases by $\frac{410-301}{301} \times 100 = 36\%$. Table 5.4 illustrates that the higher post-synthesis frequency of the approach using standard VHDL is about $\frac{44}{15} \simeq 2.9$ and $\frac{44}{29} \simeq 1.5$ times higher than those of the other two comparable designs.

Estimated timing with new frequencies			
Design	Frequency post synthesis	Calculation time[us]	Total time[us]
1	44.139MHz	318.4357996	410.3236485

TABLE 5.4: New timing for standard VHDL approach

The post-synthesis calculation and run times for this approach are also estimated. Because of not using the RAM resources, the placing and routing of this design took too much time[2], so the results from Place and Route is missing. This means that the power

[2]Several days

estimation tool could not be used properly, as it requires the Place and Route to be done. It also lead to that the post-synthesis simulation never launched.

5.2.4 Power estimates

It was initially difficult to get a power consumption estimate for the standard VHDL approach. This is due to the fact that the power analysis tool *XPower Analyzer* needs to run a Place and Route to generate an *.ncd*-file to get good estimates for the power consumption. This was overcome by using the *.ncd*-file from synthesis, which probably will not give as good estimates as the ones generated from Place and Route. However, it is clear from the power estimates shown in Table 5.5 that the estimates of the Standard VHDL approach are a lot larger than those of the package approaches. This is most likely due to the arrays being synthesised as logic instead of memory. The power estimates seems to be proportional to the number of LUTs used. So according to this, the power consumption of the standard VHDL approach would be similar to that of the fixed package approach if the RAM resources were utilized.

Power estimates[mW]				
Approach	Dynamic/static	Logic	Signal	Clocks
Float package	55.24/2951.79	3.90	16.32	32.68
Fixed package	51.56/2951.79	2.90	11.81	34.60
Standard	622.35/2951.79	65.52	116.27	435.82

Table 5.5: Table of power estimates.

Chapter 6

Discussion

All the designs have one thing in common; accessing arrays at different indices(multiple addresses) adds RAMs to design. This implies not unrolling k and unrolling i by a factor of 2 leads to implementation of two RAMs of size 32×2048 and two RAMs of size 32×7, because this means accessing the xV-array and $curr_xV$-array at two different indices in the same clock cycle. This multiple accessing is as shown in Listing 3.3. If all of the accesses shown in Listing 3.3 are carried out within the same clock cycle, the xV-array and $curr_xV$-array are accessed at seven locations simultaneously, causing seven copies of the xV-array and $curr_xV$-array RAMs. Increased unrolling factor increases the number of RAMs implemented in the synthesis leading to increased area.

When looking into area vs performance some designs for both the floating point and fixed point package approaches can be discarded rather quickly. This is due to the fact that they consume a larger area than available on the Virtex5 platform. This upper area limit is displayed as a horizontal red line in Figure 6.1 for the float approach. This line eliminates design 3, 4, 7 and 8. They could be discussed if the FPGA was larger, but additionally, their run times are not low enough to make up for the area consumption, as there are smaller designs which spend less time or almost equal time, such as Design 10. A graphical representation of the area and estimated calculation times and total run times is presented in Figure 6.1.

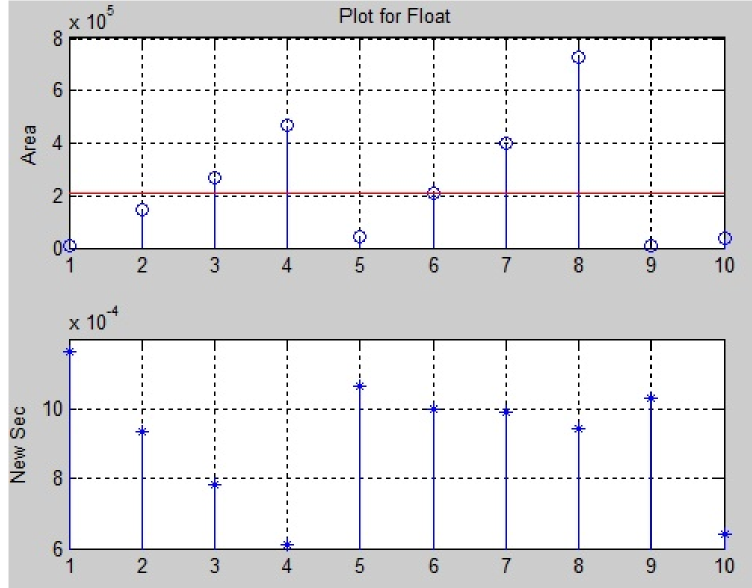

Figure 6.1: Graph showing area and speed(in estimated new total runtime, new sec) of different design choices.

Further it is important to notice that the increased area also leads to prolonged critical paths, which in turn result in reduced frequencies. Figure 6.1 shows how the performance gains are diminished by significantly reduced clock frequencies. Another aspect is that the performance gain of unrolling only applies to the calculation time. The number of cycles required for reading in and reading out are constant across designs, when applicable. This makes the decrease in run times stagnate at the larger unrolling factors, where the read in plus read out times are up to $\frac{33+34}{4} = 16.75$ times larger than the calculation time. This effect especially manifests itself for designs 5, 6, 7 and 8, where the k-loop is fully unrolled. This could be addressed by implementing multiple I/Os. However, one would also then have to implement the same number of conversion modules because having only one conversion for the multiple I/Os would cause the same bottleneck considered.

As designs with both small area and low time consumption are wanted, Design 10 seems to stand out as the most optimal design amongst the designs of the floating point package approach. This is, however, a design without a *Readout*-module, which needs to be taken into consideration when the accelerator shall cooperate with surrounding hardware.

Regarding performance for the fixed point package approach these designs seem to benefit more from the loop unrolling than the corresponding floating point designs. This

is likely due to the less hardware demanding nature of the fixed point logic. The higher maximum frequencies post synthesis also reflect the same. As for the floating point designs, the reduction in run time stagnates at higher unroll factors, as illustrated in Figure 6.2, especially for designs 5 to 8. This is for the fixed point designs, as for the floating point designs, due to the fact that the calculation times are small compared to the read in and read out times and reduced frequency.

The graphical representation of area and run time shown in Figure 6.2 also immediately discards designs 3, 4, 7 and 8 due to area limitations. Generally, the designs using the fixed point logic have more equal run times than the floating point designs. However, the designs using no unrolling, Designs 1 and 9, stand out as particularly time consuming especially compared to Designs 5 and 10, which spend about 400 us less and not significantly more area. These qualities make Designs 5 and 10 the most interesting for further work and implementation to the SHMAC platform. Design 10 does not include the *Readout*-module, while Design 5 does. Apart from this, these two designs seem to have reasonably similar qualities in both area and performance.

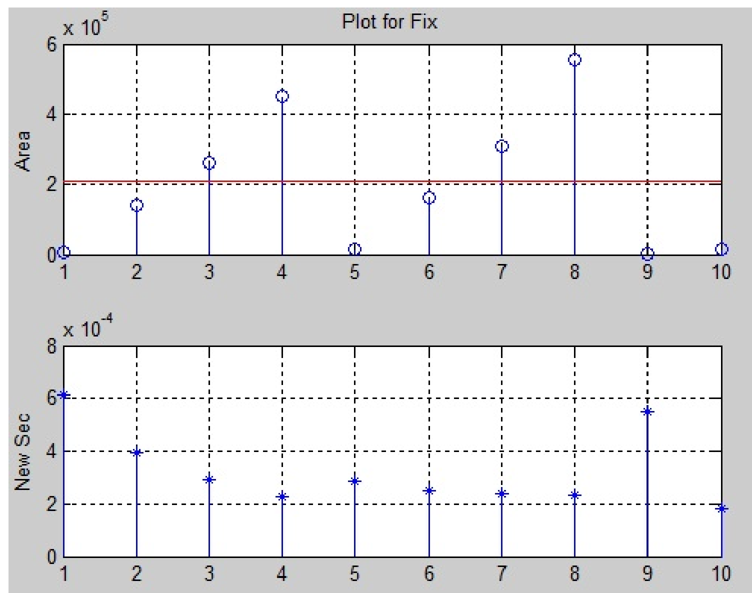

FIGURE 6.2: Graph showing area and speed(in estimated new total runtime, new sec) of different design choices.

The results from not utilizing the packages and implementing fixed point operations from scratch are quite interesting. Firstly, the post-synthesis frequency is significantly higher than for the other two approaches. This frequency makes the approach with

standard VHDL to be most effective. It is faster than all of the floating point package-designs and two of the fixed point package-designs even though only Design 1 presented in Tables 5.2 and 5.3 have the same building structure with no parallel calculation. The standard VHDL approach does, however, occupy a smaller area than the designs which consume less time.

Secondly, the area of this design is rather large compared to the corresponding designs in the two other approaches, i.e. with unroll factors k and i as 0. This is likely due to the trouble of getting the arrays implemented utilizing the RAM resources on the FPGA. Instead the arrays are implemented as one FlipFlop per bit within the array, resulting in a rather large design.

