NTNU – Trondheim
Norwegian University of
Science and Technology

Implementation of Biomedical Algorithm on the SHMAC Platform

Ilias Lousis

Embedded Computing Systems
Submission date: June 2015
Supervisor: Per Gunnar Kjeldsberg, IET
Co-supervisor: Donn Morrison, IET

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

NTNU – Trondheim
Norwegian University of
Science and Technology

Implementation of Biomedical Algorithm on the SHMAC Platform

Ilias Lousis

Embedded Computing Systems
Submission Date: June 2015
Supervisor: Per Gunnar Kjeldsberg, IET
Co-supervisor: Donn Morrison, IDI

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Problem Description

Candidate name: Ilias Lousis

Thesis title: Implementation of Biomedical Algorithm on the SHMAC Platform

Problem description:

Single-ISA Heterogeneous MAny-core Computer (SHMAC) is an ongoing research project within the Energy Efficient Computing Systems (EECS) strategic research area. EECS is a cooperation between researchers from the CARD-group at IDI and CAS-group at IET. SHMAC is running on an FPGA and is an evaluation platform for research on heterogeneous tile-based network-on-chip 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. See: http://www.ntnu.edu/ime/eecs/shmac

In the spring and autumn of 2014 work has been performed by students to implement a biomedical algorithm on the SHMAC platform. In the spring a floating point software version (C language) was ported to run on an ARM processor with floating point emulation. At the same time, parts of the code were moved into a fixed point hardware accelerator. In the autumn the code was converted to a complete fixed point version and a hybrid fixed point/floating point version to be run on an ARM instruction set simulator. In parallel, two different hardware floating point units have been developed for the SHMAC platform. An interface has also been defined for integration of accelerators on the SHMAC platform.

Choice of floating point or fixed point implementation, including choice of accuracy is important for the performance, energy consumption and design complexity. Selection of general floating point accelerators and/or more application specific accelerators also strongly influences the same parameters.

The assignment builds on the work performed during the autumn of 2014. The main task will be to investigate performance, energy and area usage on the SHMAC platform for different implementation alternatives related to complete floating point, complete fixed point, and hybrid solutions. This should include use of general floating point units as well as application specific accelerators. Accelerators for efficient fixed point variable scaling can also be considered.

Supervisor: Per Gunnar Kjeldsberg
Co-supervisor: Donn Morrison

Abstract

Biomedical applications are becoming more and more important as they can improve the life conditions of millions of people around the world. Implementing them on low power embedded systems is a very challenging task as many among them demand numerous signal intense calculations. A major part of the epilepsy prediction algorithm proposed by Iasemidis et al. [2] called *Short Term Maximum Lyapunov Exponent* belongs to this category and comprises the study subject of this thesis.

The algorithm is ported to be executed on the single-ISA Many-core Computer (SHMAC) developed at NTNU, which is an evaluation platform for studying heterogeneous, power constrained systems. Different software versions of the algorithm (floating-point, fixed-point and hybrid), written in C language, are compared to each other and the most suitable ones are profiled and considered for further investigations. Corresponding hardware modules that implement the main bottleneck in each version are designed in VHDL hardware description language, and compared against each other. The most efficient module turns out to be an accelerator for the hybrid software version. This is selected to be further integrated within the SHMAC infrastructure, in order to evaluate its impact on the overall behavior of the algorithm and the target platform.

The performance, area usage, power and energy consumption as well as the energy efficiency are evaluated with or without the use of the hardware accelerator. Although that at the end of the thesis the application's real-time requirements were not met, the mixed (hardware/software) implementation that makes use of the accelerator, turns out to be 66% faster and 88% more energy efficient compared to the corresponding pure software implementation. Considerations about further modifications that can allow real-time performance are also discussed.

Preface

This thesis is submitted to the Norwegian University of Science and Technology in partial fulfillments of the requirements for the European Master in Embedded Computing Systems (EMECS).

Due to the fact that the algorithm is subjected to a non-disclosure agreement (NDA), extensive use of code snippets or other information that can be considered as confidential has been intentionally omitted.

Acknowledgements

At this point, I would like to thank my supervisors Per Gunnar Kjeldsberg and Donn Morrison for their guidance, remarks as well as for the proofreading of the current thesis. Furthermore I would like to thank the EMECS consortium for accepting me to the programme, and giving me the opportunity to gain valuable skills, competencies and a lot of memories over the past two years. Last but not least, I would like to thank “the usual suspects”; if they ever bother with reading this thesis they should know who they really are!

Contents

List of Abbreviations

A/D Analog to Digital

APB Advanced Peripheral Bus

ALU Arithmetic Logic Unit

ASIC Application Specific Integrated Circuit

ASU Arizona State University

CARD Computer Architecture and Design

CAS Circuits and Systems

CPU Central Processing Unit

DMA Direct Memory Access

DSP Digital Signal Processing

EDP Energy-Delay Product

EECS Energy Efficient Computing Systems

EEG Electroencephalograph

FPGA Field Programmable Gate Array

FPU Floating Point Unit

FSM Finite State Machine

GPU Graphics Processing Unit

HW Hardware

I/O Input/Output

IDI Department of Computer and Information Science

IET Department of Electronics and Telecommunications

ISA Instruction Set Architecture

LSB Least Significant Bit

LUT Look-Up Table

MPSoC Multi-Processor System on Chip

MSB Most Significant Bit

NDA Non-Disclosure Agreement

NoC Network on Chip

NTNU Norges Teknisk-Naturvitenskapelige Universitet

RAM Random Access Memory

RISC Reduced Instruction Set Computer

SHMAC Single-ISA Heterogeneous Many-core Computer

SIMD Single Input Multiple Data

SoC System on Chip

STLmax Short-Term Maximum Lyapunov Exponent

SW Software

XOR Exclusive or

XPE Xilinx Power Estimator

Chapter 1

Introduction

Epilepsy is one of the most common neurological brain disorders that affects approximately 1% of the global population [1]. It is characterized by sudden seizures that can last from seconds to minutes and which cause temporary disturbance of the brain functions. Such disturbances make the patient technically incapable of functioning normally and that has a severe impact on the patient's quality of life. One way to deal with this problem is by predicting the time points that an epileptic seizure is going to happen and warn the patient about the oncoming seizure, so that he/she will take measures to avoid the seizure (eg. take medication, change environment, stop doing the current activity etc.). Therefore it would be beneficial for the patients to have a seizure prediction system available anywhere and anytime. Such a system could drastically improve their quality of life, and thus the capabilities of portable (ideally implantable) systems with battery life as long as possible should be explored.

The implementation of such an application on an embedded system is a really challenging task since many system oriented requirements such as cost, performance, power consumption and size have to be met at the same time. The epilepsy prediction algorithm proposed by Iasemidis et al. [2], which is the object of this thesis, is an algorithm that requires complex calculations of EEG signals that capture the brain activity. Such calculations can be performed without difficulties on conventional computers such as desktop computers because this category of computing systems can easily bear the corresponding computational cost. However, when they have to be performed on low power embedded systems, the performance requirements are very often violated due to the insufficient computational power and therefore the overall system has to be modified.

The SHMAC platform [3] developed at NTNU is a very promising platform for studying and developing methodologies related to the mapping of various applications on embedded systems. This is because of the versatile nature of this platform; it can be configured with different components that affect the processing power, the memory availability, the interconnection infrastructure, the consumed power and many other system parameters that affect the behavior of the platform. Furthermore, the SHMAC platform belongs to the class of the so called heterogeneous computer architectures.

This class of computing systems tries to combine a variety of computing elements with different performance and power characteristics into a single architecture. The reason behind this is that some applications can benefit from some specific processing elements, while other applications can benefit from other processing elements. The gain in each case can be high

performance; in terms that the application can be executed faster on a certain processing element rather than on another one, high energy efficiency; in terms that the energy consumed during the execution of an application on a processing element is lower than if the application would be executed on a different processing element, and many other metrics. Performance and energy efficiency are however of vital importance since they affect directly the system requirements.

This thesis describes the implementations of a major part of the epilepsy prediction algorithm proposed by Iasemidis et al. [2], called *Short Term Maximum Lyapunov Exponent* (*STLmax*) on the SHMAC platform. This is done by evaluating different software implementations of this algorithm on the target platform, designing and integrating within the SHMAC platform application specific hardware modules as well as evaluating the performance, power consumption and energy efficiency of the overall system after the inclusion of the additional hardware modules.

1.1 Heterogeneous Systems

Central Processing Units (CPUs) have been the cornerstones of modern computing systems. The purpose of their existence is to provide an effective and efficient way for software implementations, and therefore the flexibility and ease of use that they offer can hardly be compared to other electronic components. The evolution in this field is highly driven by the so called “Moore’s Law” [4] which mentions the fact that the number of transistors of an integrated circuit doubles approximately every eighteen months. On the one hand, the continuously increasing number of transistors has offered massive increase in the performance of modern CPUs, but on the other hand this performance increase comes with the cost of continuously higher power consumption. Figure 1.1 [5], summarizes the evolution of performance, power consumption as well as the main factor that has been widely used over the past decades in order to keep the ascending trend of performance; the clock rate.

In Figure 1.1 (a), one can observe that over the past decades, processor performance has increased exponentially. The main source of this exponential increase has been the clock rate increase (also known as frequency scaling), as it can be seen in Figure 1.1 (b). The continuously increasing frequency, in combination with the continuously increasing number of transistors dictated by Moore’s Law has dramatically affected the power consumption which has also increased exponentially. The more transistors exist in an integrated circuit, the more power they consume when they all have to operate at the same time, and if they have to operate at a higher frequency than before, the power consumption will be even higher.

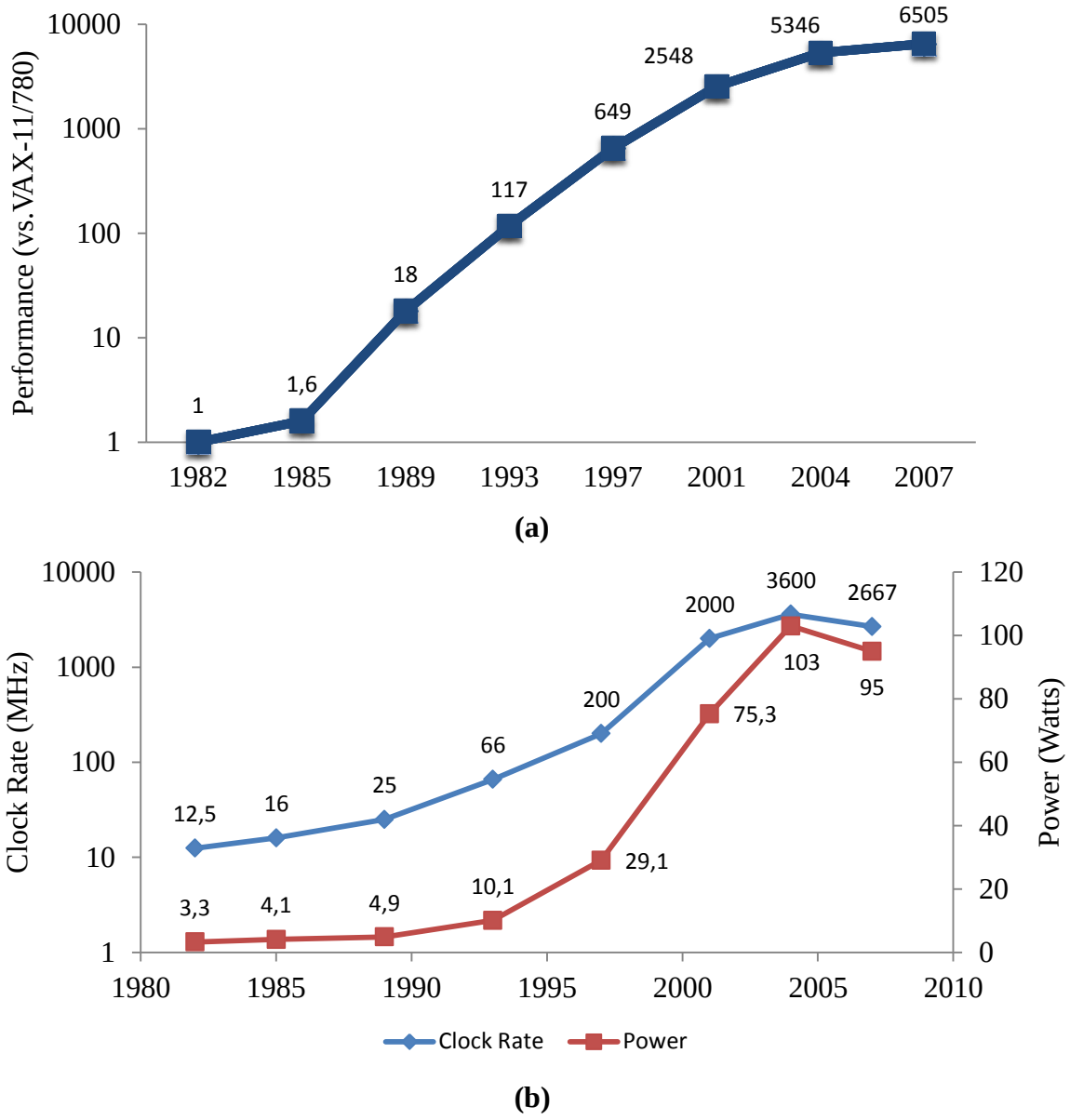


Figure 1.1: (a) Growth in processor performance (reproduced from data in [5]), **(b)** Clock rate and power variation for several Intel processors [5].

This comes as an effect of the breakdown of *Dennardian* scaling [35]. In *Dennardian* scaling, both the dimensions of the transistors and the supply voltage are scaled so that the consumed power will be the same as initially. Although this had been feasible in the past, in modern technologies (ones developed after 2005 [6]) it appears that this cannot continue anymore. Table 1.1 [35] summarizes the main differences between the *Dennardian* (that applies to older technologies) and the *post-Dennardian* (that applies to modern technologies) scaling.

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

Table 1.1: Dennardian vs post-Dennardian scaling [35].

In technologies characterized by Dennardian scaling, given a scaling factor S between two successive technologies (i.e. S=1.4), the number of transistor increases by S^2 while the supply voltage can be reduced by the same factor. This maintains stable the power consumption. In technologies characterized by post-Dennardian scaling, the number of transistors increases by a factor of S^2, however the supply voltage cannot be scaled by the same factor and this results in more power-hungry integrated circuits.

This is the cause of the Dark Silicon Effect [6], which gets more and more pronounced as technology scaling progresses. The Dark Silicon Effect means that given a fixed power budget, not all of the transistors in an integrated circuit can be powered at the same time. Most of the modern electronic devices are power limited; desktop and server computers are limited by the power their power supply can provide them, while this phenomenon is more intense in embedded applications that have to be powered by batteries, energy harvesting or other limited sources.

To keep up the performance increase and overcome the obstacles caused by Dark Silicon Effect, symmetric multicore processors [7] appeared in commercial applications during the mid-2000s. Symmetric multicore processors are processors that consist of two or more identical processing cores that exploit the spatial parallelization. The main advantage of this architecture compared to earlier single core architectures is that instead of having a single core with very high clock rate and power consumption, symmetric multicore processors can distribute the computational load to several processing cores that are able to operate at lower frequencies and therefore they consume less power. This can result into significant performance and energy gains.

Another way to optimize a processing unit with respect to performance and energy efficiency is the utilization of heterogeneous architectures [7]. This class of computing systems extends the idea of spatial parallelization introduced by symmetric multicore processors. Instead of using multiple identical processing cores, heterogeneous systems make use of a variety of different processing cores, with different performance and power characteristics. Figure 1.2 illustrates the concept of heterogeneous architecture [34].

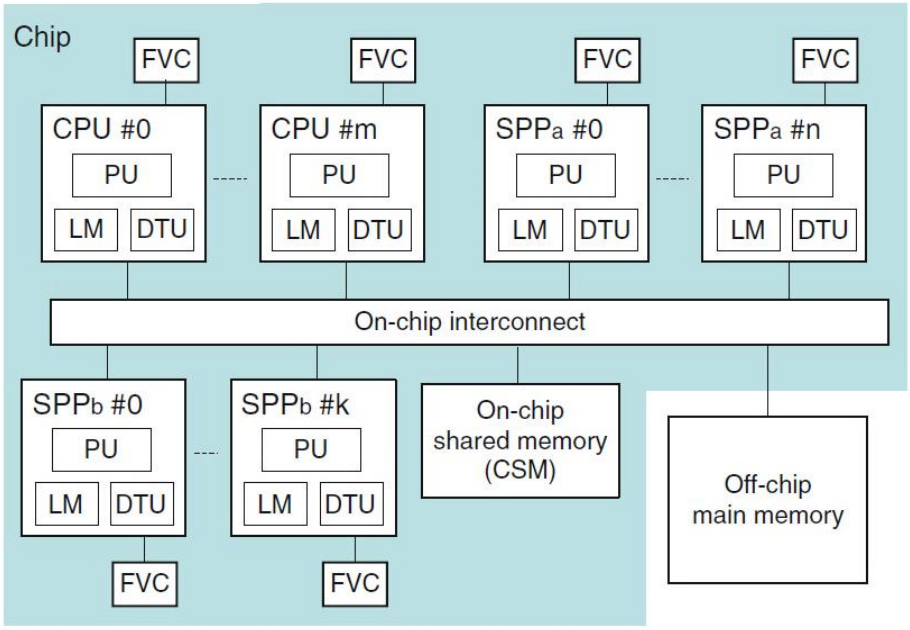


Figure 1.2: Heterogeneous architecture example [34].

The system illustrated in Figure 1.2 consists of *m* CPUs that include the main processing unit (PU), local memory (LM) and a data transfer unit (DTU) that handles the data transfers between each CPU and the main memory. In the same system there exist *n+k* special purpose processors (SPP) that they are configured according to the needs of each system (i.e. telecommunication system, media streaming, graphics processing etc.) as well as on-chip and off-chip memory units. Furthermore, each CPU and SPP has a dedicated frequency and voltage controller (FVC). This component can manipulate the frequency and the voltage and consequently the power of each corresponding core, allowing the system to achieve low on-chip power consumption. If for instance at a certain time point *CPU#0* is idle and SPP_a *#0* has to perform a demanding task, the first FVC can completely deactivate *CPU#0* while the latter FVC can increase the frequency of SPP_a *#0,* allowing it to execute its demanding task faster.

The adoption of such architectures results however into more complex systems than before, and this increased complexity has raised two main questions that have to be answered. What kind of hardware should be designed to realize heterogeneous systems and then, given a heterogeneous system, how should software applications be developed in order to take advantage of its heterogeneous architecture?

The SHMAC platform [3] is a heterogeneous platform developed within the Energy Efficient Computing Systems (EECS) initiative [8] at NTNU. Its main goal is to investigate issues and propose solutions related to both hardware and software perspective of heterogeneous systems. A very important part within its scope is the mapping of certain applications on this platform. Besides trying to answer the previous posed questions about how hardware and how software should be developed for heterogeneous platforms, application mapping on platforms

like SHMAC could also provide many insights about the advantages of heterogeneous architectures over conventional ones.

The algorithm mapped in this thesis is a major part of the epilepsy prediction algorithm proposed by Iasemides et al. [2]. The results of this algorithm are very promising for the treatment of epilepsy and the nature of this problem demands that the algorithm should be able to run on low power embedded systems. This could actually improve the quality of life of people who suffer from epilepsy, since epilepsy can take place anywhere and anytime. Low power embedded systems is a class of electronic systems made to function anywhere and anytime, and their behavior on this application can be modeled on the SHMAC platform. The proper combination of hardware and software that meets the applications requirements needs to be investigated and verified with respect to power consumption, and this is the main goal of this thesis.

1.2 Thesis Outline

Chapter 2 contains the necessary background along with the previous work that has been performed. The background consists of the biomedical algorithm, computer arithmetic, the SHMAC platform and how computer arithmetic is supported on it, as well as theory related to hardware accelerators and the 'two-process' design method that was adopted during the accelerator design.

Chapter 3 contains the high level mapping of the biomedical algorithm on the SHMAC platform. This includes the porting of the different software versions on the target platform, the profiling procedure in order to identify the main bottlenecks and finally the hardware/software partitioning, during which, the basic blocks that should be moved into hardware accelerators are determined.

In Chapter 4, the design of the hardware accelerators is discribed, their verification process, the comparison between them in order to determine which one is worth further considerations and the procedure of integrating the most suitable accelerator on the SHMAC platform.

Chapter 5 discusses the results obtained after the algorithm was executed along with the accelerator and compares them to the corresponding results of the pure software execution of the algorithm. The results include the metrics of performance, area usage on the FPGA, power consumption, energy consumption as well as energy efficiency.

Finally, Chapter 6 presents all the conclusions drawn throughout the duration of this thesis and moreover it discusses some further aspects that could be considered for future work.

1.3 Main Contributions

- Evaluation of different software versions (floating-point, fixed-point and hybrid) of the *STLmax* calculation algorithm on the SHMAC platform.
- Design of two different hardware accelerators in order to evaluate the suitability of fixed or floating-point arithmetic in the current application.
- Design of a VHDL package for performing operand scaling in 32 and 64-bit fixed-point applications.
- Integration of the most suitable hardware accelerator within the SHMAC platform.
- Reduction of the total execution time by 66% compared to the corresponding pure software implementation.
- Energy-efficiency improvement by 88% due to the addition of the accelerator.
- Suggestions of further considerations about meeting the application's real-time requirements and improving various accelerator oriented applications on SHMAC.

Chapter 2

Background and Previous Work

This chapter contains the necessary background related to the work that follows in the next chapters. More specifically, the epilepsy prediction algorithm and its particular subpart that is studied in this thesis are described first, and then its implementation related issues are discussed later. This includes the necessary theory about computer arithmetic, the SHMAC platform, as well as some theoretical aspects regarding hardware accelerators and a specific hardware design approach (two-process method) which form the basis of the current implementation. Last but not least, the previous work that has been performed is discussed in the last section of this chapter.

2.1 The Epilepsy Prediction Algorithm

There are numerous research programs worldwide that aim to the prediction of epileptic seizures. One approach is by monitoring the brain activity of a patient though EEG (Electroencephalograph) recordings, and then performing further processing on the EEG in order to get an indication about a possible seizure that will take place in the future. The algorithm proposed by Iasemidis et al. [2] makes use of this hypothesis, and its results are promising for the treatment of epilepsy. As many of the algorithms in this field, it utilizes the EEG recordings for the prediction of an epileptic seizure. However, the processing of the EEG in order to extract features relative to seizures is not a trivial task. Classic metrics that are widely used in signal processing, such as the mean value or the standard deviation of a signal, tend to be insufficient for this application. As one can observe in Figure 2.1, utilizing the mean value over time (instantaneous mean value) or the standard deviation over time (instantaneous standard deviation) results into something with inconsistent structure [2]. This means that is very difficult to detect similarities between all different patients by utilizing these metrics. It would seem reasonable for someone to assume that the structure of these signals would have many similar features; however reality turns out to be completely different. For this reason, it is necessary to look for other metrics that show a consistent structure.

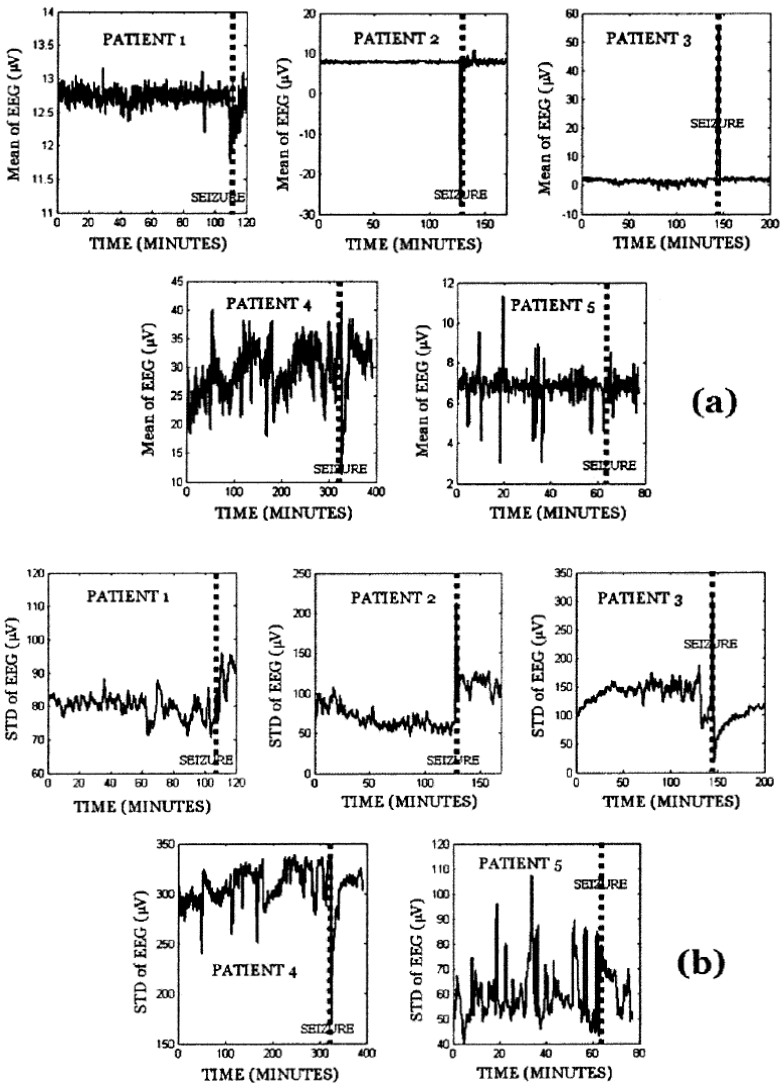


Figure 2.1: (a) The mean value over time of an EEG signal for five different patients, **(b)** the standard deviation over time, of the same EEG and the same patients. The vertical dashed lines denote the seizure onset (ictal period) [2] .

Due to the inconsistency of the EEG signals Iasemidis et. al. [2] introduced the utilization of the Short Term Maximum Lyapunov Exponents (*STLmax*).The *STLmax* is a special case of the Lyapunov Eponent. The Lyapunov exponent is a metric that measures the creation or destruction of information in chaotic systems or signals (in bits per seconds) [9]. In the case of *STLmax*, the Lyapunov exponents are calculated within short time frames (thus short term). Taking into account that the EEGs are obtained from the human brain, which can be considered as a highly chaotic system [10], the *STLmax* metric seems to perform very well for this kind of applications. As it can be seen in Figure 2.2, the *STLmax* exponents change consistently across

five different patients. For all the patients, it can be observed that during the time interval before the seizure (preictal period) the *STLmax* has always smaller value than right after the seizure (postictal period), and it gets its minimum value during the seizure onset (ictal period) [2].

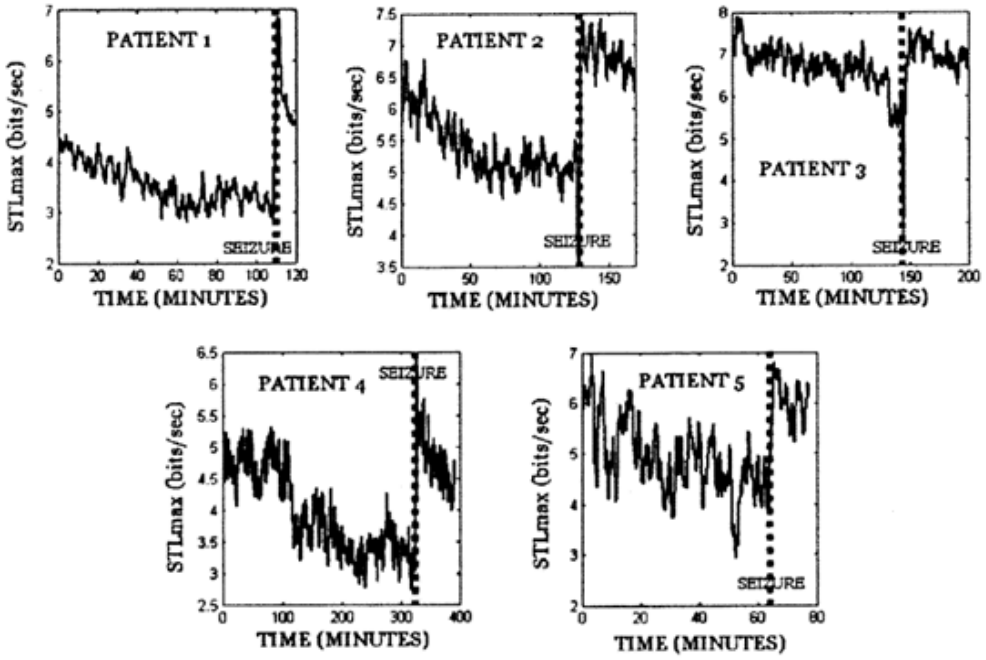


Figure 2.2: STLmax values over time, for the same EEGs and patients as in Figure 2.1 [2].

The epilepsy prediction algorithm by Iasemidis et al. [2] begins with converting the multichannel EEG recordings into a corresponding multichannel *STLmax* time series. The next step is the so called dynamical entrainment. During dynamical entrainment, the stability of each cortical site is quantified [2] according to the temporal and spatial dynamics of the brain. The temporal dynamics are already captured in the *STLmax* time series, since each *STLmax* value has been calculated within a relatively small time window (10.24 seconds in this case). The spatial dynamics are defined by the *STLmax* values on the different electrode sites (32 different electrodes in this case) and they illustrate the assumption that when a similar transition is observed within the *STLmax* time series of different electrodes, then the STLmax values right before the transition are expected to be the same for the different electrodes. This assumption is partly correct since it holds for a subset of different sites (electrodes) and not for all of them. The sites for which the assumption holds are called critical sites [2], and thus they need to be determined. In order to determine the critical sites, Iasemidis et al. [2] adopted the pair-T statistics metric [9] which determines if the dynamic entrainment has statistical content. The quantity that gets evaluated according to the T-index is the difference between the means of the STLmax values at two different sites. The set up for this application includes 60 *STLmax* values

(the *STLmax* values within a 10min time frame) from each site and the statistical test of them at a 0.01 statistical significance level. Given the *STLmax* values L_i and L_j of two different sites respectively, the T-index is calculated according to the following equations:

$$L_i^t = \left\{ STL\max_i^t, STL\max_i^{t+1}, ..., STL\max_i^{t+59} \right\} \quad (1)$$

$$L_j^t = \left\{ STL\max_j^t, STL\max_j^{t+1}, ..., STL\max_j^{t+59} \right\} \quad (2)$$

$D_{ij}^t = L_i^t - L_j^t = \left\{ d_{ij}^t, d_{ij}^{t+1}, ..., d_{ij}^{t+59} \right\}$, where d_{ij}^t is the difference: $STL\max_i^t - STL\max_j^t$ (3)

$$T_{ij}^t = \frac{\overline{D}_{ij}^t}{\frac{\sigma_d}{\sqrt{60}}} \quad (4)$$

where $\overline{D}_{ij}^t$ and σ_d are the mean value and the standard deviation of D_{ij}^t respectively.

Statistical significance level equal to a = 0.01 means that the electrode sites are disentrained if $T_{ij}^t > T_{\frac{a}{2},59} = 2.662$. Another threshold value at a 0.00001 statistical significance level ($T_{\frac{a}{2},59} = 5$) is also utilized. The way in which the thresholds are determined, is explained in details in [1] and [2]. These two threshold values are used to detect a dynamic transition which is used as an indication for a possible future seizure. Figure 2.3 illustrates this concept.

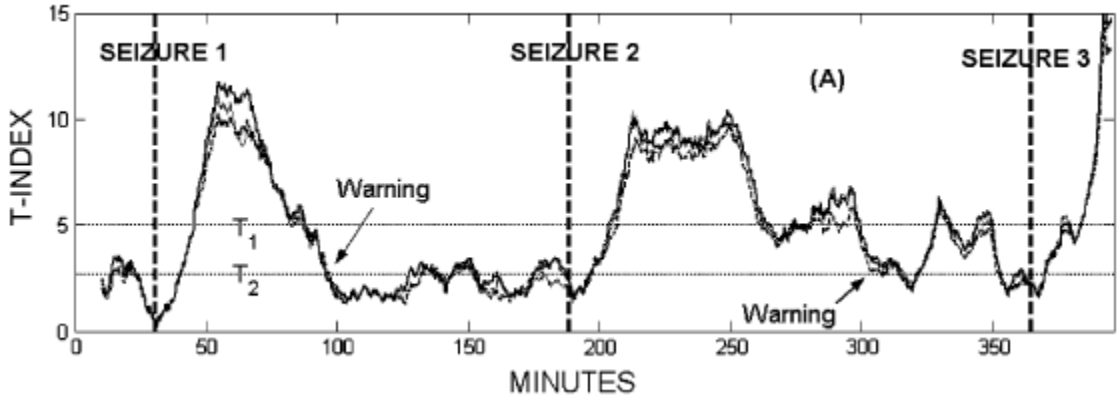


Figure 2.3: T-index values with T_1 and T_2 values. The vertical dashed lines denote the seizures onsets [1].

While the T-index crosses the T_2 value and is lower than T_1, the electrode sites are disentrained. Once it crosses T_1 it indicates that after this point we should start checking for a

dynamic transition in a future time point. If now the T-index crosses T_2, this means a dynamic transition and a warning of an oncoming seizure in issued.

One problem that arises in the previous analysis of the algorithm is the selection of the critical sites that will take part in the calculation of T-index. This procedure is performed after the occurrence of each seizure, by selecting *k* critical sites. Since there are *n* electrode sites (32 in this case) there are $\binom{n}{k}$ different combinations of *k* sites. For a value of *k* relatively small ($k \leq 6$), it turns out that an exhaustive search can be performed in order to identify the critical sites [1]. The *k* sites selection turns out to be the *k* sites that were most entrained 10 minutes before the seizure and disentrained after the seizure [1]. This procedure is repeated after the occurrence of each seizure.

A simplified overview of the way according to which the epilepsy prediction algorithm functions is demonstrated in Figure 2.4. In this figure, one can consider that the EEG signals are recorded via some kind of sensors (i.e. EEG electrodes attached to an A/D converter), they are converted into corresponding STLmax time-series, and then the resulted STLmax time-series are forwarded to the "Seizure Prediction" stage, which extracts the T-index values and performs the comparisons to T_1 and T_2 values in order to generate an alarm.

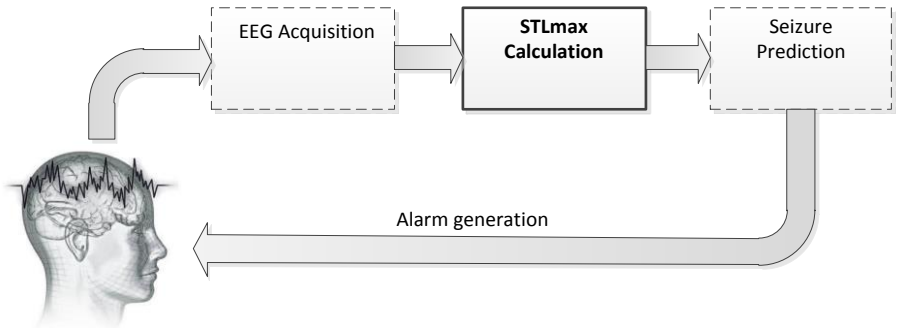


Figure 2.4: Simplified overview of the epilepsy prediction algorithm (the block of STLmax calculation stage that is studied in this thesis is highlighted with continuous line).

2.1.1 The Maximum Lyapunov Exponent

From the analysis of the epilepsy prediction algorithm so far, it should be clear that an essential part of it is the calculation of the *STLmax* time-series. For the *STLmax* calculation, the algorithm proposed by Wolf et al. [9] is adopted. This requires the expansion of the initial signal into an *m*-dimensional phase space with delay coordinates [9]. For this application the *m*

parameter is selected to be equal to 7 [1]. An expanded signal in a *7*-dimensional space can be considered as a set of points with 7 variables. This can be clear by generalizing the illustration of Figure 2.5 into 7 dimensions. Figure 2.5 (a) shows a point on the two dimensional Cartesian plane, which is defined by the *x* and *y* coordinates. In Figure 2.5 (b), it can be observed a point in the three-dimensional space. In this case the point is defined by the *x*,*y* and *z* coordinates. This observation (of defining a point according to some coordinates) can be generalized for more than three dimensions. Unfortunately it is impossible to demonstrate it on a piece of paper, and so only the cases of 2d and 3d spaces are illustrated here.

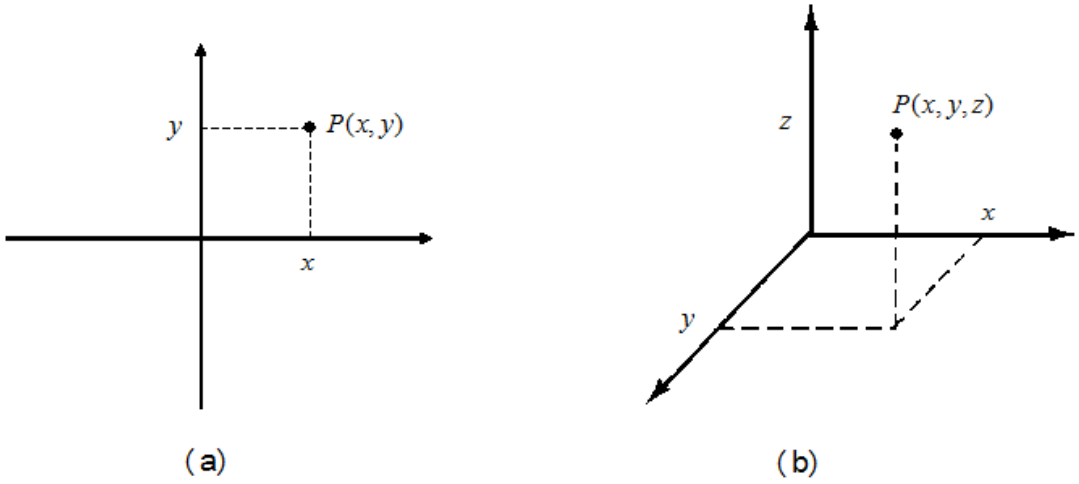


Figure 2.5. (a) A point defined in the two-dimensional space, **(b)** a point defined in the three-dimensional space.

The seven variables in the case of the Lyapunov exponent are the values of the original signal. If the original signal is $x(t)$ (function of one variable), then a point in the *m*-dimensional space (in this case, *m* is equal to 7) can be defined as:

$$P(x_0, x_1, x_2, x_3, x_4, x_5, x_6)$$

where the coordinates $x_0, x_1, \ldots, x_6$ are equal to $x(t), x(t+T), \ldots, x(t+(m-1)T)$ respectively. *T* quantity is the delay time which depends on each application; for this application the *T* variable is selected to be equal to *4 seconds* [1]. All these coordinates are values of the original signal $x(t)$. Therefore, the expanded signal can be considered as a set of points:

$$P(x(t), x(t+T), ..., x(t+(m-1)T))$$

An initial point $P_0(x(t_0), x(t_0+T), ..., x(t_0+(m-1)T))$ is selected at t_0 and its nearest neighbor (neighbor with the smallest Euclidean distance) is detected. The distance between these points is denoted as $L(t_0)$. At a later time point t_1, this distance will have evolved to a new value denoted as $L'(t_1)$. The propagation time $t_1 - t_0$ must be small enough so that the time difference between the examined points is sufficiently small [3]; this is also illustrated in Figure 2.6. At this time point, a new point that satisfies the two following criteria is searched:

a) Its distance $L(t_1)$ from the initial point is small
b) The angle θ between the evolved and the replacement point is small

If there is no such point, the initially selected point is retained. This procedure is performed for the whole signal (EEG time series in our case), and after this, the Maximum Lyapunov Exponent value is calculated by the following equation:

$$\lambda_1 = \frac{1}{t_M - t_0} \sum_{k=1}^{M} \log_2 \frac{L'(t_k)}{L(t_{k-1})}, \text{ where M is the number of replacement steps.} \quad (5)$$

The calculation of the Maximum Lyapunov Exponent is illustrated in Figure 2.6. Starting from an initial point $x(t_0)$ at time t_0, the nearest point $x(t_1)$ at time t_1 is identified. If there exist another point at time t_1 that satisfies the previously mentioned criteria (a) and (b) better than $x(t_1)$, then this point replaces $x(t_1)$.

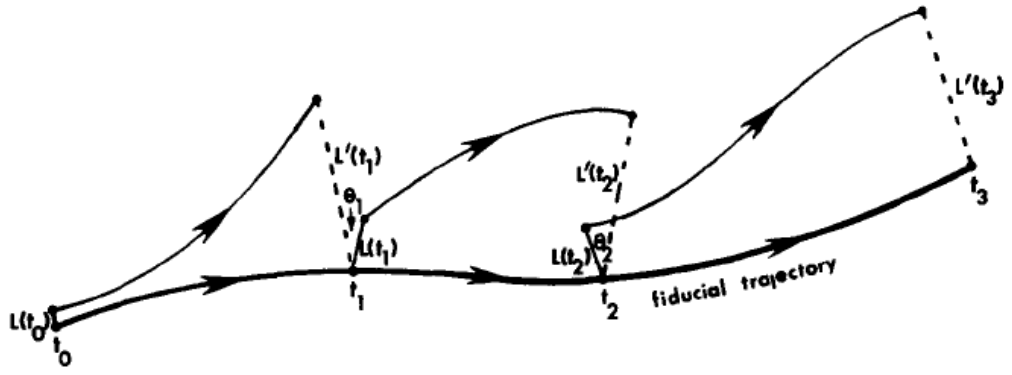


Figure 2.6 Illustration of the "evolve and replace" technique for the calculation of the largest Lyapunov exponent [3].

2.2 Computer Arithmetic

In computers and digital systems, numbers have to be represented by a sequence of binary digits (bits). Computers offer a finite number of bits and the user has to find a way to represent physical quantities in these bits. For this reason two major representations have been proposed and evolved through the past years. These are the fixed point and the floating point representations, the characteristics of which are discussed in the following section.

2.2.1 Fixed Point Representation

In fixed point representation, the quantities that define a number are the sign, the integer part of the number and the fractional part of it. The concept of fixed point representation can be illustrated in Figure 2.7 [15]. In this figure, it can be observed that given a finite number of bits (32-bits in this case) one bit can be allocated (the most significant bit) for the sign of the number, *m*-bits (8-bits in this case) for the integer part of the number and *n*-bits (23-bits in this case) for the fractional part of this number. Therefore the number is represented in Qm.n format [15]. In case where the numbers are not signed, the sign be can be omitted and that bit can be used as an additional bit for the integer or the fractional part of the number. For simplification reasons, the numbers here are represented in sign and magnitude representation, in which one bit is used to represent the sign of the number and the rest of bits are used to represent the magnitude of the number. Nevertheless, corresponding conclusions can also be drawn for other fixed-point representations (such as two's complements which dominates in modern digital systems).