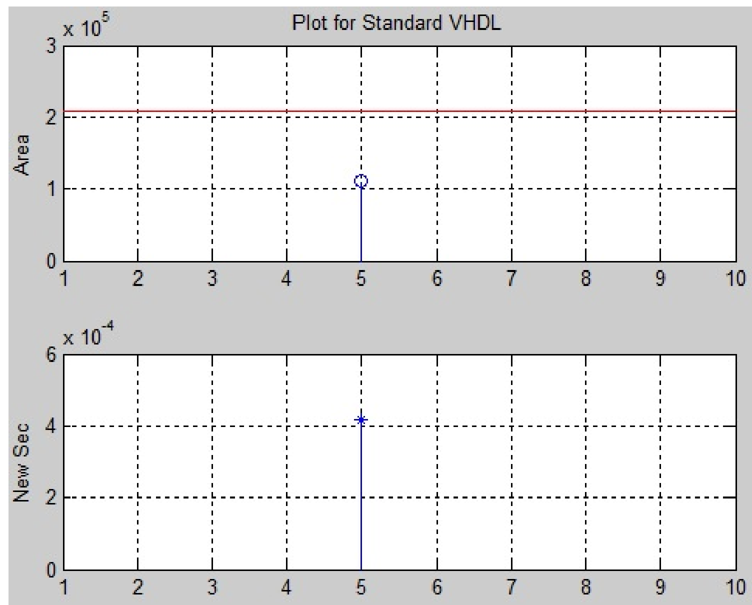

FIGURE 6.3: Graph showing area and speed(in estimated new total runtime, new sec) of the approach implementing fixed point from std_logic_vector.

Extending the standard VHDL design to include loop unrolling would be more demanding than for the other two approaches, as the calculation module of the standard VHDL approach is more intricate than for the other two. One clear-cut way to do this, would be by implementing the entire calculation as a subcomponent. Then the process of designing having different unrolling factors would be to implement the number of subcomponents equivalent to the wanted unroll factor. This would, with the current control logic, be for unrolling of the i-loop. The fully unroll of the k-loop would need seven parallel subcomponents and some alteration of the logic controlling the indexing. As for the other approaches, it is expected that unrolling will lead to increased area, lowered maximum

clock frequency, but also a reduction in clock cycles corresponding to the unroll-factor needed for calculating the $d1$-values.

Comparison

I have, in this section, compared the characteristics of the corresponding designs from the different approaches. Due to the previously mentioned memory synthesis problems with the standard VHDL approach, some estimations have been made to compare the areas. The synthesis of the float and fixed approaches' Design 1 exposed that the three arrays of sizes 32×7, 32×2008 and 32×2048 use a total of 3480 LUTs when the RAM resources are utilized. The RAMs are in this case implemented on LUTs due to currently unsupported block RAM features according to the synthesis tool. If these resources are not utilized, the arrays take up "number of bit" FlipFlops[1]. Based on a synthesis of a design without any of the arrays, it is estimated that the calculation and conversion logic of the standard approach use 1722 LUTs. It is from this estimated that if using the RAM resources for the arrays, the area of the Standard-approach is $1722 + 3480 = 5202$ LUTs.

To make the the power statistics comparable, all are run on the same clock frequency[2].

Comparison qualities of designs						
	Area	Performance	Power estimates[mW]			
Approach	Total/logic	[Max.freq]	Dynamic/static	Logic	Signal	Clocks
Float package	9505/6025	15.565	55.24/2951.79	3.90	16.32	32.68
Fixed package	5450/1970	29.485	51.56/2951.79	2.90	11.81	34.60
Standard	5202/1722	44.129	622.35(51.56)/2951.79	65.52(2.90)	116.27(11.81)	435.82(34.60)

TABLE 6.1: Table which compares qualities of the different approaches.

The data presented in Table 6.1 exposes the standard VHDL approach as the most area efficient and fastest. The numbers in the parenthesis for the standard approach are equal to those estimated by the *XPower Analyzer*-tool for the Fixed package approach. As discussed in Section 5.2.4, these are estimated to be fairly equal to the data for the Standard VHDL approach if it was synthesised with the RAM resources.

Another aspect of the difference between the approaches, is the time spent on designing. For the two package approaches this time is almost equal. The designs of these

[1] One "slice LUT" contains a FlipFlop and a LUT
[2] 12.2MHz to avoid timing conflicts in slowest design

approaches are quite similar, mostly differing in types used and the fact that the fixed point approach needs some extra attention for conversion and resizing to keep the 32-bit width. This is, however, easily done by using predefined functions included in the package. Due to the need for learning the packages, the first design, i.e. Design 1 using the float approach, was time consuming. The remaining 19 could, to a great extent, be based on this first design. Concerning the design time, the standard approach is definitely the most demanding, taking several more days than the other approaches. This is both due to that it was not possible to base anything on the package-designs, and that the fixed representation, as well as the conversion procedure, needed to be handled from scratch.

Furthermore, all the designs satisfy the timing criteria of spending less than $0.50 \times \frac{10.24}{32} = 0.16$ seconds, as the most time consuming design only spend 1.164 milliseconds. This means that there is room for 0.158837 seconds, or $0.158837 \times 60MHz = 9,530,220$ cycles for overhead, communication and data transfer.

Chapter 7

Conclusion and Future work

Conclusion

As *Xilinx* does not seem to be fully compatible with the floating point and fixed point packages, resulting in problems during and after synthesis, the best solution seems to be to keep to standard VHDL when designing for a *Xilinx* platform. Post synthesis simulation has been a major issue, as I have not been able to make it run properly. This also narrows the possibility to run a good power estimation using the built in *XPower Analyzer*, as it makes use of switching data generated during post-synthesis simulation to calculate power usage.

Due to the issues mentioned above, it seems to be more likely to avoid unexpected problems and incompatibility with the surrounding interface when the time comes for implementing the accelerator on the platform. It also seems like from scratch implementation, as it is when not using the packages, generates a smaller circuit than utilizing the packages. However, all the designs meet the timing criteria stated in Chapter 4.

All designs presented in this study are designed to be tested locally, using the previously described verification technique (ref. Chapter 4). It thus follows that these designs are different from how they would be when implemented on the SHMAC platform with a fully functioning framework with the CPU and the interface. Some alterations have to be made, and some of them will presumably reduce the area of the designs quite remarkably. These alterations are considered in the following section.

As for the assignment described in the problem description, the algorithm and its STLmax calculation have been studied. The STLmax calculation was successfully

adapted to run on a general SHMAC processing tile. A kernel was selected and an accelerator was implemented using multiple approaches to evaluate different advantages or disadvantages of the different approaches, these were performance and area, as well as estimating the power consumptions.

Future Work

Regarding future work, there is a lot to be done. Firstly it would be interesting running the algorithm in software when the FPU is implemented on the platform. As one then has support for floating point numbers rather than them being emulated in software as they currently are, this will presumably result in a much lower run time than those presented in Chapter 3.

Another interesting issue is to rewrite the Fix1-code so it utilizes the integer hardware on the SHMAC platform, avoiding the double emulation which is currently the case for the fixed point code, as stated in Section 3.2. In this case, the conversions to and from floating point conversion of the fixed point approach and standard VHDL approach can be removed, leading to less area consumption and probably a higher maximum frequency.

A natural way to proceed with the accelerator of this assignment is to firstly get the standard VHDL design to utilize RAM resources, and then implement designs on the SHMAC platform. The current designs are written to enable local testing by calculating all 2008 $d1$-values using data output from the C-code as described in Chapter 4. To be compatible with the interface designed by Marton Teilgård[23], some changes need to be made. Instead of loading in the entire matrices for calculating all 2008 $d1$ values, it's beneficial to load in for calculating one of the 2008 at a time. If one uses a design which iterates firstly over k, this is possible due to the need of only 2×7 inputs for $d1[i]$, before moving on to $d1[i+1]$. At $d1[i+1]$ one only needs seven new values, as the one matrix(curr_xV in Listing 3.2) of values is equal for all $d1$-calculations of one exponent. Together with some logic for calculating addresses of the data needed for the currently calculated exponent, this alteration should presumably be sufficient to make the accelerator work with Marton Teilgård's interface.

Appendix A

Profiling of C-similar Matlab-code

This appendix contains profiling data of running the algorithm for Lyapunov exponent calculation in Matlab. The code profiled here use syntax close to the one of C++. Note that this code calculates 3516 Lyapunov Exponents, whereas the codes in Appendix B calculate one.