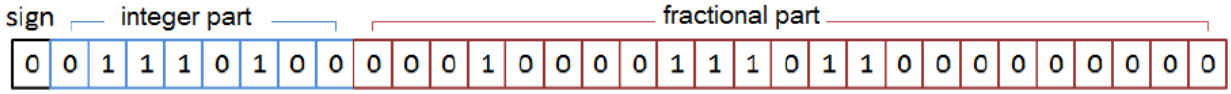


Figure 2.7: 32-bit fixed-point representation [15].

When a number is represented in fixed-point, it is necessary that the decimal point (the point which separates the integer part from the fractional part) is known either at compile time [16], or otherwise it has to be determined during runtime. Under no circumstances should it be determined by the compiler.

Determining the corresponding decimal value from the integer part of a binary number is straightforward since it can be found by summing all the necessary powers of 2 according to the following equation:

$$I = \sum_{i=0}^{m} a_i \cdot 2^i \tag{6}$$

In this equation, a_i are coefficients that determine whether a specific power of 2 is taken into account for the calculation of the integer part. For the number of the Figure 2.7 it turns out that the integer part is:

$$2^2 + 2^4 + 2^5 + 2^6 = 116$$

For the fractional part, reality turns out to be different. This has to be approximated by the sum of the negative powers of 2 as in the following equation:

$$F = \sum_{i=1}^{n} a_i \cdot 2^{-i} \tag{7}$$

Thus the fractional part of the number in Figure 2.7 is:

$$2^{-4} + 2^{-9} + 2^{-10} + 2^{-11} + 2^{-13} + 2^{-14} = 0.6610107422$$

and the final number is 116.6610107422.

From the above analysis, it is clear that there are certain limitations when a number is represented in fixed point format. First of all, the integer part is represented by *m* bits. This means that the maximum value (in decimal system) that can be represented with these bits is $2^m - 1$ and the minimum value is equal to -2^m, provided that the numbers are encoded in two's complement format (numbers in two's complement form are very convenient for performing mathematical operations between signed numbers in hardware). Any attempt to represent a value out of this range will consequently lead to overflow. The fractional part of the number consists of *n*-bits, therefore the smallest fractional quantity that can be represented is equal to 2^{-n}. This offers a resolution of 2^{-n} which means that any fractional quantity will be an integer multiple of it. For example, the range and resolution of $Q_{8.23}$ format (Figure 2.7) are [127,-128] and $1.192092896 \cdot 10^{-7}$ respectively. Besides the limitation in range and resolution, one can observe that there are visible tradeoffs between them. The range can be increased by using extra bits for the integer part but this will lead to a reduction in resolution and vice versa (taking into account that we have a constant number of bits). It is therefore necessary for someone who has to develop an algorithm in fixed-point arithmetic to evaluate the range of the variables in the algorithm and the required precision, before starting the implementation.

When fixed-point arithmetic has to be used for high level programming (eg. C/C++), then it can be considered that each number is represented in $k \cdot 2^{-q}$ format, where k and q are integers variables. Variable k determines an integer quantity, while variable q determines the position of the binary point. This can become clear by considering for instance the representation of the decimal value 12 within 5 bits. Since 12 is an integer quantity, it is not necessary to use

any decimal bits and therefore the use of $Q_{5.0}$ format is straightforward. In this case the decimal value 12 is represented as 01100, k is equal to the corresponding integer value of 01100, which is 12 and e is equal to 0 since there are not any decimal bits. If however the decimal value 12 would have to be represented in $Q_{4.1}$ format, then this would lead to the value 11000. This is because the first four bits on the left represent the integer part of the number ($1\cdot 2^3+1\cdot 2^2+0\cdot 2^1+0\cdot 2^0=12$) and the last bit represents the decimal bit (which in this case is 0). k now, is the integer value of 11000 which is 24 and *q* is equal to one. In the previous example, the values that the computer has in its memory are 01100 and 11000 respectively. The user on the other hand can manipulate these values to represent decimal numbers. The integer part can be selected by shifting the binary value *e*-positions to the right. The decimal part can be selected by shifting the binary value (5-q)-positions to the left. The same also holds for any available total number of bits. Shifting operations can be performed very fast in hardware, and for the Amber processor this can be performed in one clock cycle provided the barrel shifter that it has. This allows the programmer to use integer oriented variables (short, long, signed, unsigned etc.) for representing decimal values.

The same principle applies to the binary number of Figure 2.7. This can be considered as the integer number:

$$1\cdot 2^9+1\cdot 2^{10}+1\cdot 2^{12}+1\cdot 2^{13}+1\cdot 2^{14}+1\cdot 2^{19}+1\cdot 2^{25}+1\cdot 2^{27}+1\cdot 2^{28}+1\cdot 2^{29}=973.633.024$$

Thus *k*=973.633.024 and *q* is equal to 23 (the number of fractional bits in Figure 2.13). Shifting *k*, 23 positions to the right (arithmetic shift), results in the decimal value 116 (the MSB is reserved for the sign and therefore it is ignored during these calculations). Shifting *k*, 31-23=8 positions to the left and summing the negative powers of two (the MSB after the sign bit now corresponds to 2^{-1} and the LSB to 2^{-31}) results in the value 0.6610107422 as calculated earlier. When performing shifting operations it is important not to lose the sign bit, otherwise the calculated results will be wrong.

Performing mathematical operations in fixed-point format has been discussed in details in [26], and for this reason it is omitted here. Nevertheless, Appendix A contains all the necessary material that explains how such operations are performed.

2.2.2 Floating Point Representation

In fixed-point representation, the sign, the integer part and the fractional part of a number are utilized to represent the number. Another approach is to represent other features of this number. The most popular approach is the one presented in Figure 2.8 [15].

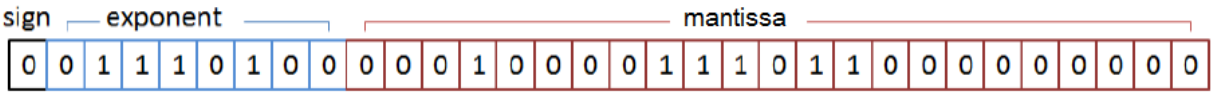


Figure 2.8: 32bit floating-point representation [15].

In the previous figure, it can be observed that instead of representing the integer and the fractional parts of a number, two other quantities can be used: the exponent and the mantissa [19]. By utilizing these quantities, a number is represented in the format:

$\alpha = (-1)^{sign} \cdot mantissa \cdot 2^{exponent}$, where the exponent can have both positive and negative values

The position of the binary point depends on the mantissa and the exponent and that's the reason why this representation is called floating. Different mantissas and exponents give different binary point position. For instance, if the mantissa is equal to 2 and the exponent is equal to 0, the decimal result is 2 (no binary point), while a mantissa equal to 2 and an exponent equal to -2 give the decimal value 0.5. The floating-point format which is the de facto format used nowadays, is defined by *IEEE Standard 754* [20]. According to this standard, floating-point variables can also have 64 or 128 bits, which allows higher dynamic range and precision. In the *IEEE 754* standard, the exponent (supposing that the exponent consists of n-bits) is represented as a biased fixed-point number. The exponent is given by the equation:

$$E = \sum_{i=0}^{n} e_i \cdot 2^i - Bias \text{, where } Bias = 2^{n-1} - 1 \quad (8)$$

For a single precision floating point variable defined according to *IEEE 754*, there is one sign bit, 8 exponent bits and 23 mantissa bits (32 bits in total). The *Bias* is equal to 127 and the exponent range is [-126 , 127]. It is therefore straightforward that the dynamic range in this case is much higher that the fixed point representation and the precision is also orders of magnitude higher. One extra advantage of the floating point format is that it allows special representations such as $\pm\infty$ and NaN (not-a-number).

Viewing it from a more abstract level, a programmer who develops an algorithm in a high level programming language, can define the variable as floating-point, and let the compiler do the rest (select the mantissa and exponent for each variable). An extensive description about how mathematical operations are performed in floating-point has been performed by Indergaard [32].

2.3 The SHMAC Platform

SHMAC is an ongoing research project at NTNU, and its main goal is to explore the capabilities of heterogeneous architectures with respect to performance and energy efficiency. This makes it a suitable evaluation platform, when the performance and energy efficiency of an application need to be evaluated. SHMAC adopts a tile-based architecture, which means that the system consists of several tiles, connected as a two-dimensional grid. The architecture of this platform is illustrated in Figure 2.9 [3].

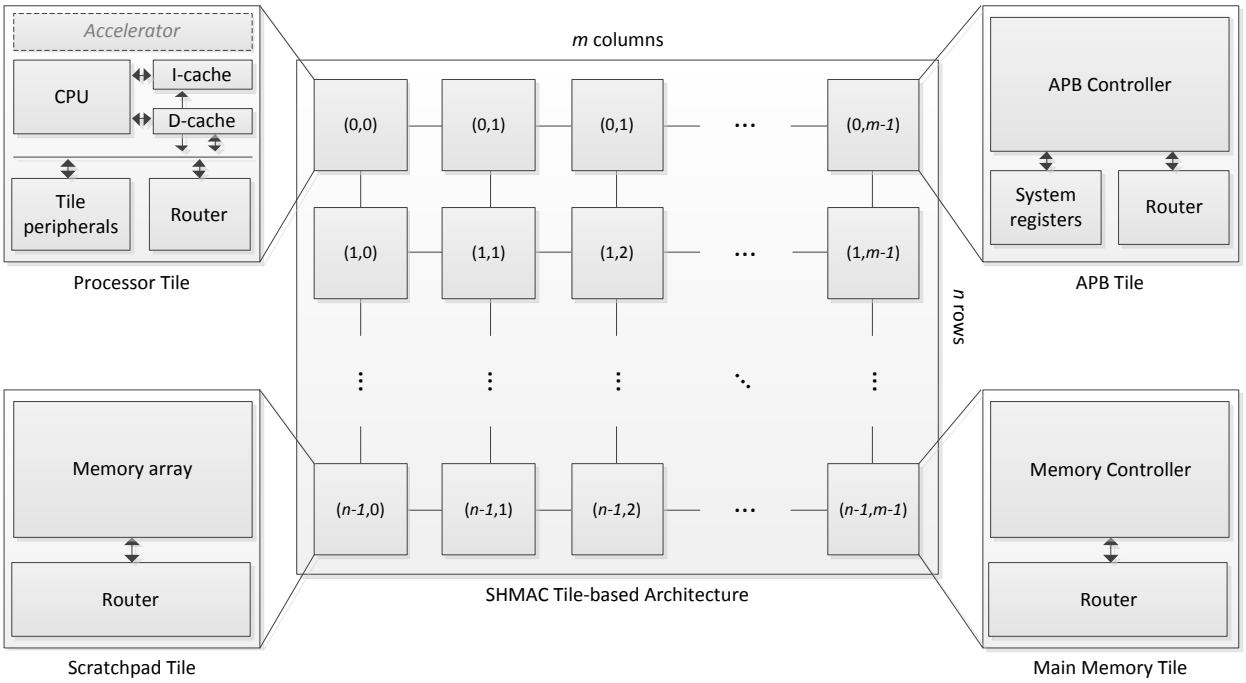


Figure 2.9 The SHMAC platform architecture [3].

In Figure 2.9, one can observe that the SHMAC platform consists of m×n tiles, where each tile can be configured as a system specific component. The setup of the platform is done on an FPGA device, which simplifies hardware development. The tiles communicate with their neighbors by using a network-on-chip (NoC) architecture, and the available tile configurations include currently the following options [3]:

- Processor Tile: it contains a processor unit, caches, peripherals and a router for communicating with other tiles. It also offers the option to be extended with additional accelerators.
- Scratchpad Tile: it includes on-chip memory and a router.
- Main Memory Tile: it contains a memory controller that gives SHMAC access to off-chip memory resources and a router.

- APB Interface Tile: this tile implements the Advanced Peripherals Bus (APB) slave interface which is necessary for the communication of the FPGA with the host system.
- Dummy Tile: this tile contains only a router and is used to fill remaining tiles when there are not enough resources available in the target FPGA.

The memory space of the SHMAC platform is depicted in Table 2.1 [33]. It consists of the *Exception Table*, the *Main Memory* that is located outside of the FPGA, the *Scratchpad Memory* that includes memory positions located inside the FPGA (provided that corresponding Scratchpad tiles exist in the FPGA), the *Tile Registers* that are memory positions addressed within a processor tile (i.e. addresses of the tile peripherals) and the *System Registers*.

Description	Start Address	End Address
Exception Table	0000 0000	0000 001F
Main Memory	0000 0020	F7FF FFFF
Scratchpad Memory	F800 0000	FFFD FFFF
Tile Registers	FFFE 0000	FFFE FFFF
System Registers	FFFF 0000	FFFF FFFF

Table 2.1: SHMAC memory space.

2.3.1 SHMAC Parent System

The SHMAC platform can currently be instantiated on two different system-on-chip (SoC) prototyping platforms. The first one is the ARM RealView Versatile platform, which includes the host system, a Xilinx Virtex-5 FPGA and 32MB off-chip RAM, while the second one is the ARM Versatile Express, which also includes the host system (a different one), and provides a Xilinx Virtex-7 FPGA and 4GB off-chip RAM. For the needs of this thesis, the ARM RealView Versatile platform was used. Figure 2.10 depicts the setup of the SHMAC platform, as used during this thesis along with the main parts of the development platform.

The Virtex-5 FPGA used in the the RealView Versatile platform (Virtex-5 XC5LVX330) [36] is an FPGA device fabricated at 65nm technology that aims for high performance applications. It contains 51.840 logic slices, 10.368 KB of on-chip RAM memory distributed as blocks at different points within the chip in several sizes, as well as 192 DSP48E slices. All these resources provide the developer a valuable asset for the implementation of complex digital systems such as multiprocessor systems on chip (MPSoC).

Figure 2.10: (a) The SHMAC set-up in the current thesis, **(b)** snapshot of the interior part of the RealView Versatile platform

The host system consists of a quad-core ARM11 processor running Linux OS. In the studied case it is only used to handle the required communication between the FPGA and external systems that are necessary for the function of the platform (i.e. the developer's computer), however in more advanced architectures it can be used along with the FPGA as one single multiprocessor system.

Communication between the host system and the FPGA is performed through an APB slave interface; this simply means that the host system acts as a bus master, while the FPGA as a bus slave. The communication between the host's computer and the RealView platform is realized via a USB to RS232 adapter. The RealView platform needs also to be connected to a local server located at NTNU in order to perform control operations on the host system; this is performed via an Ethernet connection. Communication between the FPGA and the off-chip RAM is realized via a RAM bus. Figure 2.11 illustrates the interconnection of the parent system with respect to the rest of the functional components. In reality, the RealView platform includes many other components, however for simplification reasons they are omitted.

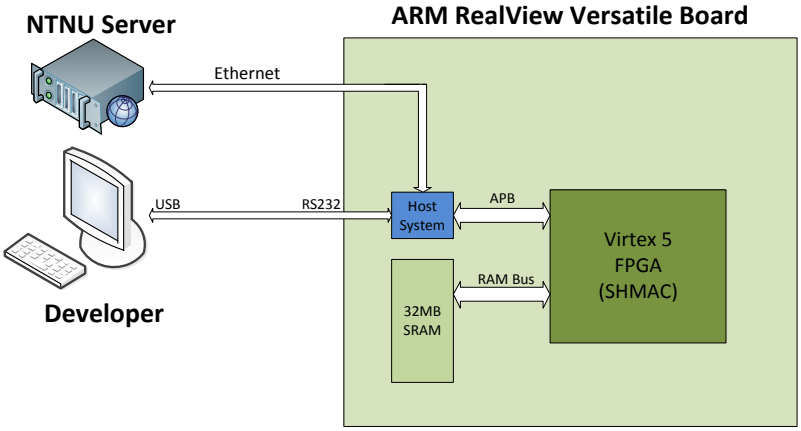


Figure 2.11: SHMAC interconnection.

2.3.2 SHMAC Processor Tile

When an application has to be executed on the SHMAC platform, this will take place on one or more processor tiles. The currently available processors are the Amber 25 processor [11] and Turbo-Amber [14]. Both of them are 32-bit RISC processors with five pipeline stages that support the ARMv2a ISA and run at 60 MHz (clock frequency when implemented on the RealView board). Figure 2.12 depicts a high level architecture of a SHMAC processor tile. The main part of this tile is the *Processor System* which includes the CPU, the main bus (Wishbone) as well as its peripherals (timer modules, the interrupt controller and the tile registers). Another important part is the router, which handles the communication between the tile and its neighboring tiles. There also exists a bridge that connects *Processor System* with the router.

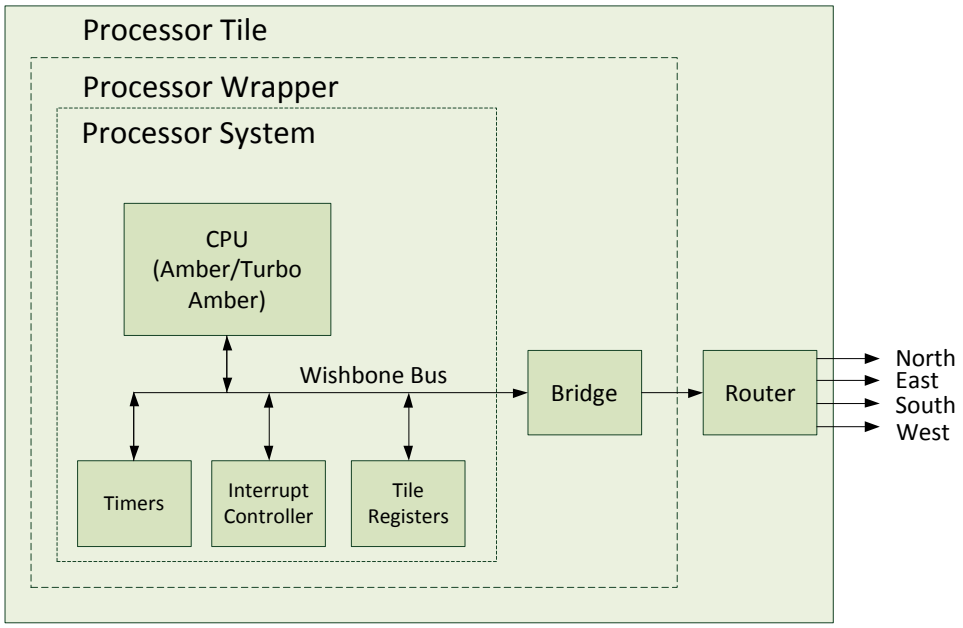


Figure 2.12: SHMAC processor tile.

- **Amber 25**

Figure 2.13 illustrates Amber 25's pipeline architecture. Register based instructions (except instructions involving multiplication and division) need one clock cycle to be executed. The same applies for load and store instructions. It is however necessary that there are no register conflicts and no cache miss occurrences, otherwise the processor has to be stalled until the proper data are fetched from the main memory and this will result to extra clock cycles per instruction.

The cache memory consists of two separate Level-1 caches; instruction and data cache [11]. Both caches are *n*-way associative (where *n* can be 2,3,4 or 8) with each way offering *4KB*

of memory. Consequently, the cache size can vary from *16KB* (*2×2×4KB*) to *64KB* (*2×8×4KB*). On the SHMAC platform, the user can select whether to have the cache activated or deactivated.

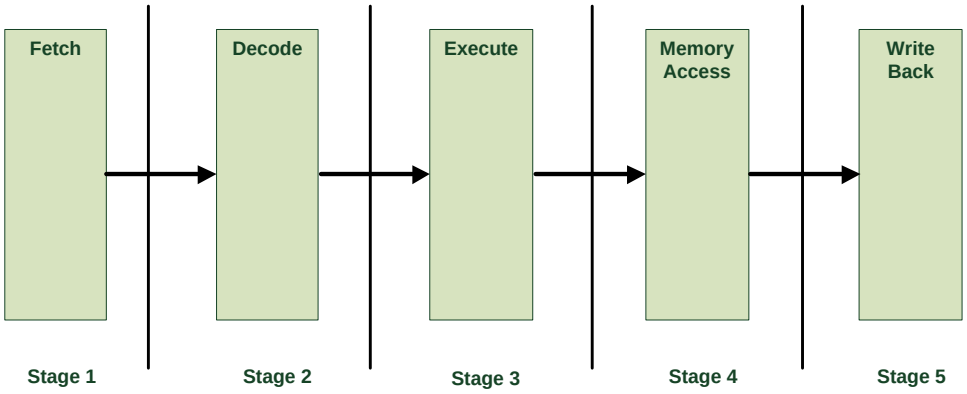


Figure 2.13: Amber's 25 pipeline architecture.

Multiplication is performed using the Booth's algorithm [12] which is a small but slow multiplier that takes 34 clock cycles for a multiplication of two 32-bits numbers [11]. At this point, it needs to be noted that a slightly modified version of Amber with a fast single cycle multiplier was available throughout the duration of this thesis as well. Division is not supported by the current instruction set and thus it has to be emulated by software [13], resulting in high latency. Another feature of the Amber processor is that it does not explicitly support floating point operations. This means that all operations that involve floating-point variables have to be emulated by software and as in the case of division the latency will be very high.

- **Turbo-Amber**

Turbo-Amber [14], is a high performance processor based on Amber 25. The only differences between them lie inside the *Fetch* and *Execute* stages of the pipeline. More specifically, Turbo-Amber's *Fetch* stage is designed in a superscalar way in order to support branch prediction and the *Execute* stage includes a different multiplier.

Branch prediction is a method used in microprocessor architecture that aims to performance increase by the proper handling of branch instructions (i.e. *if-then-else* statements). All the instructions that need to be processed by a microprocessor are usually fetched sequentially by the *fetch* unit illustrated in Figure 2.13. A problem that is caused by this sequential instruction fetching is that the next instruction is not known until the current instruction has computed it [14]. In architectures that include branch prediction, the *fetch* unit is designed so that it can predict the next instruction or if its prediction is wrong it can recover by fetching the proper instruction.

The Booth or the single-cycle multiplier of Amber 25 processor is in this case replaced by a 2-cycle multiplier inside the *Execute* stage. According to the designers, a single-cycle multiplier is not able to be integrated within the current processor because it violates the time constraints of the system [14] and therefore the 2-cycle one was adopted. Division and floating-point hardware support are also absent and thus they are also emulated by software; however Turbo-Amber has performance increase by 49% and area increase by 70%, compared to Amber 25 [14]. At the time of the current thesis, Turbo-Amber is available only on ARM RealView Versatile platform.

- **The Wishbone Bus**

The Wishbone bus [38] is an open source bus standard, used for the interconnection between components within the *Processor System*. The current bus uses 32-bits for the address, 128-bits for the data as well as some additional control signals and implements both a Master and a Slave interface. The master interface is used by the processor core, so that it can occupy the bus and start performing data transfers to/from the tile peripherals, other tiles or the main memory. All the peripheral components within the *Processor System* (timers, interrupt controller and tile registers) are connected via the slave interface. This implies that whenever the processor core needs to access them, it can select them and start performing read or write operations on them. Figure 2.14 [38] illustrates the Master/Slave interconnection interface on Wishbone.

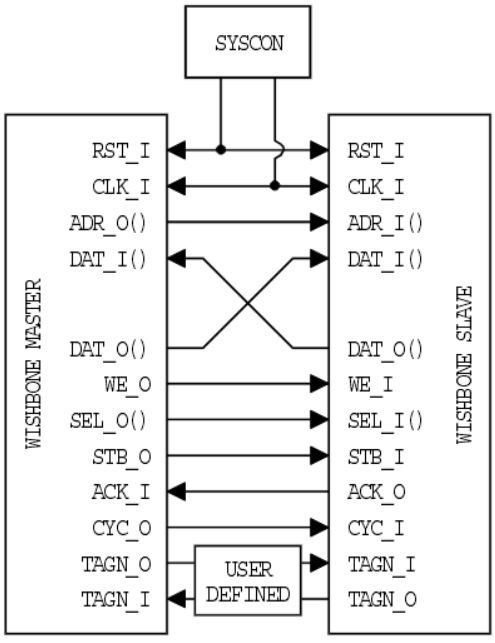


Figure 2.14: Wishbone's Master/Slave interconnection [38].

Several processor tiles can be placed within the FPGA device. This allows the concurrent execution of several applications as well as the multithreading implementation of a single application (on condition that the application can be parallelized). In the latter case, the application is broken down into several sub-processes (threads), that each of which can be executed on a different processor tile.

2.3.3 SHMAC Floating-Point Support

As discussed earlier, both Amber and Turbo-Amber do not support by default floating-point operations in hardware. Nevertheless, they implement the ARM ISA. All processors of the ARM architecture can support floating point operations with the following ways [28]:

1. Hardware coprocessor that executes floating-point instructions
2. Software floating-point emulation

The hardware floating-point coprocessor (or FPU) is a unit to which all the floating-point instructions are forwarded. An FPU for the Amber processor and the SHMAC platform has been integrated by Knutsen [29]. The main advantage of an FPU is its high performance compared to software solutions. An FPU takes usually 2 to 10 clock cycles for the execution of a floating point operation, while an equivalent software solution would take 50 to 100 clock cycles [30]. However, equipping Amber with an FPU comes with the cost of the extra silicon area that is needed for the FPU and thus higher dynamic power consumption. The static power in this case remains the same, provided that the system is synthesized in an FPGA. In an ASIC implementation, the static power would also increase with the addition of the FPU coprocessor.

Emulating floating-point operations in software can be performed by software routines that break down the floating-point operators into segments, and they manipulate each segment by using integer operation [30]. These software routines can be simple routines that simulate the behavior of an FPU or more complex library routines generated by the compiler [30]. The SHMAC platform currently utilizes the *SoftFloat* library [31]. A floating-point number according to the *IEEE 754* standard consists of three segments: sign, exponent and mantissa. For instance, a simplified version of the multiplication of two numbers would be to break down the numbers into these three segments, and then multiply the two signs (integer multiplication), multiply the mantissas (integer multiplication), add the exponents (integer addition) and finally reconstruct the result according to the produced segments. From the previous example it is clear that the overhead in this case (which was a very simplified version of a real situation) is significantly high. Table 2.2 summarizes the latency times of fixed and floating-point operations (of 32-bit variables) on the Amber processor when they are emulated by software as well as when they are executed on the FPU integrated on SHMAC by Knutsen[29].

Corresponding data for the Turbo-Amber or the Amber with the single-cycle multiplier were not available in the literature. Nevertheless for the latter two cases it can be estimated that

addition and subtraction have the same performance in all cases (fixed-point, software floating-point and hardware floating-point) as in Table 2.2 because of the identical ALU that executes these instruction. Multiplication in fixed-point and software emulated floating-point will be different (and definitely faster than Amber with the Booth multiplier) in each case according to the multiplier (single-cycle or two-cycle). Estimations about the division can hardly be made because it is unclear whether the multiplier is involved in the corresponding emulated operations. The floating-point performance in hardware is in any case the same as in Table 2.2, since the FPU coprocessor does not change.

Instruction	Fixed-point	Floating-point (SW)	Floating-point (HW)
Addition	1	59	52
Subtraction	1	59	52
Multiplication	34	193	52
Division	14 - 200	145	105

Table 2.2: Latency values in clock cycle of floating-point addition, subtraction, multiplication and division instructions on the Amber processor (enabled caches).

As one can observe in the above table, the FPU is in every case faster than emulating the corresponding operations in software, with the multiplication being benefited more than the others. However, the latency of this specific FPU is much higher than the fixed-point operations. The current FPU [29] is a 64-bit FPU. This requires that the overhead for loading and storing data in the FPU will be twice as in the case of using a 32-bit FPU. Furthermore the latency of the FPU (when executing the operations, without the load and store overhead); 20 clock cycles for addition, 21 for subtraction, 24 for multiplication and 71 for division [29], is also significantly high compared to fixed-point operations (except in the multiplication).

Integration of one of the FPUs designed by Indergaard [32] could potentially produce better performance results. Indergaard [32] investigated several FPU implementations for the SHMAC platform and it was found that all floating-point operations could be performed in a single cycle. This could result in better performance than the default fixed-point operations (considering that in fixed-point operations, multiplication takes 34 clock cycles and division takes 14-200 clock cycles), however at the current point of the SHMAC project, the compiler is not able to automatically produce co-processor instructions that are necessary for the FPU function and therefore the only option for someone to make use of the FPU is to write assembly code. Walstad [38], has included all the relevant assembly instructions inlined within C-functions that comprise a floating-point library for the SHMAC project. By making use of these functions, one can use the FPU, without having to manually code all the assembly instructions.

2.4 Hardware Accelerators

Hardware accelerated systems, is a typical paradigm of HW/SW co-design [24]. Adding hardware accelerators is an approach of customizing the target platform to the needs of the application. According to Wolf [24], using high end processors can be a very expensive option (both in monetary cost as well as power consumption) compared to breaking down the application into smaller parts and implement some of these parts in hardware accelerators. Breaking down an application into smaller tasks requires additional effort, however if these tasks can be scheduled in an optimal way, then meeting deadlines and real-time requirements becomes more and more feasible. The distinction between accelerators and co-processors must be denoted at this point. A coprocessor is an internal part within a CPU that executes instructions. The accelerator on the other hand, is a component that communicates with the CPU but is placed outside of it and its purpose is to execute a single task (not instructions). A typical hardware accelerator example is for instance the GPU within a desktop computer. Figure 2.15 depicts a computing system that uses both a coprocessor and an accelerator, as well as a memory unit and the necessary interconnection.

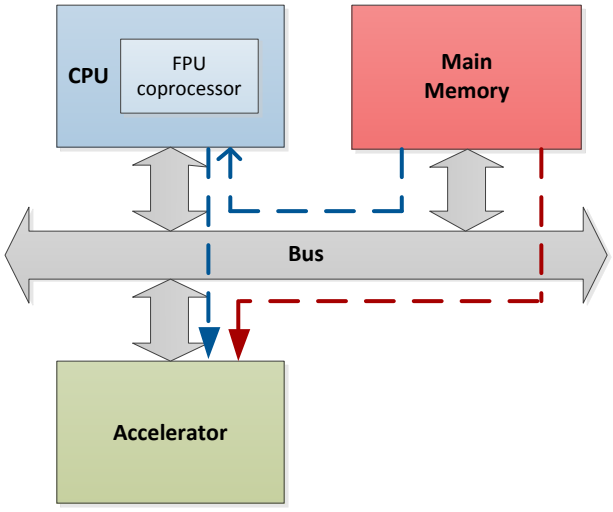
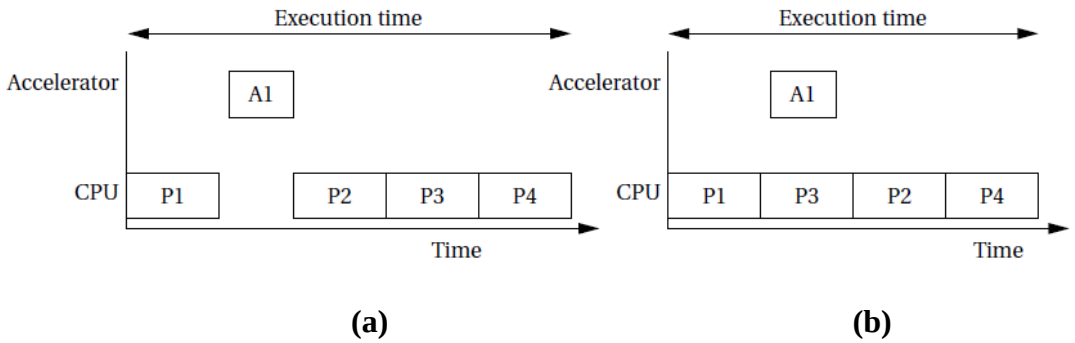


Figure 2.15: Hardware accelerated system. The dashed lines illustrate possible paths between the main memory and the accelerator.

In any case, the CPU has to signal the accelerator that it should start performing its task. Communication between the CPU can be achieved via registers (control, status, data registers etc.) in the accelerator and the interface between them usually resembles the interface of I/O devices [24]. The CPU can either pass the input data to the accelerator or if the data demands are really high, the accelerator should be able of fetching the data from the memory by itself (DMA).

In the latter case, the designer has to be very careful because the bus is a shared resource between the CPU and the accelerator. If the CPU needs to access the main memory while the accelerator is fetching data, then the CPU will have to wait or interrupt the accelerator and this comes with the cost of extra delay that has to be accounted during the design phase. This brings up the concept of single-threaded and multi-threaded systems. Figure 2.16 [24] illustrates this concept.



(a) **(b)**

Figure 2.16: (a) Single-threaded execution, **(b)** multi-threaded execution [24].

In Figure 2.16 (a), the CPU executes process P1 and then it has to wait until the accelerator has finished A1 in order to continue with the execution of P2, P3 and P4. In this case, only one process can be executed at a time. In Figure 2.16 (b), one can observe that while the accelerator is executing A1, the CPU can execute P3, which means that two processes can be executed at the same time. At this point, it has to be noted that whether a system can be single-threaded or multi-threaded depends on the data dependencies among the processes. If for instance P3 would require as input the output of A1, then it would be impossible to execute them at the same time. Consequently, special attention during the partitioning of the application must be given.

An important parameter that determines whether an accelerator should be used or not, is the speed-up factor that it can offer. Assuming that executing a process on the accelerator requires [24]:

$$t_{accel} = t_{in} + t_x + t_{out} \tag{9}$$

where t_{in} is the time needed to read the input data, t_x the time needed to perform the calculations and t_{out} the time needed to write the output data, then according to Wolf [24], the speed-up factor (of a single-threaded system) can be defined as:

$$S = n(t_{CPU} - t_{accel}) \tag{10}$$

whereas, *n* is total number of times that the accelerated process is executed. From the previous equation, it is clear that the more times a process has to be executed as well as the smaller the time that the accelerator needs and the bigger the time that the CPU needs, the bigger the speed-up will be. Therefore, processes that have to be executed many times and/or are very slow when executed on the CPU, are usually good candidates for acceleration [24].

The utilization of hardware accelerators along with general purpose CPUs can be considered as a very efficient way to realize electronic systems [23]. On the one hand, the easiest way for someone to implement an application is to implement it in software since software development is a relatively easy task (compared to hardware development), the availability of development systems (desktop computers or laptops) is very high, many off-the-shelf CPUs that can execute software exist, so it would be reasonable for one to argue whether new hardware should be developed. On the other hand, besides the performance of a system, another major parameter that dominates the design of electronic systems nowadays is the energy efficiency that they can offer. In [22], it is reported that the energy efficiency for different implementation alternatives (from pure software to pure hardware implementations) can differ by orders of magnitude (favoring the pure hardware implementations).

Consequently, accelerators allow the realization of mixed (HW/SW) implementations, whereas some parts of a certain application are implemented in software and some others in hardware. This is very important as they can trade-off the ease of implementation versus performance and energy efficiency.

2.5 The ‘two-process’ Design Method

As the complexity in the design of modern electronic systems is constantly increasing, methods that simplify the design process, offer fast simulation times and guarantee that the design is synthesizable are getting more and more interesting. Traditional VHDL design flow usually includes the description of the desired system by several concurrent processes. For complex systems there can be hundreds of such processes that each of which is sensitive to several signals. This makes the description of such systems difficult for someone to understand, the large number of signals leads to slow execution times and additionally there is always the chance that the designed system is not synthesizable. The two-process method developed by Gaisler [39], attempts to solve these problems by proposing the use of one sequential and one combinatorial process in each VHDL entity. Figure 2.17 [39] depicts the idea behind the two-process method.

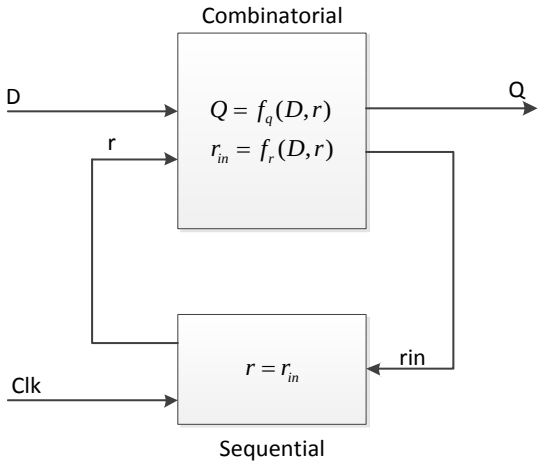


Figure 2.17: A generic system described by two processes (reproduced from [39]).

The system illustrated in Figure 2.17 is described by one combinatorial and one sequential process. Its inputs *D* are directly connected to its combinatorial part and moreover the combinatorial part drives the signal *rin* which is the input of the sequential part as well as the output *Q*. The sequential part latches this signal and on the next clock edge, it forwards it to the signal *r*, which is used as an input by the combinatorial part. This allows the designer to define directly which signals should be registered or not. While in Verilog, the designer can directly define what should be implemented in registers or simple wires, in VHDL this is often ambiguous and therefore this is a very important property of the two-process method.

Furthermore, the registered signals can be grouped in record types so that the sensitivity list of the combinatorial part can contain only these records [39]. This guarantees that these signals are synthesizable and moreover the addition or removal of extra signals can be done within the declaration of the corresponding record type, without any need to modify the sensitivity list of the combinatorial process. This keeps the sensitivity list both short and readable [39]. This feature, along with the fact that the system is described by only two processes, simplifies the maintenance and reusability of the designed entities allows the fast simulation in order to verify its functionality and provides a guide for the efficient design of modern digital systems.

2.6 Previous Work

This section describes the previous work that has been performed, with respect to the biomedical algorithm and the current thesis. Figure 2.18 illustrates all the intermediate steps that have been performed so far. The initial version (I) of the *STLmax* calculation algorithm was initialy implemented in Matlab by researchers at Arizona State University (ASU). Since the algorithm was supposed to be implemented in actual software it was necessary to be

implemented in a high level programming language. Matlab offers the highest level of abstraction in programming, however the advantage of this abstracted way of programming comes with the cost of low performance; Matlab implementations are usually slower than implementations in other high level programming languages. The language that was selected to move forward with this application is C due its wide acceptance in the field of embedded systems.

The original version was modified by researchers at NTNU (as well as the rest of the discussed versions) in version (II) so that the transition in C could be performed easier, and then the first C translation (III) was done. Version (III) was afterwards revised in an optimized version (IV) by applying high level optimization techniques such as loop unrolling and minimizing the usage of computationally expensive functions such as the logarithmic function (from equation (5) in Section 2.1.1, the *STLmax* is calculated as a sum of logarithms). Finally, another version (V) was created from (IV), for the mapping of the algorithm on the STM32F microcontroller [25], which has an ARM processor with hardware floating-point support and single-cycle DSP oriented instructions [15].

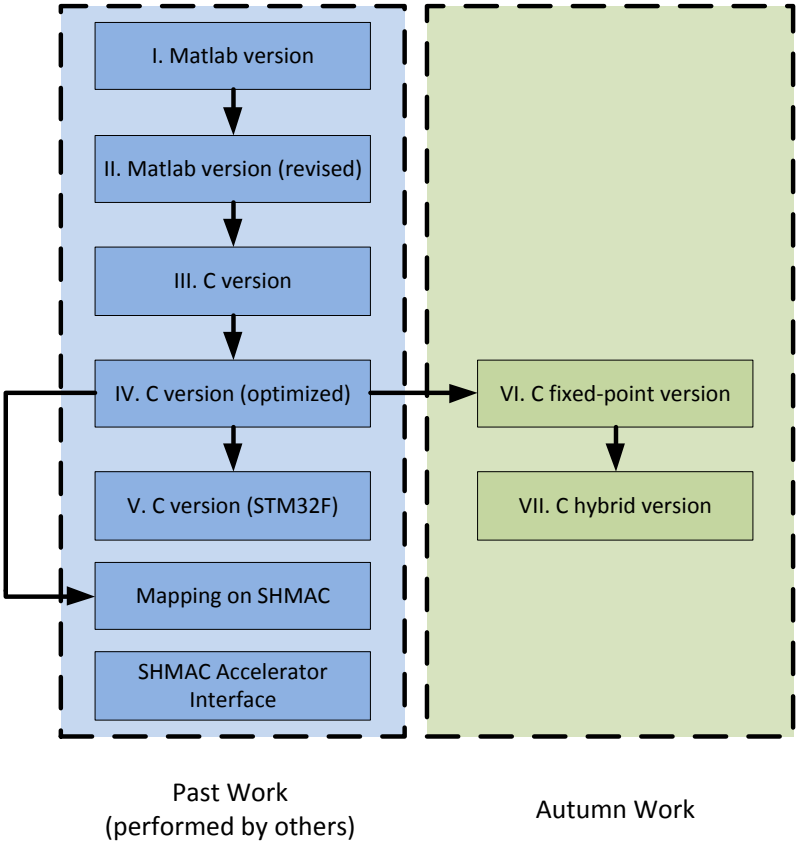


Figure 2.18: Previous work related to the current thesis.

Version (IV) was mapped by Berg [27] on the SHMAC platform. The algorithm was profiled on SHMAC and its execution time was found to be 52.46s. Some parts of its code were

at that time moved into a hardware accelerator. Different accelerators were studied, and it was found that the most efficient one is a fixed-point accelerator. Version (IV) as well as all the software versions discussed so far, involves floating-point operations. Therefore, the proposed accelerator had to perform conversion from floating-point into fixed-point, process the fixed-point variables and finally convert the calculated result back into floating-point representation.

As long as hardware accelerator support on the SHMAC platform is concerned, this problem has been tackled by Teilgård [33]. In that project, different interfaces for the integration of accelerators within a SHMAC processor tile were designed. The accelerator in this case occupies some of the tile registers (as discussed in Section 2.3) so that the communication between it and the CPU can be performed by memory-mapped instructions; to the CPU, the accelerator is nothing else than memory positions.

Versions (VI) and (VII) of the *STLmax* calculation algorithm were developed during the project work [26] in the autumn of 2014. Both of them are based on version (IV) which at that moment was the most optimized platform-independent version. The main idea behind them is the translation of version (IV) (an optimized floating-point implementation) into a fixed-point version. In this way, the processor's hardware can be utilized in a more efficient way since in both Amber and Turbo-Amber, floating-point operations have to be emulated by software routines, and this results in high execution time (as also reported by Berg [27]).