Profiling				
Line number	Code	Total #of Calls	Total Time(s)	% of Total Time
Main				
30	$lmax = lmax_hazelnut(data, 7, 4, 0.005, 12, 24, 20);$	3516	4975,142	99.8%
lmax_hazelnut				
156	$curr_d1partial = curr_xV(k) - xV(n + (k - 1) * tau);$	8,253,289,632	893,554	17,92%
157	$curr_d1partial = curr_d1partial * curr_d1partial;$	8,253,289,632	834,379	16,74%
158	$d1(1, n) = d1(1, n) + curr_d1partial;$	8,253,289,632	881,613	17,68%
334	$select = cv(n) + (k - 1) * tau;$	2,747,136,252	285,832	5,73%
335	$dot2partial(n) = dot2partial(n) + (diffpoint(k) * xV(select));$	2,747,136,252	303,283	6,08%
342	$dot2(i) = (dot2(i) * dot2(i))/(d1(cv(i)) * df);$	392,448,036	117,833	2,36%
361	$sub = fudpoint - xV(ind : tau : ind + xVexpanse)';$	585,811	19,505	0,39%

Table A.1: Profiling of C-syntax like Matlab code

Appendix B

Timing of most time consuming calculations

Timing				
Algorithm version	Line	Calculation code	Time in msec	% of total time
Float 1		**Total execution time:**	71,662.400018	100
	140 141 142	$curr_d1partial = curr_xV[k] - xV[i + k * tau];$ $curr_d1partial = curr_d1partial * curr_d1partial;$ $d1[i] = d1[i] + curr_d1partial;$	223.265181	0.31
	201 202 203	$curr_d1partial = curr_xV[k] - xV[i + k * tau];$ $curr_d1partial = curr_d1partial * curr_d1partial;$ $d1[i] = d1[i] + curr_d1partial;$	37,686.958847	52.59
	393 394	$select = cv[i] + k * tau;$ $dot2partial[i] = dot2partial[i] + (diffpoint[k] * xV[select]);$	14,535.652115	20.28
	406	$dot2[i] = (dot2[i] * dot2[i])/(d1[cv[i]] * df);$	2777.115973	3.88

Table B.1: Timing of most time consuming calculations, algorithm float 1

Timing				
Algorithm version	Line	Calculation code	Time in msec	% of total time
Float 2		**Total execution time:**	69,568.340331	100
First calculation	144, 153	$curr_d1partial = curr_xV[k] - xV[i + k * tau];$		
k=0 in separate	145, 154	$curr_d1partial = curr_d1partial * curr_d1partial;$		
loop	146, 155	$d1[i] = d1[i] + curr_d1partial;$	216.182544	0.31
First calculation	212, 222	$curr_d1partial = curr_xV[k] - xV[i + k * tau];$		
k=0 in separate	213, 223	$curr_d1partial = curr_d1partial * curr_d1partial;$		
loop	214, 224	$d1[i] = d1[i] + curr_d1partial;$	36,488.212672	52.45
	412	$select = cv[i];$		
	413	$dot2partial[i] = (diffpoint[0] * xV[select]);$		
	422	$select = cv[i] + k * tau;$		
	423	$dot2partial[i] = dot2partial[i] + (diffpoint[k] * xV[select]);$	14,057.002424	20.21
	435	$dot2[i] = (dot2[i] * dot2[i])/(d1[cv[i]] * df);$	2777.064773	3.99

TABLE B.2: Timing of most time consuming calculations, algorithm float 2

Timing				
Algorithm version	Line	Calculation code	Time in msec	% of total time
Fix 1		**Total execution time:**	7,935,481.866744	100
	297	$curr_d1partial = curr_xV[0] - xV[i];$		
	298	$assertFract(curr_d1partial);$		
	299	$curr_d1partial = fixround(curr_d1partial, bitwidth);$		
	300	$curr_d1partial = curr_d1partial * curr_d1partial;$		
	301	$curr_d1partial = fixround(curr_d1partial, bitwidth);$		
	302	$d1[i] = curr_d1partial;$		
	308	$curr_d1partial = curr_xV[k] - xV[i + k * tau];$		
	309	$assertFract(curr_d1partial);$		
	310	$curr_d1partial = fixround(curr_d1partial, bitwidth);$		
	311	$curr_d1partial = curr_d1partial * curr_d1partial;$		
	312	$curr_d1partial = fixround(curr_d1partial, bitwidth);$		
	313	$d1[i] = curr_d1partial;$		
	314	$assertFract(d1[i]);$		
	315	$d1[i] = fixround(d1[i], bitwidth);$	37,870.464553	0.48
	367	$curr_d1partial = curr_xV[0] - xV[i];$		
	368	$assertFract(curr_d1partial);$		
	369	$curr_d1partial = fixround(curr_d1partial, bitwidth);$		
	370	$curr_d1partial = curr_d1partial * curr_d1partial;$		
	371	$curr_d1partial = fixround(curr_d1partial, bitwidth);$		
	372	$d1[i] = curr_d1partial;$		
	379	$curr_d1partial = curr_xV[k] - xV[i + k * tau];$		
	380	$assertFract(curr_d1partial);$		
	381	$curr_d1partial = fixround(curr_d1partial, bitwidth);$		
	382	$curr_d1partial = curr_d1partial * curr_d1partial;$		
	383	$curr_d1partial = fixround(curr_d1partial, bitwidth);$		
	384	$d1[i] = curr_d1partial;$		
	385	$assertFract(d1[i]);$		
	386	$d1[i] = fixround(d1[i], bitwidth);$	6,393,424.958187	80.57
	412	$select = cv[i] + k * tau;$		
	413	$dot2partial = dot2partial + (diffpoint[k] * xV[select]);$		
	422	$assertFract(dot2partial);$		
	423	$dot2partial = fixround(dot2partial, bitwidth);$	918,770.471380	11.58

Table B.3: Timing of most time consuming calculations, algorithm fix 1

Timing				
Algorithm version	Line	Calculation code	Time in msec	% of total time
	577	$dot2[i] = mulpoint - dot2partial;$		
	578	$assertFract(dot2[i]);$		
	579	$dot2[i] = fixround(dot2[i], bitwidth);$		
	581	$dot2nom = (dot2[i] * dot2[i]);$		
	582	$assertFract(dot2nom);$		
	584	$remove0return = fixRemove0(dot2nom);$		
	585	$dot2nom = remove0return.x;$		
	586	$removed0nom = remove0return.num;$		
	587	$dot2nom = fixround(dot2nom, bitwidth);$		
	588	$dot2denom = d1[cv[i]] * df;$		
	589	$assertFract(dot2denom);$		
	591	$remove0return = fixRemove0(dot2denom);$		
	592	$dot2denom = remove0return.x;$		
	593	$removed0denom = remove0return.num;$		
	594	$dot2nom = fixround(dot2nom, bitwidth); ;$		
These two are	598	$dot2[i] = dot2nom/dot2denom;$		
in an if-statement	599	$dot2[i] = dot2[i]/powf(2, removed0nom - removed0denom);$		
Associated else	602	$dot2[i] = MAXFIX;$	504,950.097280	6.36

Table B.4: Timing of most time consuming calculations, algorithm fix 1

Timing				
Algorithm version	Line	Calculation code	Time in msec	% of total time
NoMalloc		**Total execution time:**	69,303.887770	100
First calculation	149, 157	$curr_d1partial = curr_xV[k] - xV[i + k * tau];$		
k=0 in separate	150, 158	$curr_d1partial = curr_d1partial * curr_d1partial;$		
loop	151, 159	$d1[i] = d1[i] + curr_d1partial;$	216.188233	0.31
First calculation	212, 221	$curr_d1partial = curr_xV[k] - xV[i + k * tau];$		
k=0 in separate	213, 222	$curr_d1partial = curr_d1partial * curr_d1partial;$		
loop	214, 223	$d1[i] = d1[i] + curr_d1partial;$	36,487.143166	52.65
	400	$select = cv[i];$		
	401	$dot2partial[i] = (diffpoint[0] * xV[select]);$		
	409	$select = cv[i] + k * tau;$		
	410	$dot2partial[i] = dot2partial[i] + (diffpoint[k] * xV[select]);$	13,983.792425	20.18
	422	$dot2[i] = (dot2[i] * dot2[i])/(d1[cv[i]] * df);$	2759.093650	3.98

Table B.5: Timing of most time consuming calculations, algorithm no malloc

Timing				
Algorithm version	Line	Calculation code	Time in msec	% of total time
FloatOpt		**Total execution time:**	52,455.45	100
		$curr_d1partial = curr_xV[0] - xV[i + 0 * tau];$ $d1temp+ = curr_d1partial * curr_d1partial;$ $curr_d1partial = curr_xV[1] - xV[i + 1 * tau];$ $d1temp+ = curr_d1partial * curr_d1partial;$ $curr_d1partial = curr_xV[2] - xV[i + 2 * tau];$ $d1temp+ = curr_d1partial * curr_d1partial;$ $curr_d1partial = curr_xV[3] - xV[i + 3 * tau];$ $d1temp+ = curr_d1partial * curr_d1partial;$ $curr_d1partial = curr_xV[4] - xV[i + 4 * tau];$ $d1temp+ = curr_d1partial * curr_d1partial;$ $curr_d1partial = curr_xV[5] - xV[i + 5 * tau];$ $d1temp+ = curr_d1partial * curr_d1partial;$ $curr_d1partial = curr_xV[6] - xV[i + 6 * tau];$ $d1temp+ = curr_d1partial * curr_d1partial;$ $d1[i] = d1temp;$	156.91	0.30
		$curr_d1partial = curr_xV[0] - xV[i + 0 * tau];$ $d1temp+ = curr_d1partial * curr_d1partial;$ $curr_d1partial = curr_xV[1] - xV[i + 1 * tau];$ $d1temp+ = curr_d1partial * curr_d1partial;$ $curr_d1partial = curr_xV[2] - xV[i + 2 * tau];$ $d1temp+ = curr_d1partial * curr_d1partial;$ $curr_d1partial = curr_xV[3] - xV[i + 3 * tau];$ $d1temp+ = curr_d1partial * curr_d1partial;$ $curr_d1partial = curr_xV[4] - xV[i + 4 * tau];$ $d1temp+ = curr_d1partial * curr_d1partial;$ $curr_d1partial = curr_xV[5] - xV[i + 5 * tau];$ $d1temp+ = curr_d1partial * curr_d1partial;$ $curr_d1partial = curr_xV[6] - xV[i + 6 * tau];$ $d1temp+ = curr_d1partial * curr_d1partial;$ $d1[i] = d1temp;$	26,464.31	50.45