Version (VI) was developed first. However, at that time it was not feasible to execute it on the SHMAC platform and consequently an ARM simulator was used to evaluate its performance. After the execution of the algorithm on the simulator, it was found that Version (VI) was slower than the initial floating-point version. This was caused due to the increased accuracy demands in one specific point of the algorithm. At this specific point, the required operations are two multiplications and one division and their operands need to be scaled (discussed in [26] and also in Appendix A). Snippet 2.1 illustrates the way that the operand scaling in a multiplication is performed in the C-code.

Since the variables are encoded in two's complement representation it is important to know when they are positive or negative in order to handle them properly. If they are positive the identification of the leading and trailing zeros can be performed relatively easy, however if they are negative then they will have several MSBs equal to '1' and therefore it is beneficial to invert their sign, perform the operand scaling and then invert them back. This is done by marking the flags *flagNegativeA* and *flagNegativeB*. After that, the trailing zeros of the first operand (*operandA*) are identified in the first *while* loop. Variable *s1* is initialized to zero, and during the *while* loop it counts the trailing zeros of *operandA*. The condition inside the *while* loop: *!((operand>>s1)&1)* holds as long as the *s1*-bit of *operandA* is equal to zero and the loop keeps iterating. Otherwise it means that all the trailing zeros have been found and it stops iterating. Exactly the same procedure is performed to the second operand (*operandB*) in the second *while* loop. After the identification of the trailing zeros, both operands are shifted to the right.

After the identification of the trailing zeros and the shifting of the operands, the sum of their leading zeros needs to be found. This is performed in the last two *while*-loops. If the sum of the leading zeros is at least equal to 32, then the operands can be multiplied and their product

will occupy 32-bits, otherwise their product will require more than 32-bits and therefore it cannot be represented by a 32-bit variable. In the latter case the algorithm ignores the current operation and moves on with the rest (final *if*-statement).

```
//  Mark negative numbers (two's complement)
    flagNegativeA=0;
    flagNegativeB=0;
    if(operandA<0){
        operandA=-operandA;
        flagNegativeA=1;
    }
    if(operandB<0){
        operandB=-operandB;
        flagNegativeB=1;
    }

//  Count the trailing zeros in both operands
    s1=0;
    while(!((operandA>>s1)&1)){
        s1++;
    }
    operandA=operandA>>s1;

    s2=0;
    while(!((operandB>>s2)&1)){
        s2++;
    }
    operandB=operandB>>s2;
//  Count the sum of the leading zeros
    s3=31;
    s4=0;
    while(!((operandA)&(1<<s3))){
        s3--;
        s4++;
    }
    s3=31;
    while((!((operandA)&(1<<s3)))&&(s2<32)){
        s3--;
        s4++;
    }
    if(s4==32){
        result=operandA*operandB; //Keep in mind the new Q-notation
        if(flagNegativeA^flagNegativeB){
            result=-result;
        }
    }
    else{
//      ignore current operation
    }
```

Snippet 2.1: Operand scaling in version (VI).

The multiplication is performed in the final *if*-statement and the sign is defined by performing the XOR operation between *flagNegativeA* and *flagNegativeA*. The XOR operation (operand ^ in C) is used because the sign of the result needs to be inverted only when one of the two operands is negative. If both of them are positive or negative, then the result will be positive. The developer must be very careful when performing such tricks because the *Q*-notation of the result depends on the shifting operations of the operands.

According to the above analysis, it is evident that such operations have a corresponding computational cost. Keep iterating within the *while* loops (the iterations may vary from 0 to 32 in each loop, depending on the operands) results in high execution time for this code segment, and considering the fact that the part of code that uses this technique needs to be executed many times (140 390 times in total, involving 280 780 multiplications and 140 297 divisions [26]), it turns out that the operand scaling is not an efficient solution for this application and consequently the code was revised into version (VII).

Version (VII) is a hybrid (fixed-point/floating-point) version (VII) also developed during the autumn project [26]. In Table 2.2, one can observe that the latency of the emulated floating-point multiplication (193 clock cycles) can be potentially less than performing massive operand scaling operations. After this observation, the operations that involve operand scaling in version (VI) (two multiplications and one division) were converted into floating-point operations. The revised algorithm was executing again on the ARM simulator, and it was found that it is faster by 48% than the initial floating-point version (IV).

The conversion form fixed into floating-point was done using the following macro from ARM [16]:

```
#define tofloat(a, q) ( (float)(a) / (float)(1<<(q)) )
```

This macro, simply converts a fixed-point number ‘a’ with q-bits in its fractional part into a floating-point number in IEEE-754 format. Despite the fact that the conversion requires an expensive (in terms of latency) floating-point division, it is still more efficient than performing the earlier described operand scaling in this specific application.

Chapter 3

Application Mapping

In this chapter the mapping of the algorithm on the SHMAC platform is discussed. The methodology followed during the implementation is described first. The porting of the algorithm on SHMAC along with some system parameters that affect the performance of the algorithm as well as candidate implementation alternatives are discussed secondly, followed by the algorithm profiling and the hardware/software partitioning considerations of the application.

3.1 Methodology

Figure 3.1 illustrates the methodology adopted in the current thesis. As discussed in Section 2.6, the fixed-point version and the hybrid version of the *STLmax* calculation algorithm have not been executed on the SHMAC platform so far. Consequently, the first step is the porting of the algorithm on SHMAC and the evaluation of its performance on a processor tile, with respect to the application requirements. If they are met, then one can say that the software implementation of the algorithm is sufficient and no extra modifications should be performed. In this case, the energy efficiency will have to be verified as a function of the measured execution time and the power consumed by the SHMAC platform.

If, on the other hand, the applications requirements are violated, then the addition of extra hardware that speeds up the application can be considered inevitable (provided that the software cannot be optimized any further). In the current thesis, this additional hardware is considered to be hardware accelerator(s) (as discussed in Section 2.4). Consequently, the algorithm will have to be profiled in order to identify its bottlenecks and then one or more of the identified bottlenecks should be moved into the accelerator(s), and therefore the design of the accelerator(s) should follow next. After the design phase is complete, the accelerators(s) should be integrated in the SHMAC platform, and later the algorithm should be executed again in order to evaluate its performance when the accelerator(s) are utilized. The energy efficiency should be evaluated as well, as a function of the final execution time (which will be decreased due to the addition of the accelerator) and the consumed power (which will be increased due to the accelerator(s)).

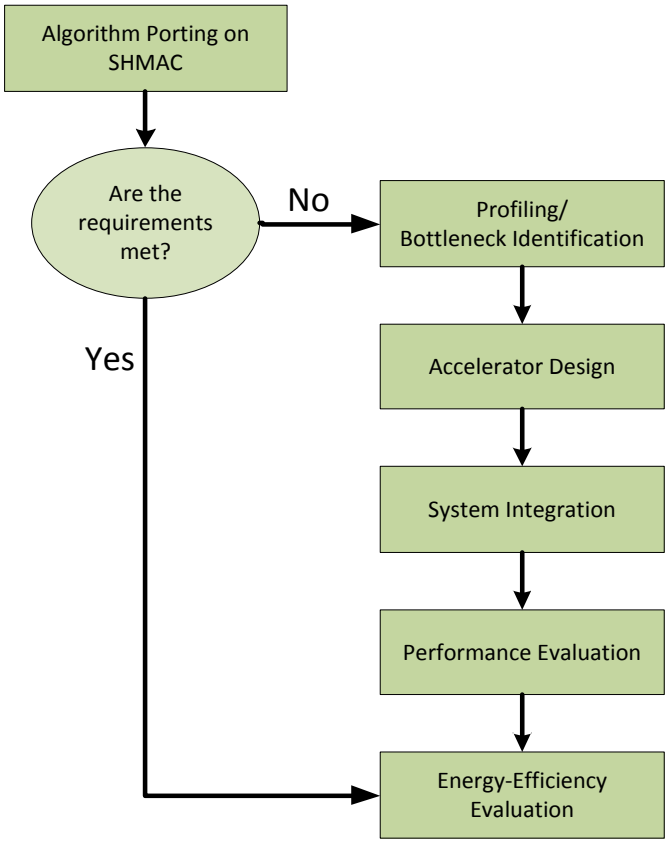


Figure 3.1: Methodology adopted in the current thesis.

As analyzed in Section 2.1 the *STLmax* calculation algorithm is not a straightforward procedure. The algorithm has to search within an EEG signal and check whether there exist certain points with certain properties (Section 2.1.1) and then act accordingly. This results into different runtimes for different input EEG signals, and therefore it is beneficial to select a suitable EEG sample for further experiments on SHMAC instead of executing the full dataset. The full dataset of EEG recordings, consists of 112 512 different signals; this is a full set of EEG recordings from 32 different channels with duration of 10 hours each.

For this purpose, the optimized floating-point version (IV) was selected for the execution of the whole dataset on an Intel I5 processor at 2.6GHz and 4GB of RAM. Executing versions (VI) and (VII) would make no difference in this case, because the target processor adopts a superscalar architecture with several optimized FPUs. Moreover, using one of the rest of the existing software versions would also make no difference for the current purpose, which is the algorithm's performance estimation. After the execution of the algorithm with all the available EEG signals as input, the following histogram (Figure 3.2) with the runtimes in seconds was obtained.

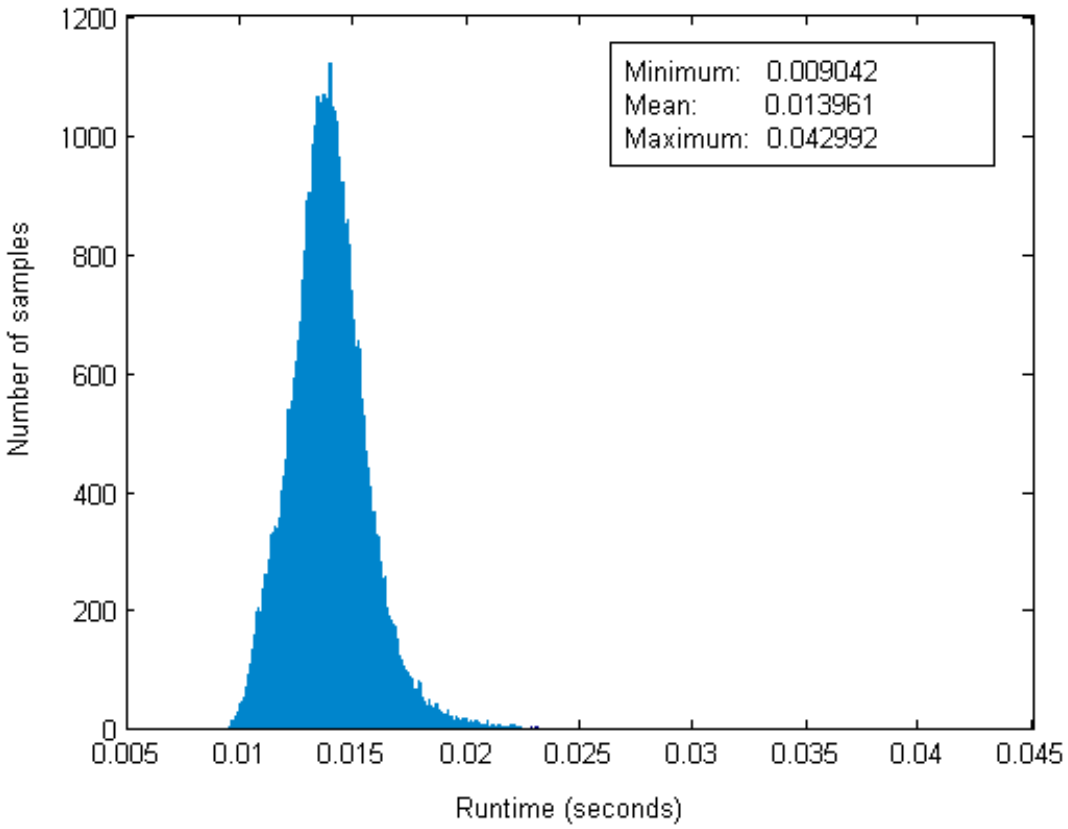


Figure 3.2: Histogram of runtimes of all STLmax values.

In the above histogram, one can observe that there is a big deviation over the different runtimes. The mean value of them is 0.013961s while the minimum and the maximum values are 0.009042s and 0.042992s respectively. According to the application's requirements, the execution of the algorithm on the SHMAC platform should result in a mean execution time of 0.32s and so porting an EEG signal that results in the worst case or in the best case execution time must be avoided. An EEG signal that requires a runtime slightly over (by a factor of 10%) the mean value (0.01542s) was extracted for this purpose. It would be reasonable for someone to assume that an EEG signal that requires exactly the average runtime should be extracted, however due to the fact that from an architecture point of view, the target processor (ARM) is completely different than the processor used here (Intel), a safety factor of 10% was empirically considered.

3.2 Porting the Algorithm on SHMAC

In order to execute the algorithm on the SHMAC platform, a SHMAC instance must be synthesized for this purpose. This was selected to be the layout presented in Figure 3.3, which includes the minimum but necessary hardware tiles.

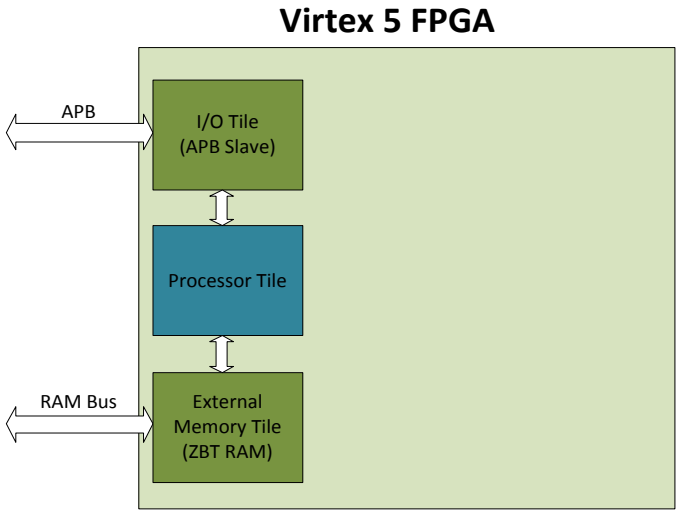


Figure 3.3: SHMAC layout used for the execution of the algorithm.

The layout presented in Figure 3.3 does not necessarily correspond to the actual placed and routed circuit inside the FPGA device, since this is determined by the synthesis tool (in this case was Xilinx XST). Nevertheless, it illustrates that these are the minimum necessary resources that have to be used. According to the analysis of the platform in Section 2.3, one I/O tile has to be used for the communication with the host system, one tile for the communication of the FPGA with the external off-chip memory and one processor tile that will be used for the execution of the algorithm.

The starting point for the processor tile was the modified version of Amber 25, with the single cycle multiplier, as discussed in Section 2.3. Attempting to use Amber with the Booth multiplier would result in an execution time far from meeting the requirements (according to the execution time reported by Berg [27]) and thus it was initially rejected as an option. It could only be considered as a design option only if the fast multiplier can speed up the application beyond the requirements. The floating-point version (IV), fixed-point version (VI) and hybrid version (IV) of the algorithm were executed on SHMAC. These are the most optimized versions compared to the rest and thus they were considered for further platform-dependent analysis. The execution times for the three different software implementations are presented in Table 3.1.

	Floating-Point	Fixed-Point	Hybrid
Cache disabled	215.73	146.23	76.65
Cache enabled	13.33	7.81	4.50

Table 3.1: Runtimes in seconds, of the three different software implementations (compiler optimization –O0).

From the runtimes in Table 3.1, it appears that the fixed-point version performs better than the original floating-point version, in contrary to the results reported in [26]. However the hybrid version is still the fastest one. This proves the assumption that emulating a small subset of all the involved operations is for this application more efficient than performing massive operand scaling operations. More specifically, the fixed-point version is faster by 41% compared to the floating-point and the hybrid implementation is faster by 66% compared to the initial floating-point. Furthermore, it is also clear that enabling the processor's cache affects massively the performance of the application (approximately 17 times faster when cache is enabled). Nevertheless, even the hybrid version cannot meet the application's requirements (a runtime of 0.32s). The software was compiled without any compiler optimization options enabled (optimization flag –O0), so the next step was to activate them. The code was recompiled with the compiler set to optimize it with respect to speed (optimization flag –O3). The results in this case are presented in Table 3.2.

	Floating-Point	Fixed-Point	Hybrid
Cache disabled	134.24	44.92	33.19
Cache enabled	8.79	4.36	2.12

Table 3.2: Runtimes in seconds, of the three different software implementations (compiler optimization –O3).

Comparing the values of Table 3.1 and Table 3.2, indicates that the compiler optimization can be a very important asset, since it can improve the performance of the algorithm by a high degree (over 50% improvement for the hybrid version). The execution time is still violating the performance requirements, and therefore additional hardware has to be utilized. At this point, the FPU coprocessor was added to the processor tile. Due to the compiler problems mentioned in Section 2.3.3, only the hybrid version was possible to make use of the FPU. The fixed-point version can in no case benefit from the FPU, while the floating-point version would have to be rewritten so that it utilizes the floating-point library developed by Walstad [38]. Considering the latency values of the FPU, this option was disregarded because even by using the FPU, the floating-point version cannot compete the hybrid, in which most of the mathematical operations are performed by fixed-point operations that are executed in one clock cycle. The execution time of the hybrid version with the FPU, was found to be equal to 2.09s (with the cache enabled and compiler optimization –O3). The performance gain in this case is just 30ms (with respect to the

initial runtime of 2.12s) and is again not sufficient in order to meet the requirements. The next step is to try out a faster processor (since that's the limit for Amber). A SHMAC layout with Turbo-Amber in the processor tile was synthesized and the algorithm was ported again on the platform. The runtimes in this case are presented in Table 3.3.

Compiler Optimization	Floating-Point	Fixed-Point	Hybrid
Flag -O0	11.73	7.18	4.06
Flag -O3	7.32	2.24	1.79

Table 3.3: Runtimes in seconds, of the three different software implementations on Turbo-Amber.

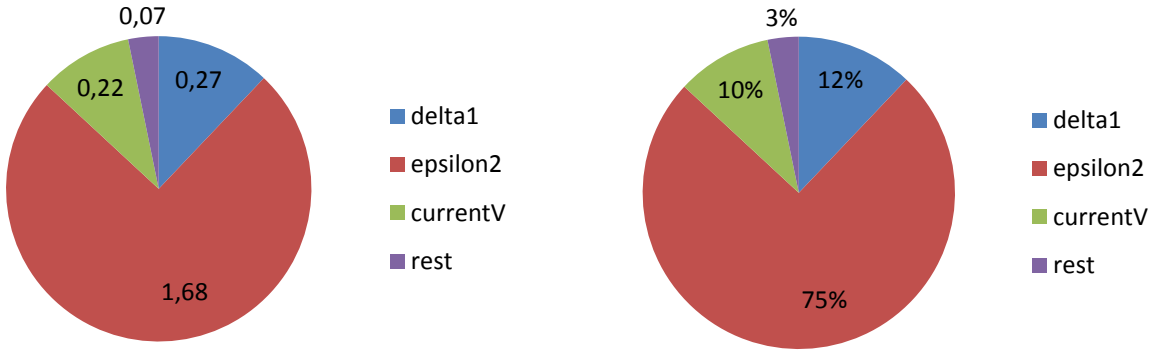
Having the cache disabled would be senseless in this case, so the results in Table 3.3 are with the cache enabled. Turbo-Amber has increased performance in all cases, with the fixed-point version being most benefitted (approximately 50% performance improvement in this case). This is because of the branch prediction that Turbo-Amber has. The operand scaling operations involved in the fixed-point version, are performed in the *while* loops presented in Snippet 2.1. When these *while* loops are translated from C into assembly and machine language (during the compiling and linking phases) result into branch instructions that the processor has to execute. This can become clear by considering the fact that a *while* loop keeps iterating if a specific condition holds. If the condition holds, then the program jumps to a corresponding memory address (the one that contains the next instruction). If not, it stops iterating and jumps to the next instruction, consequently such condition checks correspond to branch instructions. The floating-point version as well as the hybrid version get also benefited from Turbo-Amber, however since both of them involve floating-point operations that have to be emulated by software, the performance gain is relatively low compared to the fixed-point version. The utilization of the FPU was intentionally disregarded in the case of Turbo-Amber, since according to the runtime measured on Amber, it is certain that the FPU cannot accelerate the application by a sufficient degree.

With an execution time equal to 1.79s and a given time frame of 10.24s, Turbo-Amber is able to calculate the *STLmax* values of five different channels within the given time frame. In order to meet the requirements of 32 *STLmax* values per 10.24s (or 0.32s per value), extra hardware has to be utilized. This can be done in three different ways. The first one is to use extra processor tiles and calculate five *STLmax* per tile. In this case, seven processor tiles would be necessary (each one calculating the values of five channels). The second one would be again the usage of extra processor tiles and implement a multithreading version of the algorithm. The internal loops of the algorithm could be executed in parallel on the different processor tiles and this could result into a lower execution time that could potentially meet the performance requirements. Last but not least, is the utilization of hardware accelerators. Hardware accelerators are usually components less complicated than conventional processors, since, as mentioned in Chapter 2, they are designed in order to be capable of executing efficiently a single

task and not all types of tasks as it happens to the processors. This can potentially result into simple hardware that requires less silicon and offers better energy efficiency compared to using multiple processor tiles and therefore this option is investigated in the following sections.

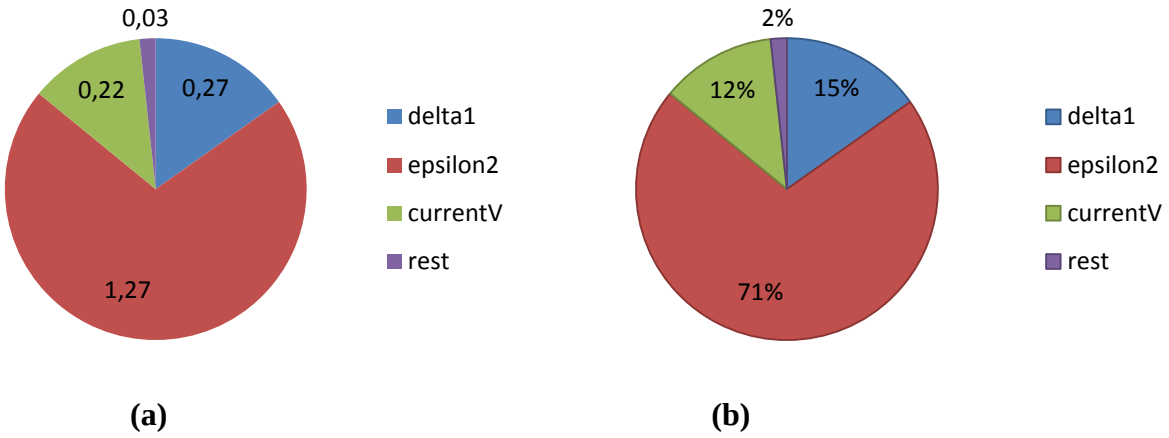
3.3 Algorithm Profiling

Before attempting to design one or more accelerators for the *STLmax* calculation algorithm, it is essential to identify the bottlenecks of the code and analyze whether they should be implemented in the accelerator or not. The identification of these bottlenecks can be performed simply by profiling the code on the SHMAC platform. The floating-point version can be easily rejected for this task, since in all cases it is by far the slowest one compared to the fixed-point and the hybrid version and it is very hard to accelerate in such degree that it can meet the requirements. The profiling was performed for the two latter software implementations on Turbo-Amber, because of their similarities and differences; they are identical except in one specific part whereas the fixed-point version performs operand scaling operations while the hybrid version replaces these operations with floating-point ones (as discussed in Section 2.6). Therefore if this part is considered to be moved into the accelerator, the structure of the accelerator will be different for these two implementations, resulting into simple or complex hardware. Figures 3.4 and 3.5 summarize the profiling results for the fixed-point and the hybrid versions respectively. The compiler optimization options were in all cases activated (compiler flag –O3) and the cache enabled.



(a) **(b)**

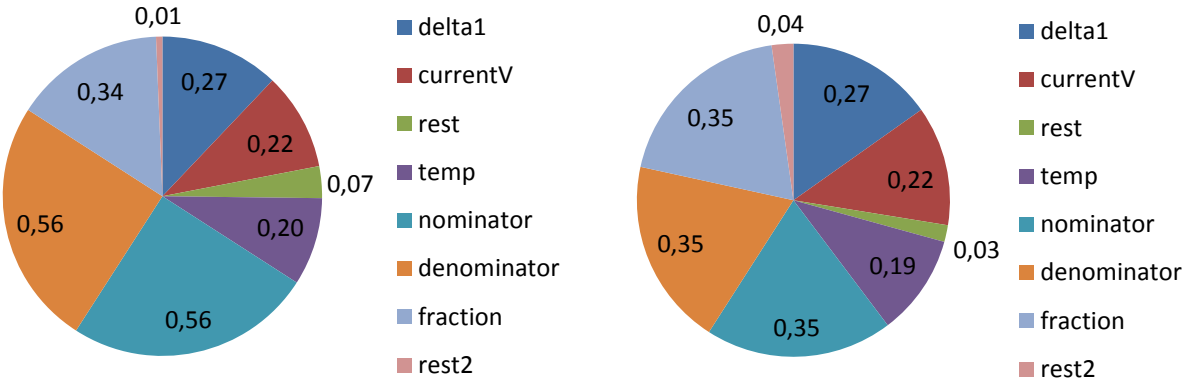
Figure 3.4: Profiling results of the fixed-point version on Turbo-Amber; **(a)** time in seconds **(b)** time as percentage of the total execution time. The total execution time is 2.24s.



(a) **(b)**

Figure 3.5: Profiling results of the hybrid version on Turbo-Amber; **(a)** time in seconds **(b)** time as percentage of the total execution time. The total execution time is 1.79s.

In Figures 3.4 and 3.5, it can be observed that the most intensive part of the algorithm is the calculation of the *epsilon2* quantity. This is the part of the code that the precision requirements are very high and the fixed-point code has to perform a lot of operand scaling operations (as discussed in Section 2.6) while the hybrid performs those using floating-point operations. Both versions spend the same amount of time on calculating *delta1* and *currentV* quantities, something that it is expected since the operations involved in these computations are identical for both of them; they involve simple fixed-point operations. After analyzing the computations of *epsilon2* even further, the graphs illustrated in Figure 3.6 are obtained.



(a) **(b)**

Figure 3.6: Further analysis of the computations involved **(a)** in the fixed-point version and **(b)** in the hybrid version. The numbers represent time in seconds.

In Figure 3.6, the calculation of *epsilon2* is decomposed into its partial calculations. These include the calculation of the *temp* quantity, *nominator*, *denominator*, *fraction* and a

negligible part denoted as *rest2*. The *temp* calculation is identical for both versions. The calculations of *nominator*, *denominator* and *fraction* are the only different calculations between them. These are the results of two multiplications and one division respectively. In the fixed-point version, they are performed by applying operand scaling operations; keeping however all the operations in fixed-point, while the hybrid version converts these quantities from fixed-point into floating-point and then it performs the necessary operations (emulated by software). It is however evident that the target processor can emulate floating-point operations more efficiently than the operand scaling (approximately 38% performance difference for these particular calculations).

3.4 Hardware/Software Partitioning

Hardware/Software partitioning is a very important task in electronic system design. During this task, it is determined whether an application should be implemented in software, hardware or a combination of them. In the case of the *STLmax* calculation algorithm, the starting point can be considered that the whole algorithm is implemented in software and is executed on the target processor (Turbo-Amber). This however results into a system that violates the performance requirements, as mentioned in Section 3.2, and consequently it is necessary to implement some parts of the algorithm in hardware.

According to the profiling results discussed in Section 3.3 it is necessary to move the calculations of *epsilon2* and at least one between *delta1* and *currentV* into hardware, otherwise it is impossible to have a system (combination of hardware and software) that performs the demanded calculation in maximum 0.32s. Snippet 3.1 illustrates the structure of these blocks within the algorithm.

```
for (k=1; k < kMax; k++)
    {
      delta1[]
        while (condition=TRUE)
        {
          currentV[]
            if (currentV.size != 0){
                for (i=0; i < currentV.size; i++){
                  epsilon2
                }
            }
        }
    }
```

Snippet 3.1: Structure of the most demanding blocks within the algorithm.

Snippet 3.1 illustrates the fact that the computationally most intensive blocks of the algorithm are nested within loops. Consequently, the total number of executions of these blocks has to be determined. During the execution of the algorithm, it was found that block *delta1* was executed 168 times, *currentV* 685 times and *epsilon2* 140 390 times. According to these numbers and the runtimes presented in Section 3.3, the runtimes as well as the clock cycles spent on one execution of each block can be determined. For *delta1* and by taking into account the processors frequency (60MHz) and period (0.016667μs), this is:

$$t_1 = \frac{0.27s}{168} \simeq 1607\mu s \text{ , or 96 424 clock cycles}$$

The same applies to *currentV* and *epsilon2*. In this case one execution of *currentV* takes:

$$t_2 = \frac{0.22s}{685} \simeq 321\mu s \text{ , or 19 261 clock cycles}$$

And finally one execution of *epsilon2* takes:

$$t_3 = \frac{1.26s}{140390} \simeq 9\mu s \text{ , or 540 clock cycles}$$

Snippets 3.2, 3.3 and 3.4 depict how *delta1*, *currentV* and *epsilon2* are implemented in the C-code respectively. Blocks *delta1* and *currentV* are exactly the same for both the software versions, however for demonstration reasons, only the *epsilon2* implementation in the hybrid version is included here.

```
for (i = 0; i < 2008; i++ ){
    long d2temp=0;
    curr_d2partial=curr_VectorB[0]- VectorB[i];
    d2temp +=curr_d2partial*curr_d2partial;
    curr_d2partial=curr_VectorB[1]- VectorB[i+4];
    d2temp +=curr_d2partial*curr_d2partial;
    curr_d2partial=curr_VectorB[2]- VectorB[i+8];
    d2temp +=curr_d2partial*curr_d2partial;
    curr_d2partial=curr_VectorB[3]- VectorB[i+12];
    d2temp +=curr_d2partial*curr_d2partial;
    curr_d2partial=curr_VectorB[4]- VectorB[i+16];
    d2temp +=curr_d2partial*curr_d2partial;
    curr_d2partial=curr_VectorB[5]- VectorB[i+20];
    d2temp +=curr_d2partial*curr_d2partial;
    curr_d2partial=curr_VectorB[6]- VectorB[i+24];
    d2temp +=curr_d2partial*curr_d2partial;
    delta1[i] = d2temp;
}
```

Snippet 3.2: C snippet of *delta1*.

```
scalmxfixed=(xMaxfixed>>5)*LUT_1[z];
scalmxfixed=convertq(scalmxfixed,26,18);
cvCount = 0;
for (i = 0; i < 2008; i++ )
{
    if (d1fixed[i]<= scalmxfixed){
            currentV [cvCount] = i;
            cvCount ++;
    }
}
```

Snippet 3.3: C snippet of *currentV*.

```
temp=0;
select=alfa[i];
temp=temp+VectorA[0]*VectorB[select];
select=alfa[i]+tau;
temp=temp+VectorA[1]*VectorB[select];
select=alfa[i]+2*tau;
temp=temp+VectorA[2]*VectorB[select];
select=alfa[i]+3*tau;
temp=temp+VectorA[3]*VectorB[select];
select=alfa[i]+4*tau;
temp=temp+VectorA[4]*VectorB[select];
select=alfa[i]+5*tau;
temp=temp+VectorA[5]*VectorB[select];
select=alfa[i]+6*tau;
temp=temp+VectorA[6]*VectorB[select];
temp=temp>>2;

nom= mpoint - temp;
float nomFloat=tofloat(nom,20);
nomFloat=nomFloat*nomFloat;
float denomFloat = tofloat(delta1[alfa[i]],18);
denomFloat=denomFloat*tofloat(deltaf,20);

if (denomFloat > nomFloat){
    fraction = nomFloat / denomFloat;
}
else{
    fraction = 1;
}
if (fraction > epsilon2max){
    epsilon2max = fraction;
    epsilon2 = i;
}
```

Snippet 3.4: C snippet of *epsilon2* in the hybrid version.

An overview of the above snippets can draw useful conclusions about which of these should be moved into an accelerator. The *delta1* block needs to calculate 2008 different values

every time it is executed; it has to perform seven multiplications, additions and subtractions respectively. Such an amount of operations justifies the fact that it is the computationally most intensive block (for one execution). An accelerator that implements these calculations should be able to fetch the data by itself (it should have DMA capabilities). Otherwise the amount of time that should be spent by the processor to pass such volume of data to the accelerator, wait for the accelerator to calculate the results and then read back the calculated results will not be sufficient to speed up the application by a desired factor.

The *currentV* on the other hand has to perform relatively simple operations (mostly comparisons). It iterates for 2008 iterations and after each iteration, it adds or not one extra element the *currentV* matrix, while its data requirements for one iteration are not excessive. In addition, it is nested inside a *while* loop. The implementation of such block in hardware would be meaningful if the *while* loop could be unrolled in order to exploit the data parallelism and the accelerator could calculate the corresponding matrix within one of a few executions. The given loop is however difficult to be unrolled due to its condition, and consequently the implementation of this block in hardware can be disregarded.

Finally, *epsilon2* is the block that needs to be executed many more times than the previously mentioned blocks. It is nested within a loop with variable number of iterations (its size depends on the number of elements contained in the *currentV* matrix). However it can be observed that its input-output requirements are relatively low as well as in each iteration not all of the input values need to be updated (i.e. *VectorA[]* is the same for most of its iterations). Consequently, implementing this block in hardware could potentially speed up the application by a high factor. If for instance its execution time can be reduced from 540 to 50 clock cycles (hypothetical value) then 490 clock cycles are saved in each iteration, resulting in 490×140390 less clock cycles consumed in total. Up to this point, the following equation holds:

$$t_{delta1} + t_{currentV} + t_{epsilon2} + t_{rest} = 1.79s \tag{11}$$

By replacing the actual runtimes it becomes:

$$0.27 + 0.22 + 1.26 + 0.04 = 1.79s \tag{12}$$

The above relation should however be:

$$t_{delta1} + t_{currentV} + t_{epsilon2} + t_{rest} \leq 0.32s \tag{13}$$

otherwise the performance requirements are violated.

It would be ideal at this point to have some estimation metrics about the performance of these blocks in hardware. In this way, one could have an idea whether implementing them in hardware would result into a system that meets the performance requirements or if it impossible to meet them, and the hardware-software partitioning of the application could be performed in a formal way. However, absence of such metrics, the next step is the direct design of an

accelerator module that implements *epsilon2*. As discussed in Section 2.5, software blocks that need to be executed many times should be preferred to be moved to an accelerator and in addition the communication cost between the processor and the accelerator in this case can be kept low. The performance of this module will also indicate whether it makes sense to utilize hardware accelerators in this application or whether the utilization of additional processor tiles is necessary. After evaluating this option, then additional considerations about moving *delta1* to another accelerator will have to be made.

Chapter 4

Accelerator Design and System Integration

As discussed in the previous chapter, the *epsilon2* block (the most time consuming one) of the algorithm is implemented in the hybrid and the fixed-point version using mixed (fixed-point and floating-point) operations and only fixed-point operations respectively. Consequently, since the *epsilon2* block is going to be moved into a hardware accelerator, it is necessary to evaluate these options in hardware, compare them to each other and finally integrate the most suitable one within the SHMAC processor tile in order to evaluate the benefits of this approach.

4.1 Accelerator for the Hybrid Version

Designing an accelerator that implements a corresponding software block, first of all requires the structure of the block that will be moved into the accelerator (*epsilon2* in this case). This is illustrated in Snippet 3.4 (in Section 3.4). One execution of this code segment, takes 540 clock cycles, consequently the accelerator that will be designed has to perform these computations in as less as possible time, while at the same time it should fit inside the FPGA by occupying as few resources as possible. These two desired characteristics (fast and small) are hard to be achieved at the same time, therefore specific design decisions that trade-off the one against the other have to be made. These decisions are explained in the following paragraphs.

The starting point for designing the accelerator is its input and output requirements. An overview of Snippet 3.4 can reveal both of them. At first sight, the output would seem straightforward; this is the *epsilon2* value. This is partially correct because the value *epsilon2max* needs also to be passed back to the processor, so that it can be used in some other parts of the algorithm. If however one observes both the Snippet 3.4 along with the profiling results in Figure 3.6 (b) it is evident that almost all the computational cost of this block is used for the calculation of the variable *fraction* (along with the prerequisite values *nominator* and *denominator*). The last part of this block (the last four lines with the *if*-statement) have almost zero overhead when executed on the processor. Therefore the accelerator that is going to be designed may calculate this variable instead of *epsilon2* and *epsilon2max*. By applying this, the outputs are reduced by one and the corresponding hardware becomes less complicated since it has to perform fewer calculations.

The input requirements can be revealed by the calculations that have to be performed. The first value that has to be calculated in this block is the *temp* variable. Given the values of *VectorA* and *VectorB* (fourteen values in total), *temp* can be calculated as the sum of products among them. Variable *nom* (which is prerequisite for the calculation of *nomFloat*) can be calculated after *temp* has been calculated and it also requires variable *mpoint* which can be provided as input to the accelerator without any problem. Variables *delta1[alfa[i]]* and *deltaf* are required for the calculation of *denomFloat* value. This in total results in seventeen input values and one output.

Data dependencies are also very important for the implementation of this block. The first variable that needs to be calculated is *temp* along with *denomFloat* since these are the only variables that depend directly on the inputs and do not on any intermediate results. Function *tofloat(var , #bits)* performs the conversion from fixed-point into floating-point of variable *var* that uses *#bits* for representing the fractional part. As can be seen in Snippet 3.4, there are two of such conversions; one that converts fixed-point numbers with eighteen bits and second one with twenty bits in the fractional part, into the *IEEE754* floating-point representation. Implementing this in hardware would require two fixed to floating-point converters. Changing the fractional part of *deltaf* so that it is represented by eighteen bits could possibly make the use of the second converter unnecessary. Therefore, this option was investigated by executing the algorithm and observing whether this change would have any impact on the final STLmax value. The algorithm was executed for 100 different EEG input signals and it was found that the calculated values were the same as without it, so it was adopted. This indicates that either the two LSBs of *deltaf* are always zero or that such accuracy loss for this variable is compensated by the rest of the calculations. A fixed to floating-point converter was generated using Xilinx Core Generator for this purpose.

Variables *temp* and *nom* of Snippet 3.4, are fixed-point variables and they can be directly calculated by using the standard numerical operands of VHDL (+,-,*). On the other hand, floating-point operations are not directly supported in VHDL. Floating-point multiplications (for the calculations of *nomFloat* and *denomFloat*) are performed by a floating-point multiplier generated by Xilinx Core Generator as well. The same also applies to the case of the division (for the calculation of *fraction*) and the comparison (within the *if*-statement); all the floating-point components (a converter, a multiplier, a comparator and a divider) were generated in Core Generator. Other options would be their manual design, a complicated task within the given time frame and the use of VHDL packages, such packages however may have several incompatibilities with the target FPGA, as reported by Berg [27]. All the floating-point components were selected to be synchronous in order to keep the design complexity low.

Core Generator allows the designer to choose between different configurations, such as latency, handshake signals and mapping resources (DSP or LUT slices for the multiplier) for the floating-point components. According to Xilinx [37], the size of the floating-point components is proportional to their latency. This means that components that require more clock cycles to perform their task, they also require more resources. Although this sounds paradox, it should be noted that components with higher latency values, are usually pipelined and can operate in

higher frequencies than low-latency components. Table 4.1 summarizes the resource utilization and maximum operating frequencies for different floating-point multipliers available by Xilinx. Similar results also hold for the rest of the floating-point components and therefore there are no data included here.

Multiplier Type	Latency	LUTs	FFs	DSPs	Max. Freq. (MHZ)
Logic Only	2	663	145	-	137
	3	701	244	-	184
	4	593	433	-	246
	5	625	524	-	250
	6	634	593	-	316
DSP Medium-Usage	2	122	53	1	122
	3	190	92	1	190
	4	188	133	1	188
	5	282	237	1	282
	6	279	240	1	279
DSP Full-Usage	2	80	53	2	154
	3	97	88	2	218
	4	100	101	2	236
	5	101	77	2	295
	6	102	120	2	395
DSP Max-Usage	2	79	52	3	183
	3	85	59	3	236
	4	96	75	3	280
	5	96	111	3	350
	6	99	114	3	410

Table 4.1: Performance and resource utilization for Xilinx floating-point multipliers.

By keeping in mind that the final module should be able to operate at 60MHz in order to meet the time constraints of the SHMAC platform, the usage of components with minimum latency was considered as an initial design option. Although this is not absolutely necessary since by using for instance a clock divider the accelerator can be able to operate at lower frequency than the rest of the system, it is better to avoid such options at this phase of the design [40]. Such options can however be considered in case whereas the accelerator fails to meet the time requirements at all. After trying out different latency configurations for these components, it was found that the converter and the comparator can have one clock cycle latency, the multiplier two clock cycles and the divider fourteen clock cycles latency in order not to violate the time constraints. These latency values were obtained during the integration of the accelerator within the SHMAC platform (described later in Section 4.5). Attempting to obtain these numbers by studying only the accelerator and not the whole system was unsuccessful since the synthesis tool could estimate that the accelerator can meet the time constraints (for instance when the divider was selected to have four clock cycles latency), however after its integration and the synthesis of

the whole system there were many timing errors. This made the selection of the floating-point components a very time consuming task, since the SHMAC platform had to be synthesized several times.

The multiplier was selected to be designed with maximum usage of DSP slices while the rest of the components were designed with no DSP slices at all (such option was not available by Xilinx). This can result in a balanced design that utilizes both DSP and logic slices. For scheduling the partial computations according to the data dependencies (i.e. *nomFloat* and *denomFloat* must be calculated before *fraction*) a finite state machine (FSM) approach was adopted. The two process method (described in Section 2.6) was applied during the design of this module as well, so that it can be clear which signals will be synthesized as registers or simple wires. The inputs to the floating-point components along with all the intermediate results were synthesized as registers so that they can maintain their values while something that can affect them on the next clock cycle can happen (i.e. a floating-point component signals that it has finished its operation, so the calculated result is latched and forwarded to the combinatorial process on the next clock cycle).

The FSM updates its current state on every clock cycle (in the sequential process) and the conditions that trigger the state transition are calculated in the combinatorial process. A high level architecture of the accelerator is illustrated in Figure 4.1.