Timing				
Algorithm version	Line	Calculation code	Time in msec	% of total time
		$select = cv[i];$ $dot2partial = dot2partial + (diffpoint[0] * xV[select]);$ $select = cv[i] + tau;$ $dot2partial = dot2partial + (diffpoint[1] * xV[select]);$ $select = cv[i] + 2 * tau;$ $dot2partial = dot2partial + (diffpoint[2] * xV[select]);$ $select = cv[i] + 3 * tau;$ $dot2partial = dot2partial + (diffpoint[3] * xV[select]);$ $select = cv[i] + 4 * tau;$ $dot2partial = dot2partial + (diffpoint[4] * xV[select]);$ $select = cv[i] + 5 * tau;$ $dot2partial = dot2partial + (diffpoint[5] * xV[select]);$ $select = cv[i] + 6 * tau;$ $dot2partial = dot2partial + (diffpoint[6] * xV[select]);$	14,475.42	27.60
$if denom > nom$ else		$dot2nom = mulpoint - dot2partial;$ $dot2nom = dot2nom * dot2nom;$ $dot2denom = d1[cv[i]] * df;$ $dot2[i] = dot2nom/dot2denom;$ $dot2[i] = 1;$	20.63	0.04

Table B.6: Timing of most time consuming calculations, algorithm FloatOpt from fixed point experiences.

Appendix C

Deviations

Mean absolute value between $d1$ values calculated by C script and values calculated by the hardware modules. *Channel* in Table 4.4 describes which of the files containing EEG-signals[1] the calculation is based on. *Exponent* in Table 4.4 describes which exponent from the channel the deviation calculation is based on.

Deviations of module utilizing the floating point expansion package

Deviations	
Channel,exponent	Absolute mean deviation
1,1	0.000077
1,1000	0.000033
1,2000	0.000029
1,3516	0.000038
2,1	0.000023
2,503	0.000019
2,2000	0.000026
2,3516	0.000031
3,1	0.000007
3,1005	0.000015
3,2000	0.000018
3,3516	0.000014

[1] Channels 1 to 32.

Deviations	
Channel,exponent	Absolute mean deviation
4,1	0.000021
4,1000	0.000019
4,1507	0.000066
4,3516	0.000015
5,1	0.000008
5,1000	0.000010
5,2009	0.000012
5,3516	0.000008
6,1	0.000009
6,1000	0.000009
6,2511	0.000012
6,3516	0.000015
7,1	0.000016
7,1000	0.000027
7,2000	0.000029
7,3516	0.000014
Average mean deviation:	0.000022

TABLE C.1: Deviation for float extension package approach

Deviations of module utilizing the fixed point expansion package

Deviations	
Channel,exponent	Absolute mean deviation
1,1	0.000075
1,1000	0.000033
1,2000	0.000029
1,3516	0.000037
2,1	0.000023
2,503	0.000019
2,2000	0.000026
2,3516	0.000032
3,1	0.000008
3,1005	0.000015
3,2000	0.000019
3,3516	0.000015

Deviations	
Channel,exponent	Absolute mean deviation
4,1	0.000020
4,1000	0.000019
4,1507	0.000066
4,3516	0.471068
5,1	0.000007
5,1000	0.000010
5,2009	0.000012
5,3516	0.000009
6,1	0.000009
6,1000	0.000009
6,2511	0.000012
6,3516	0.000015
7,1	0.943861
7,1000	0.000028
7,2000	0.000029
7,3516	0.000015
Average mean deviation:	0.050554

Table C.2: Deviation for float extension package approach

Deviations of module using standard VHDL syntax

Deviations	
Channel,exponent	Absolute mean deviation
1,1	0.000091
1,1000	0.000041
1,2000	0.000034
1,3516	0.000042
2,1	0.000031
2,503	0.000024
2,2000	0.000030
2,3516	0.000043
3,1	0.000010
3,1005	0.000021
3,2000	0.000019
3,3516	0.000017

Deviations	
Channel,exponent	Absolute mean deviation
4,1	0.000024
4,1000	0.000021
4,1507	0.000073
4,3516	0.000021
5,1	0.000010
5,1000	0.000014
5,2009	0.000013
5,3516	0.000017
6,1	0.000009
6,1000	0.000014
6,2511	0.000016
6,3516	0.000014
7,1	0.000020
7,1000	0.000031
7,2000	0.000029
7,3516	0.000016
Average mean deviation:	0.000027

TABLE C.3: Deviation for the approach using standard VHDL

Appendix D

VHDL codes

D.1 Float approach

D.1.1 Design 1

```
library std;
library IEEE;
library ieee_proposed;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use std.textio.all;

use ieee_proposed.fixed_float_types.all;
use ieee_proposed.fixed_pkg.all;
use ieee_proposed.float_pkg.all;

entity float_point_package_module is
    port(
    clk : in STD_LOGIC;
    enable : in STD_LOGIC;
    curr_xV : in float(8 downto -23);
    xV : in float(8 downto -23);
    d1 : out float(8 downto -23));
end float_point_package_module;

architecture Behavioral of float_point_package_module is
    constant tau : integer := 4;

    type array_float_small is array (0 to 6) of float(8 downto -23);
```

```
    type array_float_big is array (0 to 2047) of float(8 downto -23);
    type array_float is array (0 to 2007) of float(8 downto -23);

    signal read_enable, calc_enable,write_enable : STD_LOGIC := '0';
    signal array_curr_xV : array_float_small;
    signal array_xV : array_float_big;
    signal array_d1 : array_float;
    signal endoffile : bit := '0';
    signal  linenumber : integer:=1;

begin
    process(clk)
    variable i,index : integer := 0;
    variable k : integer := 0;
    variable d1temp : float(8 downto -23) :=(others=>'0');
    variable d1_fixed : array_float :=((others=> (others=>'0')));

    file      outfile  : text is out "./TestFiles/d1_1.txt";  --declare output file
    variable  outline  : line;   --line number declaration

begin
if (clk'event and clk = '1') then
    if enable = '1' then
        read_enable <= '1';
    end if;

    if read_enable = '1' then
        read_enable <= '1';
        if(i<=6)then
            array_curr_xV(i) <= curr_xV;
        end if;
        array_xV(i) <= xV;
        i := i+1;
        if(i = 2048)then
            read_enable <= '0';
            i := 0;
            calc_enable <= '1';
        end if;

    elsif calc_enable = '1' then
        d1temp := (array_curr_xV(k)-array_xV(i+k*tau))*(array_curr_xV(k)-        ↩
                  array_xV(i+k*tau));
        d1_fixed(i) := d1_fixed(i) + d1temp;

        i := i+1;

        if(i = 2008  and k < 6)then
            i := 0;
            k := k+1;

        elsif(i=2008 and k=6)then
            i := 0;
            k := 0;
            calc_enable <= '0';
            write_enable <= '1';
```

```
        end if;
    elsif write_enable = '1' then

        d1 <= d1_fixed(index);
------------------------------ADDED FOR TEST PURPOSES------------------------ Writing to file
--              if(endoffile='0') then  --if the file end is not reached.
--                  write(outline, to_real(d1_fixed(index),false,false), right, 10);
--to_real for readable numbers
--                  writeline(outfile, outline);
--                  linenumber <= linenumber + 1;
--              else
--                  null;
--              end if;
----------------------------------END TEST PURPOSES-------------------------
            index := index+1;
        if(index = 2008)then
        index := 0;
        write_enable <= '0';
        end if;
    end if;

end if;
end process;
end Behavioral;
```

D.1.2 Design 2

```
library std;
library IEEE;
library ieee_proposed;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use std.textio.all;

use ieee_proposed.fixed_float_types.all;
use ieee_proposed.fixed_pkg.all;
use ieee_proposed.float_pkg.all;


entity float_point_package_module is
    port(
    clk : in STD_LOGIC;
    enable : in STD_LOGIC;
    curr_xV : in float(8 downto -23);
    xV : in float(8 downto -23);
    d1 : out float(8 downto -23));
end float_point_package_module;

architecture Behavioral of float_point_package_module is
    constant tau : integer := 4;

    type array_float_small is array (0 to 6) of float(8 downto -23);
```

```
    type array_float_big is array (0 to 2047) of float(8 downto -23);
    type array_float is array (0 to 2007) of float(8 downto -23);

    signal read_enable, calc_enable,write_enable : STD_LOGIC := '0';
    signal array_curr_xV : array_float_small;
    signal array_xV : array_float_big;
    signal array_d1 : array_float;

begin
    process(clk)
    variable i,index : integer := 0;
    variable k : integer := 0;
    variable d1temp : float(8 downto -23) :=(others=>'0');
    variable d1_fixed : array_float :=((others=> (others=>'0')));


begin
if (clk'event and clk = '1') then
    if enable = '1' then
        read_enable <= '1';
    end if;

    if read_enable = '1' then
        read_enable <= '1';
        if(i<=6)then
            array_curr_xV(i) <= curr_xV;
        end if;
        array_xV(i) <= xV;
        i := i+1;
        if(i = 2048)then
            read_enable <= '0';
            i := 0;
            calc_enable <= '1';
        end if;

    elsif calc_enable = '1' then
        d1temp := (array_curr_xV(k)-array_xV(i+k*tau))*(array_curr_xV(k)-        ↩
                    array_xV(i+k*tau));
        d1_fixed(i) := d1_fixed(i) + d1temp;
        d1temp := (array_curr_xV(k)-array_xV(i+1+k*tau))*(array_curr_xV(k)-      ↩
                    array_xV(i+1+k*tau));
        d1_fixed(i+1) := d1_fixed(i+1) + d1temp;

        i := i+2;

        if(i = 2008  and k < 6)then
            i := 0;
            k := k+1;

        elsif(i=2008 and k=6)then
            i := 0;
            k := 0;
            calc_enable <= '0';
            write_enable <= '1';
        end if;
```

```
    elsif write_enable = '1' then
        d1 <= d1_fixed(index);
        index := index+1;
        if(index = 2008)then
        index := 0;
        write_enable <= '0';
        end if;
    end if;

end if;
end process;
end Behavioral;
```