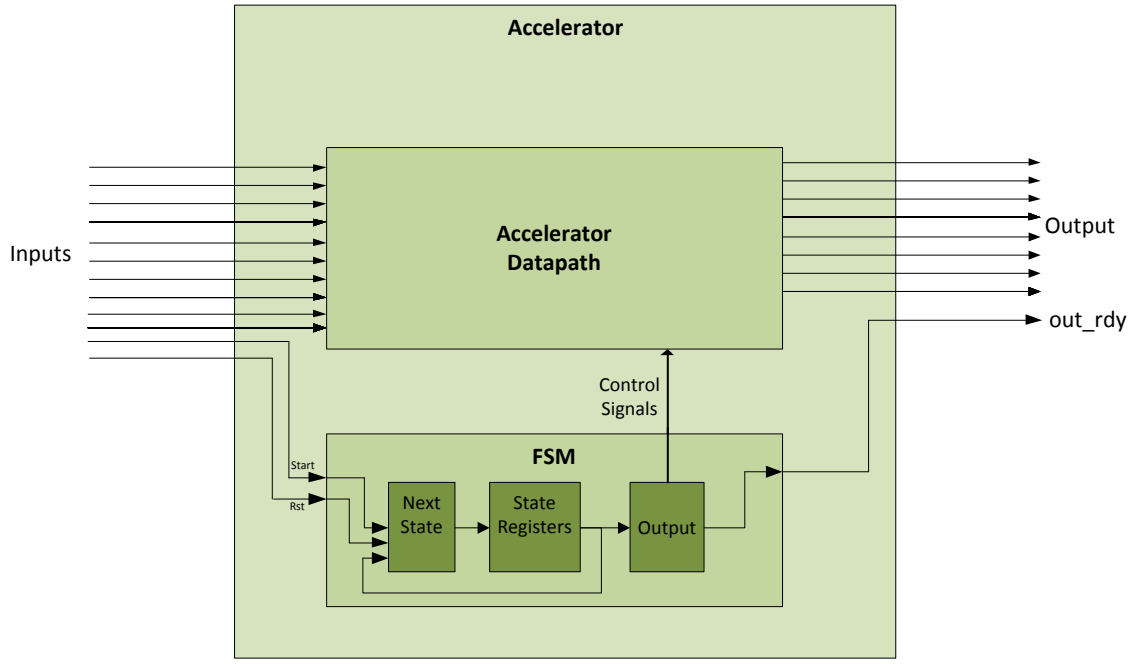


Figure 4.1: Accelerator's high level architecture (clock is omitted).

Once the *start* signal gets high (for one clock cycle), the FSM performs transitions to each state and when the calculations are completed, the *out_rdy* signal gets high for one clock cycle. Figure 4.2 illustrates the possible transitions between the states. During the design of the current accelerator, it was found that the FSM should consist of eight states. More specifically these states are:

a) Idle state

During this state the accelerator remains idle and no activity takes place within it. If the *start* signal gets high, *State-1* is triggered, otherwise the *idle* state is maintained.

b) State-1

If the converter is ready to accept new data, the variable delta1 is applied to its input, the variable *temp* is calculated (as the sum of seven products calculated by seven fixed-point multipliers), variable *deltaf* is shifted by two positions to the right and *State-2* is triggered. Otherwise the FSM remains in *State-1* until the converter is ready to accept new data.

c) State-2

Variable *temp* is shifted by two positions to the right, variable *delta1float* is calculated at the output of the converter (if the converter has finished its calculations), and variable *deltaf* is set at the input of the converter (if the converter is ready to receive new data). *State-3* is triggered as next state. If the converter is not ready, it remains in this state until it is.

d) State-3

Variable *deltaffloat* is ready at the output of the converter and is forwarded to the input of the multiplier along with the already calculated *delta1float*. The converter's input gets now the value *mulpoint-temp*. State-4 is triggered, except when the converter hasn't finished the conversion or is not ready to accept new data (either the converter or the multiplier). In the latter case, the FSM remains in *State-3*.

e) State-4

Variable *denomfloat* is calculated by the multiplier, while the multiplier's inputs are set equal to the output of the converter (floating-point value of *mulpoint-temp*).

f) State-5

Variable *nomfloat* is calculated by the multiplier and forwarded to the comparator along with *denomfloat* (provided that the comparator is ready to receive new data). State-6 is triggered if so, otherwise it waits in State-5.

g) State-6

If the comparator's output is high and the divider is ready to receive new data, then *nomfloat* and *denomfloat* are set to the inputs of the divider and *State-7* is triggered. If the comparator's output is low, *fraction* is set equal to '1' and the *out_ready* signal is set high. Idle state is triggered next.

h) State-7

This state is triggered if the *fraction* has to be calculated by the divider. If so, this state is maintained until the divider finishes its computation. The *fraction* gets equal to the output of the divider and *out_ready* signal is set high. Idle state is triggered next.

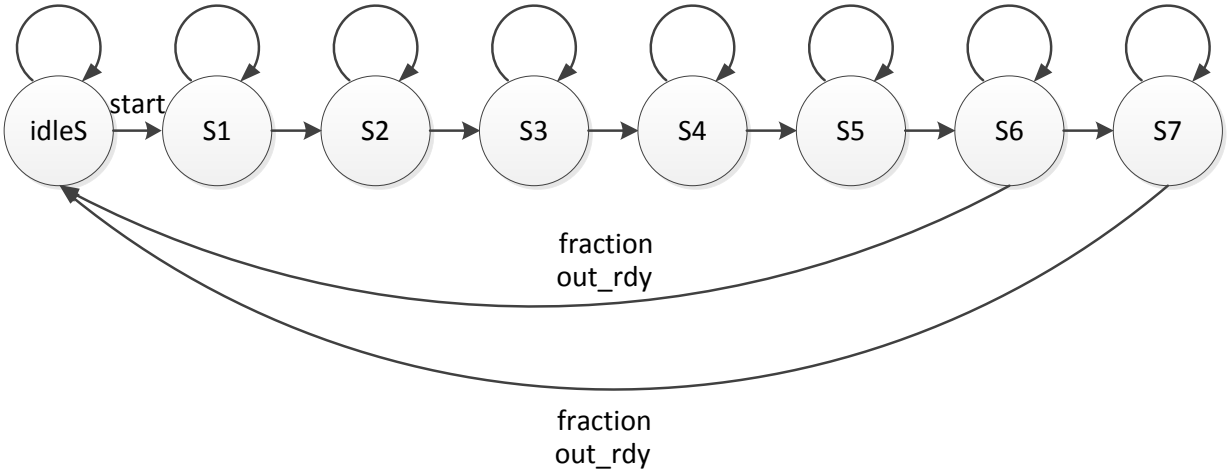


Figure 4.2: State transition diagram of the hybrid accelerator.

The discussed accelerator was synthesized by Xilinx XST and Table 4.2 summarizes its main features with respect to the occupied area, performance and power consumption. The full VHDL code can be found in Appendix B. The dynamic power was estimated by Xilinx Power Estimator (XPE), considering that the operating frequency is 60MHz and default (according to XPE) signal toggle rate settings.

Slice LUTs	DSP slices	Maximum Frequency (MHz)	Dynamic Power (mW)
1508	24	86.347	24

Table 4.2: Area, performance and power consumption of the hybrid accelerator.

4.2 Accelerator for the Fixed-Point Version

The *epsilon2* block as implemented in the fixed-point version in software, involves the operand scaling instead of floating-point operations. In the C-code, this block is similar to Snippet 3.4, however instead of having the *nomfloat*, *denomfloat* and *fraction* variables in floating-point, corresponding fixed-point values are computed by applying the operand scaling as discussed in Section 2.6. Due to its extensive length, a full snippet of the current block is not included here; nevertheless, Snippet 4.1 describes its function in pseudo-C code.

```
temp=… //As in Snippet 3.4
nom= mpoint - temp;
//perform operand scaling
scale(nom);
if(leading_bits_nom == 16){
    nom2 = nom*nom; //Q20*Q20 => Q40
    scale(delta1[alfa[i]]);
    scale(deltaf);
    if(leading_bits_denom == 32){
        denom = delta1[alfa[i]]*deltaf; //Q18*Q20 => Q38
        //Align the operands so that they can be compared
        align(nom,denom);
        if(align == 1){
            if(denom>nom2){
                s1 = leading(nom2);
                s2 = trailing(denom);
                if(s1+s2 == 14 ){
                    fraction = (nom2<<s1)/(denom>>s2); //Q14
                    if(fraction > epsilon2max){
                        epsilon2max = fraction;
                        epsilon2 = i;
                    }
                }
            }
            else{
                fraction = 1;
                if(fraction > epsilon2max){
                    epsilon2max = fraction;
                    epsilon2 = i;
                }
            }
        }
    }
}
```

Snippet 4.1: Pseudo-C code of the *epsilon2* block as implemented in the fixed-point version (red rows indicate whether the algorithm is allowed to continue to perform the calculations, while green rows show when the required value is calculated).

Variable *temp* is calculated as in Snippet 3.4. Variable *nom* is calculated next and scaled as discussed in Section 2.6. If it has at least 16 leading bits, then *nom2* can be calculated as the

nom square. If so, the algorithm proceeds to the calculation of *denom*. This variable depends on *delta1[alfa[i]]* and *deltaf*, consequently, these operands are scaled in order to determine whether their multiplication is feasible or not. If the sum of their leading zeros is at least 32 (the counting stops after the first 32 zeros are detected as discussed in Section 2.6) then the multiplication can be performed and *denom* is calculated. The next step, is the comparison between *nom2* and *denom*. These variables have different exponents (*Q*-notation) and therefore they need to get aligned (have the same exponents) so that the comparison can be performed. If they can be aligned (the actual process that performs this is discussed later in this Section), then the comparison between them is performed, and according to the comparison's result, variable *fraction* needs to be calculated.

If *denom* is smaller than *nom2* then *fraction* gets equal to '1'. If on the other hand *denom* is bigger than *nom2*, then *fraction* gets equal to the quotient of their division. It must be noted that this is a fixed-point division and therefore it must be taken care that the precision loss is acceptable. This is performed by considering the leading zeros of *nom2* and the trailing zeros of *denom*; if their sum is at least 14, then the division can be performed with the required accuracy [26], by shifting *nom2* to the left and *denom* to the right. In this case *fraction* will have an exponent equal to 14. If the conditions within the red-marked rows in Snippet 4.1 do not hold, then the current calculation is ignored, in terms that *fraction* and consequently *epsilon2max* and *epsilon2* do not get updated and they keep their initial values.

The approach about implementing this block in hardware is similar to the one used for the hybrid accelerator. At this point it must be noted that the fixed-point accelerator needs to have one extra input; the previous value of *fraction*, so that in case where it has to ignore the current calculation, it can present this value as output result. Its high level architecture is the same with the hybrid's, depicted in Figure 4.1. More specifically an FSM is used for the scheduling of the partial calculations that involve data dependencies, however the two-process method was not adopted here, because it was found beneficial for performance reasons to include some additional processes for the operand scaling.

As described earlier, the operand scaling requires the identification of the leading and trailing zeros of the corresponding operand. In the software implementation this is performed by counting them within some *while*-loops. Counting them synchronously (i.e. identify one leading/trailing zero in each clock cycle) would result in a simple hardware module (since counters are among the least complicated components in digital design) but with high latency and thus this option was quickly discarded. This task was decided to be performed by VHDL functions embedded inside a VHDL package (the VHDL code is available in Appendix B). This ensures the reusability of the designed functions in future projects and moreover they allow the asynchronous identification of leading and trailing zeros every time that a new operand is presented.

During the design of the accelerator, it was found that when a multiplication of two *n*-bit variables has to be performed, the synthesis tool defines (when using the '*' operator, or by using a fixed-point multiplier from Core Generator) that the result occupies twice the number of bits. Consequently the utilization of some 64-bit values for the intermediate results was

considered in this case. At this point, it needs to be noted that in a different case (i.e. an ASIC implementation) this should have been avoided since the size of the components (multipliers) would be a lot different for 32 and 64-bits.

According to Snippet 4.1, variable *temp* is calculated first, as the sum of products of some of the inputs. This value is saved in a 64-bit register. Furthermore instead of applying the operand scaling before the calculations of *nom2* and *denom*, corresponding 64-bit registers were considered for these values (*nom64* and *denom64*). Utilizing these 64-bit values would be ideal, since it could potentially eliminate the necessity of operand scaling. However, a problem about utilizing this technique appeared when the divider (for the calculation of *fraction*) had to be added in the design. While most of the fixed-point operators (addition, subtraction, multiplication, comparison, shift etc.) are directly supported by the corresponding operators in VHDL. The division on the other hand is not supported and therefore a divider was generated in Core Generator.

According to Xilinx [41], the maximum width of the divider inputs is 54-bits, and therefore at least 10 bits have to be truncated in each operand. Such a large divider is usually extremely big (in terms of area) and slow (in terms of latency). Table 4.3 summarizes the resource utilization and performance of several fixed-point dividers (the target device is Xilinx Virtex 5) as obtained by [41].

Dividend Width	Divisor Width	Latency	LUTs	FFs	DSPs	Max.Freq. (MHz)
10	14	17	355	412	7	450
10	14	2	267	70	7	75
36	36	28	972	1104	13	374
36	36	4	693	187	13	62
54	50	43	1398	1649	17	337
54	50	8	1009	266	17	47

Table 4.3: Performance and resource utilization for Xilinx fixed-point dividers.

As one can observe in Table 4.3, the selected operand width, along with the latency affect heavily the occupied resources as well as the maximum operating frequency. In order to meet the time constraints on the SHMAC platform and minimize the cases that the current calculation should be ignored, one could move on by selecting a 54-bit divider with maximum latency (43). This however is not efficient at all, provided that the whole hybrid accelerator (discussed in the previous section) occupies less LUTs than this specific divider. Consequently the divider selection is not a straightforward task and therefore the following assumptions were taken into consideration: (i) the operands of the divider must have minimum width and (ii) the latency must be as low as possible while not violating the time constraints (operating frequency at 60MHz). By following these assumptions, one can potentially find the smaller acceptable divider than fits to each application.

Following the first assumption, the operands were selected to be 32-bits wide, because in the pure software implementation such kind of operands were able to produce acceptable results

(results according to which the final seizure predictions are correct [26]). Attempting to use smaller width would require the full evaluation of the epilepsy prediction algorithm and the code that performs the epilepsy prediction stage was not accessible due to the NDA. The latency was selected to be equal to 19 clock cycles so that the divider can meet the time constraints on the SHMAC platform. After the divider selection, it is evident that the operand scaling will have to be again present. Furthermore, it needs to be noted that the 32-bit fixed-point divider is bigger and slower than the 32-bit floating-point divider. One would expect that floating-point components are always more complicated than corresponding fixed-point components. In case of the fixed-point division, the division has to be performed on all 32-bits, while in floating-point the division is performed only on the operands' mantissas (the exponents are simply subtracted).

Figure 4.3 illustrates the way that the operand scaling is performed. This example uses 16-bit values that have to be scaled (truncated) in 8-bit values, nevertheless the same reasoning was used in order to obtain the 32-bit values (inputs of the divider) from the initial 64-bit values (results of multiplications). As discussed earlier, the leading and the trailing zeros of the operands are detected asynchronously every time that a new operand is calculated.

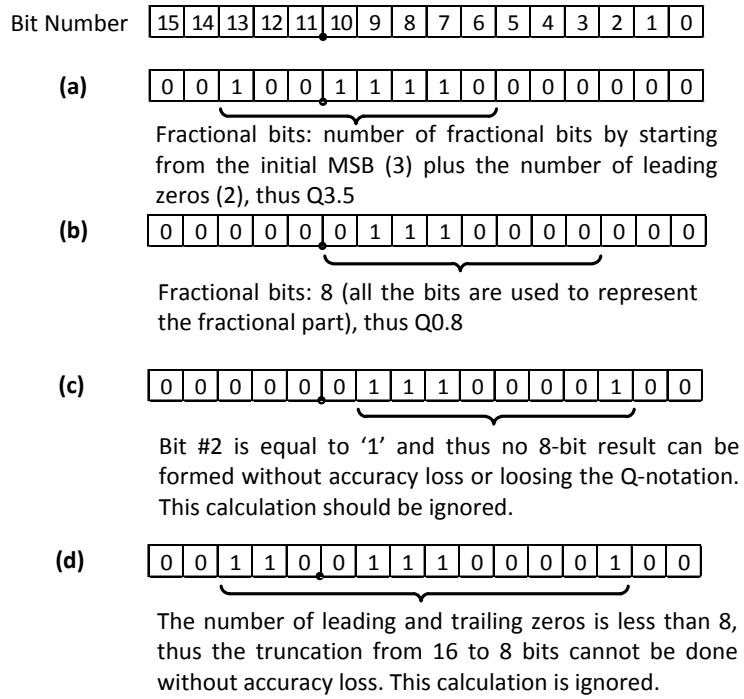


Figure 4.3: Truncation of a 16-bit value in $Q_{16.11}$ format in 8 bits, **(a)** , **(b)** the truncation can be performed without any problem, **(c)** , **(d)** truncation results in loss of accuracy.

In this example, the 8-bit variables can be obtained without losing precision only if the sum of leading and trailing zeros is at least 8 (the bits that will be "kicked out" of the number). If

the truncation can be performed, then additional operations that determine the *Q*-notation of the final value have to be performed, otherwise the current calculation has to be ignored (Figure 4.3 (d)). The final *Q*-notation is obtained according to the binary point and the MSB position of the truncated number. If the MSB is on the left hand side of the binary point, then the first 8-bits starting from it, will form the final 8-bit value (Figure 4.3 (a)). If it is on the right hand side of the binary point, then it must be ensured that all the 8-bits will represent a pure fraction and thus the next 8-bits after the binary point are kept (Figure 4.3 (b)). If the *Q*-notation is lost (Figure 4.3 (c)), then the current calculation should be ignored.

The truncated operands (32-bits) need afterwards to get aligned, so that the comparison between them can be performed. Figure 4.4 illustrates how the alignment is performed. In this example the variables are 16-bit wide, but exactly the same principle applies also in the case of the 32-bit values involved in the actual calculations. The idea behind it is to manipulate again the leading and trailing zeros of the operands and perform corresponding shift operations in order to force them have the same *Q*-notation. If these shift instructions cannot be performed, then the current calculation is ignored. Figure 4.4 illustrates all the possible cases ((a) to (d)) that shifting without precision loss is feasible. The initial operands are presented in the upper part of each sub-figure, and the final aligned operands in the lower part.

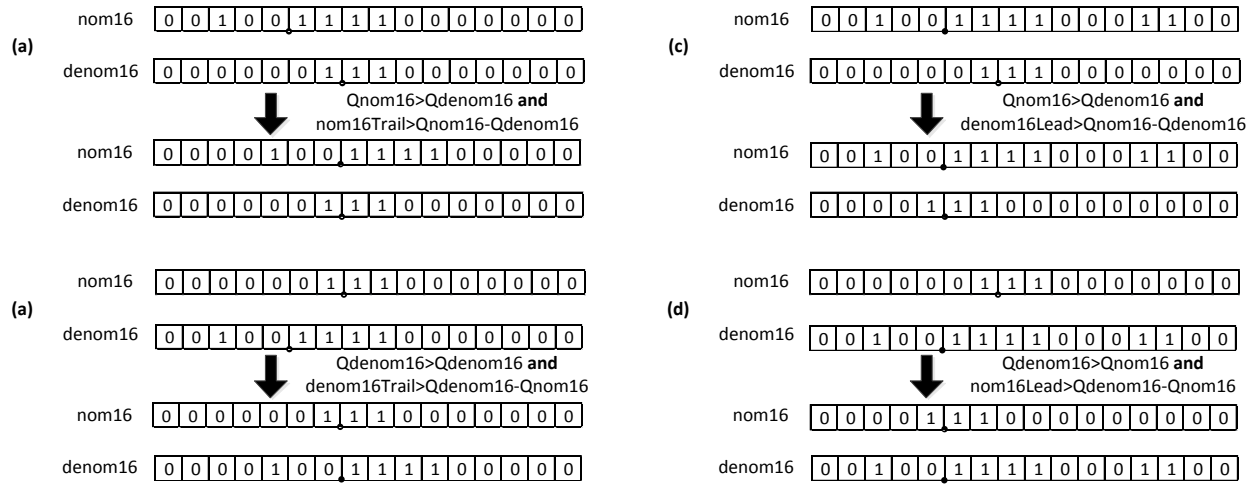


Figure 4.4: Operand alignment (*Q* here denotes only the number of fractional bits).