D.2 Fixed approach

D.2.1 Design 1

```
library std;
library IEEE;
library ieee_proposed;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use std.textio.all;

use ieee_proposed.fixed_float_types.all;
use ieee_proposed.fixed_pkg.all;
use ieee_proposed.float_pkg.all;


entity fixed_point_package_module is
    port(
    clk : in STD_LOGIC;
    enable : in STD_LOGIC;
    curr_xV : in float(8 downto -23);
    xV : in float(8 downto -23);
    d1 : out float(8 downto -23));
end fixed_point_package_module;

architecture Behavioral of fixed_point_package_module is
    constant tau : integer := 4;

    type array_fixed_small is array (0 to 6) of sfixed(11 downto -20);
    type array_fixed_big is array (0 to 2047) of sfixed(11 downto -20);
    type array_fixed is array (0 to 2007) of sfixed(11 downto -20);

    signal read_enable, calc_enable,write_enable : STD_LOGIC := '0';
    signal array_curr_xV : array_fixed_small;
    signal array_xV : array_fixed_big;
    signal array_d1 : array_fixed;
    signal d1_fixed : array_fixed :=((others=> (others=>'0')));
```

```
begin
    process(clk)
    variable i,index : integer := 0;
    variable k : integer := 0;
    variable d1temp : sfixed(11 downto -20);

    begin
        if (clk'event and clk = '1') then
            if enable = '1' then
                read_enable <= '1';
            end if;

            if read_enable = '1' then
                read_enable <= '1';
                if(i<=6)then
                    array_curr_xV(i) <= to_sfixed((curr_xV),11,-20);
                end if;
                array_xV(i) <= to_sfixed(unresolved_float(xV),11,-20);
                i := i+1;
                if(i = 2048)then
                    read_enable <= '0';
                    i := 0;
                    calc_enable <= '1';
                end if;

            elsif calc_enable = '1' then
                d1temp := resize((array_curr_xV(k)-array_xV(i+k*tau))*         ↩
                            (array_curr_xV(k)-array_xV(i+k*tau)),11,-20);
                d1_fixed(i) <= resize((d1_fixed(i) + d1temp),11,-20);
                i := i+1;

                if(i = 2008  and k < 6)then
                    i := 0;
                    k := k+1;
                elsif(i=2008 and k=6)then
                    i := 0;
                    k := 0;
                    calc_enable <= '0';
                    write_enable <= '1';
                end if;
            elsif write_enable = '1' then
                    d1 <= to_float(d1_fixed(index));
                index := index+1;
                if(index = 2008)then
                    index := 0;
                    write_enable <= '0';
                end if;
            end if;

        end if;
    end process;
end Behavioral;
```

D.2.2 Design 2

```
library std;
library IEEE;
library ieee_proposed;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use std.textio.all;

use ieee_proposed.fixed_float_types.all;
use ieee_proposed.fixed_pkg.all;
use ieee_proposed.float_pkg.all;

entity fixed_point_package_module is
    port(
    clk : in STD_LOGIC;
    enable : in STD_LOGIC;
    curr_xV : in float(8 downto -23);
    xV : in float(8 downto -23);
    d1 : out float(8 downto -23));
end fixed_point_package_module;

architecture Behavioral of fixed_point_package_module is
    constant tau : integer := 4;

    type array_fixed_small is array (0 to 6) of sfixed(11 downto -20);
    type array_fixed_big is array (0 to 2047) of sfixed(11 downto -20);
    type array_fixed is array (0 to 2007) of sfixed(11 downto -20);

    signal read_enable, calc_enable, write_enable : STD_LOGIC := '0';
    signal array_curr_xV : array_fixed_small;
    signal array_xV : array_fixed_big;
    signal array_d1 : array_fixed;

begin
    process(clk)
    variable i, index : integer := 0;
    variable k : integer := 0;
    variable d1temp, temp1, temp2 : sfixed(11 downto -20);
    variable d1_fixed : array_fixed :=((others=> (others=>'0')));

    begin
        if (clk'event and clk = '1') then
            if enable = '1' then
                read_enable <= '1';
            end if;

            if read_enable = '1' then
                read_enable <= '1';
                if(i<=6)then
                    array_curr_xV(i) <= to_sfixed((curr_xV),11,-20);
                end if;
                array_xV(i) <= to_sfixed(unresolved_float(xV),11,-20);
```

```
                i := i+1;
                if(i = 2048)then
                    read_enable <= '0';
                    i := 0;
                    calc_enable <= '1';
                end if;

            elsif calc_enable = '1' then
                temp1 :=array_xV(i+k*tau);
                temp2 :=array_xV(i+1+k*tau);
                d1temp := resize((array_curr_xV(k)-temp1)*          ↩
                          (array_curr_xV(k)-temp1),11,-20);
                d1_fixed(i) := resize((d1_fixed(i) + d1temp),11,-20);

                d1temp := resize((array_curr_xV(k)-temp2)*          ↩
                          (array_curr_xV(k)-temp2),11,-20);
                d1_fixed(i+1) := resize((d1_fixed(i+1) + d1temp),11,-20);
                i := i+2;

                if(i = 2008  and k < 6)then
                    i := 0;
                    k := k+1;
                elsif(i=2008 and k=6)then
                    i := 0;
                    k := 0;
                    calc_enable <= '0';
                    write_enable <= '1';
                end if;
            elsif write_enable = '1' then
                d1 <= to_float(d1_fixed(index));
                index := index+1;
                if(index = 2008)then
                    index := 0;
                    write_enable <= '0';
                end if;
            end if;

        end if;
    end process;
end Behavioral;
```

D.3 Standard VHDL

To make the conversion more accurate, extend the if-statement in the conversions, f.ex add elsif(d1temp(6)='1')then and so on

```
library std;
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
```

```
use std.textio.all;


entity fix_point_module is
    port(
    clk : in STD_LOGIC;
    rst : in STD_LOGIC;
    enable : in STD_LOGIC;
    curr_xV : in std_logic_vector(31 downto 0);
    xV : in std_logic_vector(31 downto 0);
    d1 : out std_logic_vector(31 downto 0);
    o_rdy : out STD_LOGIC);
end fix_point_module;


architecture Behavioral of fix_point_module is
    constant tau : integer := 4;
    constant exp_base : integer := 128; --Design for 32-bit float

    type ram_small is array (0 to 6) of std_logic_vector(31 downto 0);
    type ram_big is array (0 to 2047 ) of std_logic_vector(31 downto 0);
    type ram_8 is array (0 to 2007) of std_logic_vector(31 downto 0);

    signal read_enable, calc_enable,write_enable : STD_LOGIC := '0';
    signal ram_curr_xV : ram_small;
    signal ram_xV : ram_big;
    signal ram : ram_8;


begin
    process(clk)
    variable i,index : integer := 0;
    variable k : integer := 0;
    variable fraction_curr,fraction_xV : std_logic_vector(19 downto 0):=(others=>'0');
    variable res : std_logic_vector(63 downto 0);
    variable temp1,fix_num_xV,fix_num_curr : std_logic_vector(31 downto 0)           ↩
             :=(others=>'0');
    variable temp2 : std_logic_vector(11 downto 0);
    variable fraction_result : std_logic_vector(19 downto 0):=(others=>'0');
    variable integer_curr,integer_xV : std_logic_vector(11 downto 0):=(others=>'0');
    variable integer_result : std_logic_vector(11 downto 0):=(others=>'0');
    variable d1temp : std_logic_vector(31 downto 0) :=(others=>'0');
    variable exp,upper_index,lower_index,exp_bit,shift_number_xV,                     ↩
             shift_number_curr : integer := 0;

    begin
        if (clk'event and clk = '1') then
            if(enable = '0' and write_enable = '0')then
            o_rdy <= '0';
            else
                null;
            end if;
            if(rst = '1')then
                ram <= (others=>(others=>'0'));
                ram_curr_xV <= (others=>(others=>'0'));
```

```
                ram_xV <= (others=>(others=>'0'));
                i := 0;
                k := 0;
                index := 0;
                fix_num_curr:=(others=>'0');
                fix_num_xV:=(others=>'0');

                fraction_curr:= (others=>'0');
                fraction_xV:= (others=>'0');
                integer_curr:= (others=>'0');
                integer_xV:= (others=>'0');
                fraction_result:= (others=>'0');

                d1temp := (others=>'0');
                res := (others=>'0');
                temp1:= (others=>'0');
                temp2:= (others=>'0');

                read_enable <= '0';
                write_enable <= '0';
                calc_enable <= '0';

                o_rdy <= '0';

            else
                if enable = '1' then
                    read_enable <= '1';
                    o_rdy <= '0';
                else
                    null;
                end if;
                if read_enable = '1' then
                    read_enable <= '1';
                    if(i<=6)then

--Due to difference in fraction in fix and mantissa in float, see report section 2.5
                    shift_number_curr := to_integer(unsigned(curr_xV(30 downto 23)))↩
                                    -exp_base;
----------------NOT BENEFICIAL WITH VARIABLE INDEXING AT SYNTHESIS----------------
------------------------this code is replaced by * at synthesis--------------------
-- concatenating and adding implicit 1
                    if(shift_number_curr > 2)then

--add zeros in vector, rounding
                        fix_num_curr(20+shift_number_curr downto shift_number_curr-1) ↩
                            := '1' & curr_xV(22 downto 0);
                        fix_num_curr(shift_number_curr-2 downto 0)              ↩
                            := (others=>'0');
                    elsif(shift_number_curr < 2)then
                        fix_num_curr(20+shift_number_curr downto 0)             ↩
                            :='1' & curr_xV(22 downto 2-shift_number_curr);
                    else
                        fix_num_curr(20+shift_number_curr downto 0)             ↩
                            := '1'&  curr_xV(22 downto 0);
                    end if;
```

```
                    fix_num_curr(31 downto 21 + shift_number_curr)              ↩
                        := (others=>'0');
------------------------------Replaced by * at synthesis------------------------------

--------------------------------------------*--------------------------------------------
----------------------REPLACES ABOVE IF-STRUCTURE AT SYNTHESIS----------------------
--                  if(shift_number_curr = 9)then
--                    fix_num_curr(29 downto 6) := '1' & curr_xV(22 downto 0);
--                    fix_num_curr(31 downto 30) := (others=>'0');
--                  elsif(shift_number_curr = 8)then
--                    fix_num_curr(28 downto 5) := '1' & curr_xV(22 downto 0);
--                    fix_num_curr(31 downto 29) := (others=>'0');
--                  elsif(shift_number_curr = 7)then
--                    fix_num_curr(27 downto 4) := '1' & curr_xV(22 downto 0);
--                    fix_num_curr(31 downto 28) := (others=>'0');
--                  elsif(shift_number_curr = 6)then
--                    fix_num_curr(26 downto 3) := '1' & curr_xV(22 downto 0);
--                    fix_num_curr(31 downto 27) := (others=>'0');
--                  elsif(shift_number_curr = 5)then
--                    fix_num_curr(25 downto 2) := '1' & curr_xV(22 downto 0);
--                    fix_num_curr(31 downto 26) := (others=>'0');
--                  elsif(shift_number_curr = 4)then
--                    fix_num_curr(24 downto 1) := '1' & curr_xV(22 downto 0);
--                    fix_num_curr(31 downto 25) := (others=>'0');
--                  elsif(shift_number_curr = 3)then
--                    fix_num_curr(23 downto 0) := '1' & curr_xV(22 downto 0);
--                    fix_num_curr(31 downto 24) := (others=>'0');
--                  elsif(shift_number_curr = 2)then
--                    fix_num_curr(22 downto 0) := '1'&  curr_xV(22 downto 1);
--                    fix_num_curr(31 downto 23) := (others=>'0');
--                  elsif(shift_number_curr = 1)then
--                    fix_num_curr(21 downto 0) :='1' & curr_xV(22 downto 2);
--                    fix_num_curr(31 downto 22) := (others=>'0');
--                  elsif(shift_number_curr = 0)then
--                    fix_num_curr(20 downto 0) :='1' & curr_xV(22 downto 3);
--                    fix_num_curr(31 downto 21) := (others=>'0');
--                  elsif(shift_number_curr = -1)then
--                    fix_num_curr(19 downto 0) :='1' & curr_xV(22 downto 4);
--                    fix_num_curr(31 downto 20) := (others=>'0');
--                  elsif(shift_number_curr = -2)then
--                    fix_num_curr(18 downto 0) :='1' & curr_xV(22 downto 5);
--                    fix_num_curr(31 downto 19) := (others=>'0');
--                  elsif(shift_number_curr = -3)then
--                    fix_num_curr(17 downto 0) :='1' & curr_xV(22 downto 6);
--                    fix_num_curr(31 downto 18) := (others=>'0');
--                  elsif(shift_number_curr = -4)then
--                    fix_num_curr(16 downto 0) :='1' & curr_xV(22 downto 7);
--                    fix_num_curr(31 downto 17) := (others=>'0');
--                  elsif(shift_number_curr = -5)then
--                    fix_num_curr(15 downto 0) :='1' & curr_xV(22 downto 8);
--                    fix_num_curr(31 downto 16) := (others=>'0');
--                  elsif(shift_number_curr = -6)then
--                    fix_num_curr(14 downto 0) :='1' & curr_xV(22 downto 9);
--                    fix_num_curr(31 downto 15) := (others=>'0');
--                  elsif(shift_number_curr = -7)then
```

```
--                          fix_num_curr(13 downto 0) :='1' & curr_xV(22 downto 10);
--                          fix_num_curr(31 downto 14) := (others=>'0');
--                        else
--                          fix_num_curr(12 downto 0) :='1' & curr_xV(22 downto 11);
--                          fix_num_curr(31 downto 13) := (others=>'0');
--                        end if;
------------------END REPLACEMENT OF ABOVE IF-STRUCTURE AT SYNTHESIS------------------
--------------------------------------------*--------------------------------------------

                    if(curr_xV(31) = '1')then
                        fix_num_curr := not fix_num_curr;
                    else
                        fix_num_curr := fix_num_curr;
                    end if;
                    ram_curr_xV(i) <= fix_num_curr;
                else
                    null;
                end if;
                    if(to_integer(unsigned(xV(30 downto 23)))=exp_base)then
                        shift_number_xV := 0;
                    else
--Due to difference in fraction in fix and mantissa in float, see report section 2.5
                        shift_number_xV := to_integer(unsigned(xV(30 downto 23)))  ↩
                                           - exp_base;
                    end if;
-- concatenating and adding implicit 1
                    if(shift_number_xV > 2)then
--add zeros in vector, rounding
                        fix_num_xV(20+shift_number_xV downto 0)                     ↩
                            :='1' & xV(22 downto 0);
                        fix_num_xV(shift_number_xV-2 downto 0)                      ↩
                            := (others=>'0');
                    elsif(shift_number_xV < 2)then
                        fix_num_xV(20+shift_number_xV downto 0)                     ↩
                            :='1' & xV(22 downto 2-shift_number_xV);
                    else
                        fix_num_xV(20+shift_number_xV downto 0)                     ↩
                            :='1' & xV(22 downto 0);
                    end if;
--Fill in zeros
                    fix_num_xV(31 downto 21 + shift_number_xV) := (others=>'0');
                    if(xV(31) = '1')then
                        fix_num_xV(31 downto 0) := not fix_num_xV(31 downto 0);
                    else
                        fix_num_xV(31 downto 0) := fix_num_xV(31 downto 0);
                    end if;

                    ram_xV(i) <= fix_num_xV;
                    fix_num_xV := (others=>'0');
                    fix_num_curr := (others=>'0');
                    i := i+1;
                    if(i = 2048)then
                        read_enable <= '0';
                        i := 0;
                        calc_enable <= '1';
```

```
                    else
                        read_enable <= '1';
                    end if;

                elsif calc_enable = '1' then
                    fix_num_curr := ram_curr_xV(k);
                    fix_num_xV := ram_xV(i+k*tau);

                    fraction_curr := fix_num_curr(19 downto 0);
                    fraction_xV := fix_num_xV(19 downto 0);
                    integer_curr := fix_num_curr(31 downto 20);
                    integer_xV := fix_num_xV(31 downto 20);


--Different operations depending on sign of numbers:
--curr_xV negative & xV negative: -a - (-b) => -a+b
--curr_xV negative & xV positive: -a - (+b) => -a-b => -(+a + (+b))
--curr_xV positive & xV negative: +a - (-b) => +a+b
--curr_xV positive & xV positive: +a - (+b) => +a-b



                    if((integer_curr(11)='1' and integer_xV(11)='0')          ↩
                    or (integer_curr(11)='0' and integer_xV(11)='1'))then

--curr_xV negative & xV positive: -a - (+b) => -a-b => -(+a + (+b))
                        if(integer_curr(11)='1' and integer_xV(11)='0')then
                            fraction_curr := not fraction_curr;
                            if(unsigned(integer_curr) /= 0)then
                                integer_curr := not integer_curr;
                            else
                                null;
                            end if;
--curr_xV positive & xV negative: +a - (-b) => +a+b
                        else
                            fraction_xV := not fraction_xV;
                            if(unsigned(integer_xV) /= 0)then
                                integer_xV := not integer_xV;
                            end if;
                        end if;

                        fraction_result := std_logic_vector(unsigned(fraction_curr) ↩
                                           + unsigned(fraction_xV));

--test for overflow in fraction-addition, if overflow, add one to(one of the) integers
                        if((fraction_result<fraction_curr)                     ↩
                        or (fraction_result<fraction_xV))then
                            integer_curr := std_logic_vector(signed(integer_curr)+1);
                        else
                            null;
                        end if;

                        temp2 := std_logic_vector(unsigned(integer_curr)       ↩
                                 + unsigned(integer_xV));
```

```
--curr_xV negative & xV negative: -a - (-b) => -a+b => b-a
                    elsif(integer_curr(11)='1' and integer_xV(11)='1')then
                        fraction_curr := not fraction_curr;
                        fraction_xV := not fraction_xV;
                        integer_xV := not integer_xV;
                        integer_curr := not integer_curr;

                        fraction_result := std_logic_vector(unsigned(fraction_xV) ↩
                                    - unsigned(fraction_curr));

--ratio between numbers decides whether to add/subtract/flip
--bits(if fraction_result is negative(curr>xV))

                      if(fraction_curr > fraction_xV)then
                          temp2 := std_logic_vector(unsigned(integer_xV)      ↩
                                 - unsigned(integer_curr));
                        if(integer_curr>integer_xV)then
                            if(signed(temp2) < 0)then
                                temp2 := std_logic_vector(unsigned(not temp2) + 1);
                            else
                            temp2 := not temp2;
                            end if;
                            fraction_result := not fraction_result;
                        elsif(integer_curr<integer_xV)then
                            temp2 := std_logic_vector(unsigned(temp2)-1);
                        else
                            fraction_result := not fraction_result;
                        end if;

                      else
                        if(integer_curr>integer_xV)then
                            temp2 := std_logic_vector(unsigned(integer_curr)    ↩
                                 - unsigned(integer_xV)-1);
                            fraction_result := not fraction_result;
                        else
                            temp2 := std_logic_vector(unsigned(integer_xV)      ↩
                                 - unsigned(integer_curr));
                        end if;
                      end if;

--curr_xV positive & xV positive: +a - (+b) => +a-b
                    elsif(integer_curr(11)='0' and integer_xV(11)='0')then

                        fraction_result := std_logic_vector(unsigned(fraction_curr)↩
                                   - unsigned(fraction_xV));

--corresponding to same as above for curr neg & xv neg
                        if(fraction_curr < fraction_xV)then
                            if(integer_curr = integer_xV)then
                                temp2 := std_logic_vector(unsigned(integer_curr)↩
                                      - unsigned(integer_xV));
                                fraction_result := not fraction_result;
                            elsif(integer_curr > integer_xV)then
                                temp2 := std_logic_vector(unsigned(integer_curr)↩
                                      - unsigned(integer_xV)-1);
```

```
                                    else
                                        temp2 := std_logic_vector(unsigned(integer_xV)  ↩
                                                  - unsigned(integer_curr));
                                        fraction_result := not fraction_result;
                                    end if;

                                else
                                    if(integer_xV > integer_curr)then
                                        temp2 := std_logic_vector(unsigned(integer_xV)  ↩
                                                  - unsigned(integer_curr)-1);
                                        fraction_result := not fraction_result;
                                    else
                                        temp2 := std_logic_vector(unsigned(integer_curr)↩
                                                  - unsigned(integer_xV));
                                    end if;
                                end if;
                        else
                         null;
                        end if;
--concatenate due to multiplication, will not have overflow, so decimal point is "safe"
                        temp1 := temp2 & fraction_result;
                        res := std_logic_vector(unsigned(temp1)*unsigned(temp1));
                        d1temp := res(51 downto 40) & res(39 downto 20);
--accumulate
                        ram(i) <= std_logic_vector(unsigned(ram(i)) + unsigned(d1temp));

                        i := i+1;
                        if(i = 2008  and k < 6)then
                            i := 0;
                            k := k+1;
                        elsif(i=2008 and k=6)then
                            i := 0;
                            k := 0;
                            calc_enable <= '0';
                            write_enable <= '1';
                        else
                            null;
                        end if;
                elsif write_enable = '1' then
                    d1temp := ram(index);
--d1 assignments in following if-structure replaces next if-structure for d1 at ↩
    synthesis, as variable indexing isnt beneficial at synthesis
                        if (d1temp(30)='1') then --To float conversion
                            exp_bit := 10;
--                          d1 <= d1temp(31) & (std_logic_vector(to_unsigned((exp_base + ↩
    exp_bit),8))) & (d1temp(29 downto 7));
                        elsif(d1temp(29)= '1')then
                            exp_bit := 9;
--                          d1 <= d1temp(31) & (std_logic_vector(to_unsigned((exp_base + ↩
    exp_bit),8))) & (d1temp(28 downto 6));
                        elsif(d1temp(28)= '1')then
                            exp_bit := 8;
--                          d1 <= d1temp(31) & (std_logic_vector(to_unsigned((exp_base + ↩
    exp_bit),8))) & (d1temp(27 downto 5));
                        elsif(d1temp(27)= '1')then
```

```
                            exp_bit := 7;
--                          d1 <= d1temp(31) & (std_logic_vector(to_unsigned((exp_base + ↩
    exp_bit),8))) & (d1temp(26 downto 4));
                        elsif(d1temp(26)= '1')then
                            exp_bit := 6;
--                          d1 <= d1temp(31) & (std_logic_vector(to_unsigned((exp_base + ↩
    exp_bit),8))) & (d1temp(25 downto 3));
                        elsif(d1temp(25)= '1')then
                            exp_bit := 5;
--                          d1 <= d1temp(31) & (std_logic_vector(to_unsigned((exp_base + ↩
    exp_bit),8))) & (d1temp(24 downto 2));
                        elsif(d1temp(24)= '1')then
                            exp_bit := 4;
--                          d1 <= d1temp(31) & (std_logic_vector(to_unsigned((exp_base + ↩
    exp_bit),8))) & (d1temp(23 downto 1));
                        elsif(d1temp(23)= '1')then
                            exp_bit := 3;
--                          d1 <= d1temp(31) & (std_logic_vector(to_unsigned((exp_base + ↩
    exp_bit),8))) &(d1temp(22 downto 0));
                        elsif(d1temp(22)= '1')then
                            exp_bit := 2;
--                          d1(31 downto 1) <= d1temp(31) & (std_logic_vector(to_unsigned((↩
    exp_base + exp_bit),8))) &(d1temp(21 downto 0));
--                          d1(0) <= '0';
                        elsif(d1temp(21)= '1')then
                            exp_bit := 1;
--                          d1(31 downto 2) <= d1temp(31) & (std_logic_vector(to_unsigned((↩
    exp_base + exp_bit),8))) &(d1temp(20 downto 0));
--                          d1(1 downto 0) <= (others=>'0');
                        elsif(d1temp(20)='1')then
                            exp_bit := 0;
--                          d1(31 downto 3) <= d1temp(31) & (std_logic_vector(to_unsigned((↩
    exp_base + exp_bit),8))) &(d1temp(19 downto 0));
--                          d1(2 downto 0) <= (others=>'0');
                        elsif(d1temp(19)='1')then
                            exp_bit := -1;
--                          d1(31 downto 4) <= d1temp(31) & (std_logic_vector(to_unsigned((↩
    exp_base + exp_bit),8))) &(d1temp(18 downto 0));
--                          d1(3 downto 0) <= (others=>'0');
                        elsif(d1temp(18)='1')then
                            exp_bit := -2;
--                          d1(31 downto 5) <= d1temp(31) & (std_logic_vector(to_unsigned((↩
    exp_base + exp_bit),8))) &(d1temp(17 downto 0));
--                          d1(4 downto 0) <= (others=>'0');
                        elsif(d1temp(17)='1')then
                            exp_bit := -3;
--                          d1(31 downto 6) <= d1temp(31) & (std_logic_vector(to_unsigned((↩
    exp_base + exp_bit),8))) &(d1temp(16 downto 0));
--                          d1(5 downto 0) <= (others=>'0');
                        elsif(d1temp(16)='1')then
                            exp_bit := -4;
--                          d1(31 downto 7) <= d1temp(31) & (std_logic_vector(to_unsigned((↩
    exp_base + exp_bit),8))) &(d1temp(15 downto 0));
--                          d1(6 downto 0) <= (others=>'0');
                        elsif(d1temp(15)='1')then
```

```
                        exp_bit := -5;
--                      d1(31 downto 8) <= d1temp(31) & (std_logic_vector(to_unsigned((↩
    exp_base + exp_bit),8))) &(d1temp(14 downto 0));
--                      d1(7 downto 0) <= (others=>'0');
                    elsif(d1temp(14)='1')then
                        exp_bit := -6;
--                      d1(31 downto 9) <= d1temp(31) & (std_logic_vector(to_unsigned((↩
    exp_base + exp_bit),8))) &(d1temp(13 downto 0));
--                      d1(8 downto 0) <= (others=>'0');
                    elsif(d1temp(13)='1')then
                        exp_bit := -7;
--                      d1(31 downto 10) <= d1temp(31) & (std_logic_vector(to_unsigned↩
    ((exp_base + exp_bit),8))) &(d1temp(12 downto 0));
--                      d1(9 downto 0) <= (others=>'0');
                    elsif(d1temp(12)='1')then
                        exp_bit := -8;
--                      d1(31 downto 11) <= d1temp(31) & (std_logic_vector(to_unsigned↩
    ((exp_base + exp_bit),8))) &(d1temp(11 downto 0));
--                      d1(10 downto 0) <= (others=>'0');
                    elsif(d1temp(11)='1')then
                        exp_bit := -9;
--                      d1(31 downto 12) <= d1temp(31) & (std_logic_vector(to_unsigned↩
    ((exp_base + exp_bit),8))) &(d1temp(10 downto 0));
--                      d1(11 downto 0) <= (others=>'0');
                    elsif(d1temp(10)='1')then
                        exp_bit := -10;
--                      d1(31 downto 13) <= d1temp(31) & (std_logic_vector(to_unsigned↩
    ((exp_base + exp_bit),8))) &(d1temp(9 downto 0));
--                      d1(12 downto 0) <= (others=>'0');
                    elsif(d1temp(9)='1')then
                        exp_bit := -11;
--                      d1(31 downto 14) <= d1temp(31) & (std_logic_vector(to_unsigned↩
    ((exp_base + exp_bit),8))) &(d1temp(8 downto 0));
--                      d1(13 downto 0) <= (others=>'0');
                    elsif(d1temp(8)='1')then
                        exp_bit := -12;
--                      d1(31 downto 15) <= d1temp(31) & (std_logic_vector(to_unsigned↩
    ((exp_base + exp_bit),8))) &(d1temp(7 downto 0));
--                      d1(14 downto 0) <= (others=>'0');
                    elsif(d1temp(7)='1')then
                        exp_bit := -13;
--                      d1(31 downto 16) <= d1temp(31) & (std_logic_vector(to_unsigned↩
    ((exp_base + exp_bit),8))) &(d1temp(6 downto 0));
--                      d1(15 downto 0) <= (others=>'0');
                    else
                        exp_bit := -14;
--                      d1(31 downto 17) <= d1temp(31) & (std_logic_vector(to_unsigned↩
    ((exp_base + exp_bit),8))) &(d1temp(5 downto 0));
--                      d1(16 downto 0) <= (others=>'0');
                    end if;
                upper_index :=exp_bit+19;
                if(upper_index > 22)then
                    lower_index := upper_index -22;
```

```
                    d1 <= d1temp(31)                                              ↩
                          & (std_logic_vector(to_unsigned((exp_base + exp_bit),8))) ↩
                          & (d1temp(upper_index downto lower_index));
                elsif(upper_index < 22)then
                    lower_index := 22-upper_index;
                    d1(31 downto lower_index) <= d1temp(31)                       ↩
                          & (std_logic_vector(to_unsigned((exp_base + exp_bit),8))) ↩
                          & (d1temp(upper_index downto 0));
                    d1(lower_index-1 downto 0) <= (others=>'0');
                else
                    d1 <= d1temp(31)                                              ↩
                          & (std_logic_vector(to_unsigned((exp_base + exp_bit),8))) ↩
                          & (d1temp(upper_index downto 0));
                end if; --End to float conversion
                o_rdy <= '1';
                index := index+1;
                if(index = 2008)then
                    index := 0;
                    write_enable <= '0';
                    o_rdy <= '0';
                else
                    null;
                end if;

                else
                null;
                end if;
            end if;
        else
        null;
        end if;
    end process;

end Behavioral;
```

Appendix E

Verification scripts

E.1 C-script calculation differences

```
# include <stdio.h>
# include <math.h>
# include <float.h>

# include <stdlib.h>
# include <iostream>
# include <fstream>

int main(void)
{

    std::fstream myfile1("../d1_3_1005.txt", std::ios_base::in);
    std::fstream myfile2("../d1file_3_1005.txt", std::ios_base::in);
    std::fstream myfile3("../difference.txt", std::ios_base::out);

    float temp1;
    float temp2;
    for (int i = 0; i < 2008;i++)
    {
        myfile1 >> temp1;
        myfile2 >> temp2;
        myfile3 << abs(temp1 - temp2) << '\n';
    }

}
```

E.2 Matlab-script for sorting

```
% Construct file names
format long
file_name1 = sprintf('./Diff_script/difference.txt');
double sorted_data;
double data;
% Open the files:
fid1 = fopen(file_name1, 'r');

fid3 = fopen('data_sorted_CHANNEL_EXPONENT.txt', 'w');
% If opening of file fails, end the program:
if(fid1 < 0 || fid3 < 0)
    disp('cannot open files');
    break;
end;

samples = 0;
% Read data from the file:
[data, count] = fscanf(fid1, '%f', [2008 1]);
while (count == 2008)
   [sorted_data,indices] = sort(data);
    max_element = max(data); %Used for intermediate checking
    min_element = min(data); %Used for intermediate checking
    fprintf(fid3, '%f\n\n',mean(data))
    for i=0:2008

    end
    fprintf(fid3, '%f\n',sorted_data);
    [data, count] = fscanf(fid1, '%f', [2008 1]);
end
% Close the files:
fclose(fid1);

fclose(fid3);
```

Bibliography

[1] Leon D. Iasemidis, Deng-Shan Shiau, Wanpracha Art Chaovalitwongse, J. Chris Sackellares, Panos M. Pardalos, Jose C. Principe, Paul Richard Carney, Awadhesh Prasad, Balaji Veeramani, and Konstantinos Tsakalis. Adaptive epileptic seizure prediction system. *IEEE Transactions on Biomedical Engineering*, pages 616–627, May 2003.

[2] Leon D. Iasemidis. Seizure prediction and its applications. *Neurosurgery Clinics of North America*, pages 489–506, October 2011.

[3] EECS. Single-ISA Heterogeneous MAny-core Computer project plan. page 5, October 2013.

[4] Wayne Wolf. Computers as components, principles of embedded computing system design. *Elsevier Inc*, 2008.

[5] Greg Stitt and Frank Vahid. Hardware/software partitioning of software binaries. *Computer Aided Design*, pages 164 –170, November 2002.

[6] Jidan Al-Eryani. Floating point unit. http://opencores.org/websvn,filedetails?repname=fpu100&path=%2Ffpu100%2Ftrunk%2Fdoc%2FFPU_doc.pdf, April 2014.

[7] Alan Wolf, Jack B. Swift, Harry L. Swinney, and John A. Vastano. Determining Lyapunov exponents from a time series. *Physica D: Nonlinear Phenomena*, pages 285–317, July 1985.

[8] Michael B. Taylor. Is dark silicon useful? *Design Automation Conference*, June 2012.

[9] Leon D. Iasemidis, Deng-Shan Shiau, Panos M. Pardalos, Wanpracha Art Chaovalitwongse, K. Narayanan, Awadhesh Prasad, Konstantinos Tsakalis, Paul Richard

Carney, and J. Chris Sackellares. Long-term prospective on-line real-time seizure prediction. *Clinical Neurophysiology*, pages 532–544, March 2005.

[10] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. *International Symposium on Microarchitecture*, pages 81–92, December 2003.

[11] Kenzo Van Craeynest and Lieven Eeckhout. Understanding fundamental design choices in single-ISA heterogeneous multicore architectures. *ACM Transactions on Architecture and Code Optimization*, 9, January 2013.

[12] Open cores. datasheet amber core. http://opencores.org/websvn,filedetails?repname=amber&path=%2Famber%2Ftrunk%2Fdoc%2Famber-core.pdf, October 2013.

[13] Andrew D. Booth. A signed binary multiplication technique. *The Quarterly Journal of Mechanics and Applied Mathematics*, pages 236–240, 1951.

[14] Kunio Uchiyama, Fumio Arakawa, Hironori Kasahara, Tohru Nojiri, Hideyuki Noda, Yasuhiro Tawara, Aiko Idehara, Kenichi Iwata, and Hiroaki Shikano. Heterogeneous Multicore Processor Technologies for Embedded Systems. *Springer*, 2012.

[15] Giovanni De Micheli and Rajesh K. Gupta. Hardware/software co-design. *Proceedings of the IEEE*, pages 349 – 365, March 1997.

[16] Brian Wilson Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, September 1970.

[17] Partha Biswas, Sudarshan Banerjee, Nikil D. Dutt, Laura Pozzi, and Paolo Ienne. ISEGEN: An iterative improvement-based ISE generation technique for fast customization of processors. *IEEE Transactions on Very Large Scale Integration (VLSI) systems*, pages 754–762, July 2006.

[18] IEEE. IEEE standard for floating-point arithmetic. *IEEE std 754-2008*, August 2008.

[19] IEEE. IEEE standard VHDL language reference manual. *IEEE std 1076-2008*, August 2009.

[20] David Bishop. Floating point package user's guide. http://vhdl.org/fphdl/Float_ug.pdf, April 2014.

[21] David Bishop. Fixed point package user's guide. http://vhdl.org/fphdl/Fixed_ug.pdf, April 2014.

[22] J.-C. Roux, Reuben H. Simoyi, and Harry L. Swinney. Observation of a strange attractor. *Physica D: Nonlinear Phenomena*, pages 257–266, December 1982.

[23] Marton Leren Teilgård. Integration of Hardware accelerators on the SHMAC platform. June 2014.