Taking all the above analysis into consideration about the fixed-point operations, one can now start with their time scheduling and the actual design of the accelerator. For this purpose the states of the FSM need to be defined. Their functionality is the following:

a) Idle State

The accelerator remains idle and no activity takes place in its internal parts. If the *start* signal gets high, *State-1* is triggered, otherwise the accelerator remains in *Idle State*.

b) State-1

Variable *temp* is calculated as stored in a 64-bit register. *denom64* is also calculated by multiplying inputs *delta1* and *deltaf*. At the same time, the sign of *denom* is detected and if it is a negative number it is inverted to a positive one, so that its leading and trailing zeros can be detected. *State-2* is triggered.

c) State-2

Variable *temp* is shifted by two positions to the right, so that it can be used in the calculation of *nom*. If *denom64* can be truncated into a 32-bit number (according to the conditions presented in Figure 4.3), then it gets truncated and *State-3* is triggered next. If not, then an ignore flag is generated and the FSM transits to *State-7* (terminal state).

d) State-3

Variable *nom64* is calculated as the product (*mpoint-temp*)×(*mpoint-temp*) and is stored in a 64-bit register. *State-4* is triggered.

e) State-4

The previously calculated *denom64*, is checked whether it can get truncated into *denom32* (32-bits wide) and if so, *State-5* is triggered. Otherwise *State-7* is triggered and an ignore flag is generated.

f) State-5

The numbers *nom32* and *denom32* are checked whether they can be aligned or not as illustrated in Figure 4.4. If this is possible, *State-6* is triggered next, otherwise an ignore flag is generated and the next state is *State-7*.

g) State-6

During State-6, the aligned operands *nom32* and *denom32* are compared to each other. If *denom32* is bigger than *nom32* then their division has to be performed. *nom32* and *denom32* are shifted to the left and right respectively so that *fraction* will have 14-bits in its fractional part and if the divider is ready to receive new data, then the shifted operands are set to its inputs and *State-7* is triggered. If the operands cannot be shifted by 14-bits in total then *State-7* is triggered again, along with an ignore flag. If however the divider is not ready to receive new data, then the FSM remains in this state. If on the other hand, *denom32* is smaller than

nom32, then *fraction* gets equal to '1' *out_rdy* signal is set high and the next state is *Idle* (since the desired value has been calculated).

h) State 7

State S7 is the last state of the FMS. If during the previous states there was no ignore flag set, then the final value of *fraction* is the output of the divisor. Therefore, the FSM remains in this state until the divisor has the result ready. Once this happens, the *out_ready* signal is set high and the *Idle State* is triggered next. In addition, it is checked whether its sign needs to be reversed, according to whether the sign of one of the previous operands was reversed. If on the other hand an ignore flag was generated during the previously described states, then *fraction* retains its previous value (the input is forwarded to the output) the *out_ready* signal is set high to signal the rest of the system that the calculation is complete and the *Idle State* is triggered.

According to the description of the fixed-point accelerator, it is evident that its design was far more challenging than the hybrid because in the latter case the floating-point components were already predefined by Xilinx. Figure 4.5 summarizes its state transition diagram, while Table 4.3 its main characteristics.

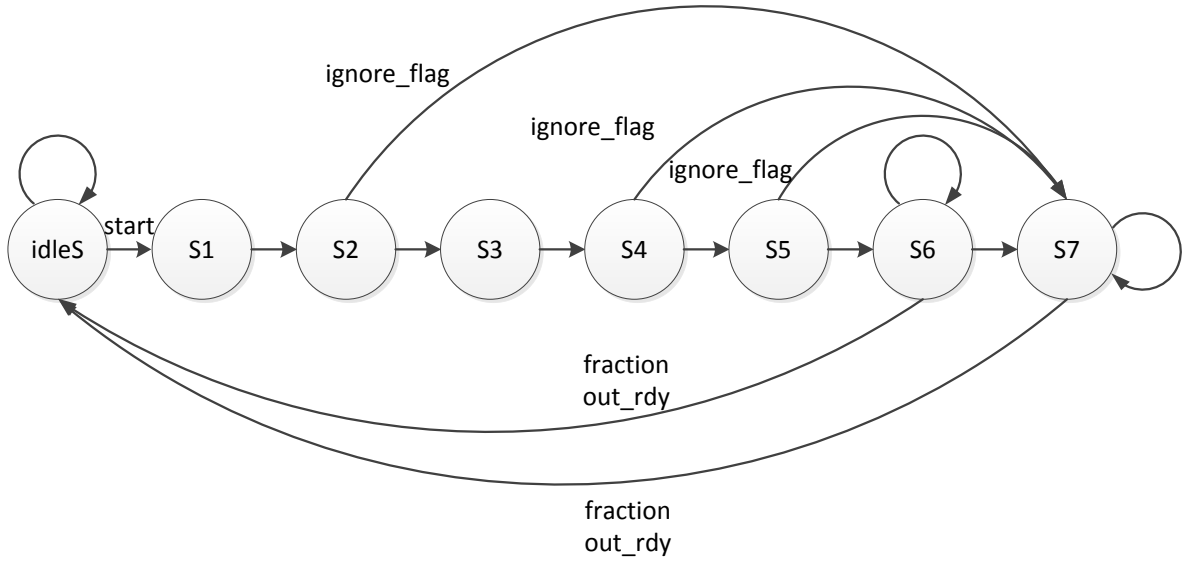


Figure 4.5: State transition diagram of the fixed-point accelerator.

Slice LUTs	DSP slices	Maximum Frequency (MHz)	Dynamic Power (mW)
3877	43	78.382	36

Table 4.4: Area, performance and power consumption of the fixed-point accelerator.

4.3 Verification

The functionality of the designed accelerators was verified by VHDL test-benches executed in a simulator environment. The simulator was Xilinx ISim; despite the fact that this is one of the simplest simulators available, it is embedded in the Xilinx ISE environment that was used for the design of the modules and since the designed accelerators are not complex components that require more advanced options during the simulation, ISim was preferred. The hybrid accelerator was verified first.

The input values that it requires and the output that it calculates were extracted during the execution of the pure software version and recorded in a text file (hexadecimal values). This file was later used as input to the test-bench, in which the hybrid accelerator was simulated with the recorded values as inputs and its calculated result (*fraction*) compared to the recorded value by the software. With this way, the proper functionality can be verified if the results calculated by the accelerator during the simulation are the same with the results produced by the software. Out of the total 140.297 (as discussed in Section 2.6) results that need to be calculated a subset of 10.000 was used to perform the verification. Although that this does not guarantee that the module has no bugs at all, it strongly indicates that it can be synthesized, integrated in the system, and then it can be afterwards re-verified by executing the STLmax calculation algorithm and by comparing the final calculated value to the corresponding value calculated by the software. Figure 4.6 illustrates a snapshot during the simulation of the accelerator. In this figure it is worth observing that the result (*fraction*) is calculated 26 clock cycles after the *i_start* signal gets high. At this time, signal *o_rdy* gets high and the *fraction* maintains its value until it gets updated after the next execution.

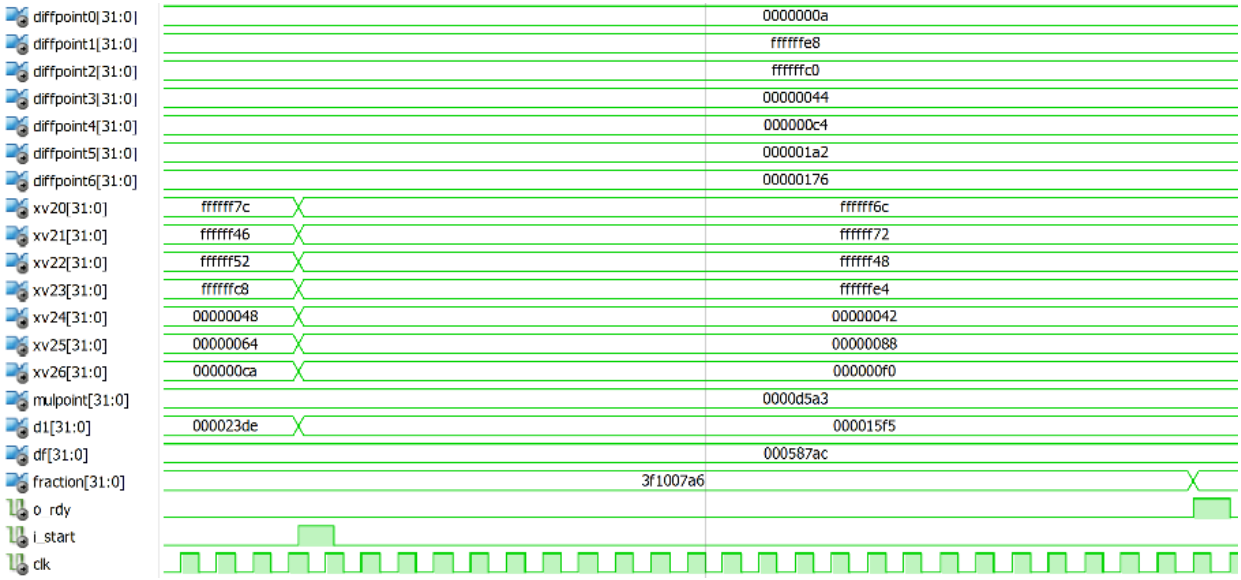


Figure 4.6: Waveform from the simulation of the hybrid accelerator.

For the verification of the fixed-point accelerator, a different technique was followed. During the execution of the fixed-point software version, there exist values that result in ignoring the current calculation, values that make variable *fraction* get equal to '1' and values that result in using the division operation. When the corresponding operations are executed in the accelerator, they can result in different behavior (i.e. a value that results in ignoring the current calculation in software, can result in performing division in the accelerator) due to the fact that the accelerator uses 64-bit values to represent the intermediate results and then truncates them in 32-bit values in order to perform the division. In the software implementation, there exist only 32-bit values because the target is a 32-bit processor, and furthermore during the project [26], the option to manipulate 64-bit signed integers was not available (there were incompatibilities with the signed values, only 64-bit unsigned values could be handled by the compiler).

Taking into account also the results presented in Tables 4.2 and 4.4, the motivation to move on with the fixed-point accelerator is not as high as with the hybrid. Therefore, the fixed-point accelerator was not thoroughly verified, since this would require time from the next step of the implementation (system integration). A coarse verification was performed in parallel with the design; the accelerator was simulated in ISim for several different input sets, and the output was manually observed, in order to verify if the FSM states change as they should. Although that this does not guarantee the proper function of the accelerator, it is almost certain that no interventions can make it outperform the hybrid accelerator. Figure 4.6 contains a snapshot during the simulation. This illustrates two executions of the accelerator. In the first one, the signal *i_start* gets high and the output *fraction* is ready 26 clock cycles later (this is a case whereas the division has to be performed). In the second execution, the output gets valid 4 clock cycles after the signals *i_start* gets high; this is a result of ignoring the current calculation.

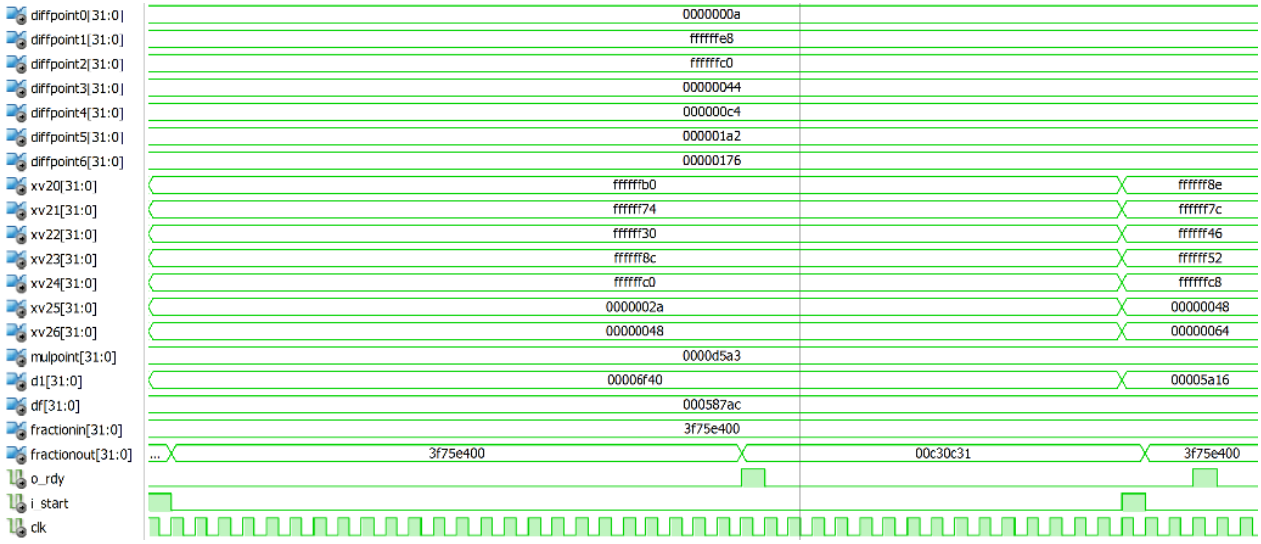


Figure 4.6: Waveform from the simulation of the fixed-point accelerator.

4.4 Comparison of the Designed Modules

Tables 4.2 and 4.3 summarize the main characteristics of the designed modules. In addition to these results, one should also take into account the total latency of each accelerator in order to make an objective comparison between them. This can be directly obtained from the simulation of the modules presented in Figures 4.5 and 4.6. For the hybrid accelerator the latency has the fixed value of 26 clock cycles. For the fixed-point accelerator, the latency varies between 3 and 26 clock cycles, depending on whether the current calculation has to be ignored or whether it requires performing the division. Nevertheless the hybrid accelerator is smaller in terms of LUTs and DSP slices, can operate at a higher frequency and consumes less power than the fixed-point accelerator. Table 4.5 summarizes the main differences between them.

Feature	Hybrid Accelerator	Fixed-Point Accelerator	Difference (%)
Slice LUTs	1508	3877	+157%
DSP slices	24	43	+79%
Max. Frequency (MHz)	86.347	78.382	-9%
Dynamic Power (mW)	24	36	+50%
Latency	26	3-26	-89% - 0%

Table 4.5: Comparison of the designed modules.

One can draw several useful conclusions from the comparison of the modules. First of all it would appear reasonable for someone to assume that the fixed-point accelerator would at least occupy less area and consume less power since it doesn't incorporate any floating-point components. The particular way that the operand scaling is performed in this case must be taken into account; the leading and the trailing zeros of 64-bit and 32-bit operands are detected asynchronously. This means that every time that an operand is detected, multiplexer circuitries need to detect the desired values. This involves 64 cases for the leading zeros of a 64-bit operand and another 64 cases for its trailing zeros than are detected simultaneously. The same principle applies also to the 32-bit variables. It is therefore expected that all these multiplexing operations will require a significant amount of LUTs. It is also very possible that such large combinatorial circuitries create a long critical path and therefore the maximum operating frequency is lower than the hybrid's.

In addition, the fixed-point divider is a lot bigger than the floating-point divider. This comes as a consequence of the fact that in floating-point representation (*IEEE 754* standard) a 32-bit number can be decomposed in three different parts: sign (1-bit), exponent (8-bits) and the mantissa (23-bits). When a division has to be performed, the actual division takes place only on the mantissas; the sign bits and the exponents can be handled otherwise (i.e. XOR operation to obtain the sign and subtraction to obtain the exponent of the quotient). In a corresponding fixed-point division, the division takes place on all the 32-bits and this justifies the necessity for more complicated hardware. Table 4.6 contains the results obtained after the synthesis of two 32-bit

dividers in Xilinx ISE. For comparison reasons, the latency was selected to be the same for both of them (19 clock cycles).

Feature	Floating-Point	Fixed-Point	Difference (%)
Slice LUTs	784	1022	+30%
DSP slices	0	14	NA
FFs	731	1165	+59%
Max. Frequency (MHz)	217.232	360.750	+66%
Latency	19	19	-

Table 4.6: Resource usage and performance of Xilinx 32-bit dividers.

In the previous table it is evident that the fixed-point divider requires many more resources compared to a corresponding floating-point (especially when the DSP slices are concerned). Taking into account the above conclusions, along with the accuracy results that were reported in [26] (the *STLmax* values calculated by the hybrid software version are almost identical to the ground-truth floating-point version, while the calculated values by the fixed-point software have a big variance) it is evident that all the arguments favor the hybrid accelerator for further considerations.

4.5 System Integration

According to the results so far, the hybrid accelerator is the most suitable to be considered for integration within the SHMAC platform. Its input-output requirements consist of 17 input values and 1 output. Such amount of data [33] can be handled by a simple slave interface attached on the Wishbone bus of the SHMAC processor tile (Figure 2.12) that uses memory mapping in order to communicate with the CPU core. This means that the accelerator module should contain some registers, in which the CPU is allowed to have read/write access; write in order to pass the required values to the accelerator and read so that it can read the calculated result. Figure 4.7 illustrates the architecture of a processor tile after the addition of the accelerator, as well as the internal architecture of the accelerator.

By adopting this architecture, the communication can be performed by using pointers in the C code, which refer to the registers of the accelerator. The available memory address space that can be used in this case, consists of the unused tile registers. As discussed in Section 2.3, the tile registers are the addresses FFFE 0000 up to FFFE FFFF (hexadecimal values). Addresses FFFE 0000 up to FFFE 2FFF are occupied by the timers and the interrupt controller, therefore the available address range is from FFFE 3000 up to FFFE FFFF. Furthermore, there should be some registers that perform the synchronization between the CPU and the accelerator. For this purpose, there exist one *options* and one *status* register. Whenever the processor writes on the *options* register, the *start* input signal of the accelerator (Figure 4.1) gets high for one clock cycle

in order to signal the accelerator core that it has to start performing its calculations. The *status* register is used as a polling register that the CPU reads in order to verify that the accelerator has finished its calculations. More specifically, this register can be updated under two conditions. The first one is when the CPU writes the *options* register; in this case the status register gets the value '0'. At this time, the CPU can start polling this register so that when it gets a specific value, the CPU can read the calculated result by the accelerator and continue its operation. Consequently, the second case whereas the *status* register is updated, is when the accelerator finishes its calculations and raises the *out_rdy* signal for one clock cycle. In this case, it gets the value '1' and maintains it until the CPU writes the *options* register again. The Verilog code that implements the discussed system can be found in Appendix C. Despite that the accelerator core was designed in VHDL, the slave interface along with the required registers were designed in Verilog due to the fact that there is plenty of documentation and examples about the Wishbone bus in Verilog. Furthermore, the synthesis tool (Xilinx XST) is able to handle mixed language designs without any problem.

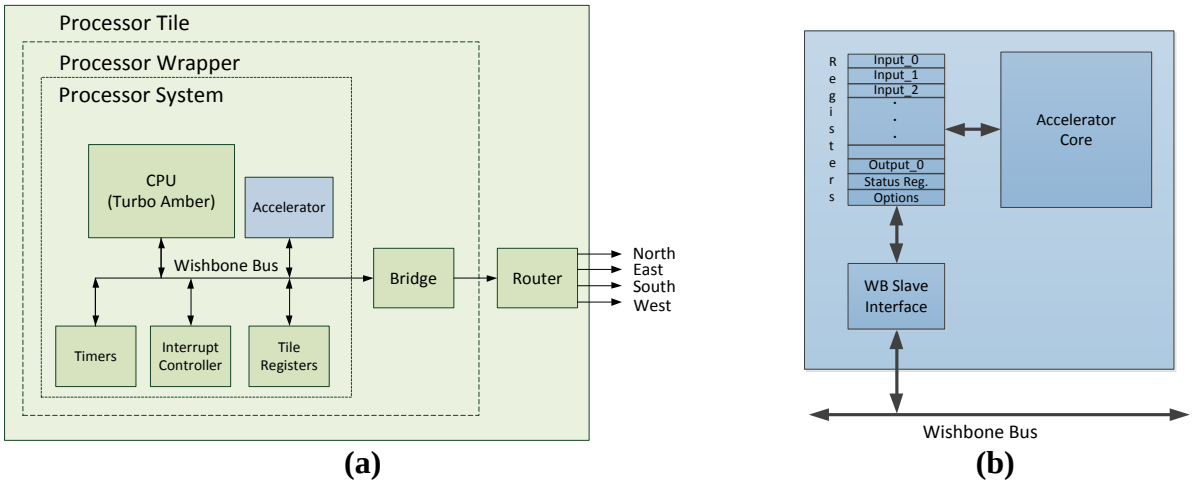


Figure 4.7: (a) Processor tile after the addition of an accelerator, **(b)** accelerator's internal architecture.

It must be noted that synchronization via polling is not an optimal way to perform this task, since the CPU is kept busy without doing anything (it just polls the *status* register) and moreover the performance depends on the polling period as defined in the C-code (Snippet 4.2). A more efficient way would be to use interrupts, this however would require a slightly more complicated system (communication between the CPU, accelerator and interrupt controller). Nevertheless, the interrupts can be adopted in a more refined version of the current system.

```
*OPTIONS=1; //STATUS gets 0
while(*STATUS==0){
    //Wait until the accelerator updates STATUS
}
//Continue
```

Snippet 4.2: Polling as used in the C-code.

Another important feature is that out of the 17 inputs, 9 of them (*VectorB[select]* values along with *deltaf* and *delta1[alfa[i]]*) need to be updated in every iteration (Snippet 3.1). The rest of them (*VectorA[i]* and *mpoint*) are updated in the outer loop (along with the calculation of the *delta1[]* block in Snippet 3.1). This keeps the communication cost between the CPU and the accelerator as low as possible and increases the performance compared to updating all the inputs in every iteration (140.390 iterations in total).

After the addition of the accelerator within the processor tile, the software of the algorithm was adapted accordingly. All the calculations of the *epslion2* block were replaced by simple read and write instructions to the accelerators registers and the algorithm was executed again in order to verify the correct function of the system. The final *STLmax* value was found to be exactly the same as the value calculated by the pure software execution without the accelerator (*STLmax* equal to 3.5064). This strongly indicates that the accelerator performs the required calculations without any problems and verifies its proper function.

Chapter 5

Results

This Chapter presents the results after executing the *STLmax* calculation algorithm on a SHMAC processor tile that includes the earlier discussed accelerator, compared to corresponding results when the algorithm is executed on a generic processor tile. More specifically, these results include the metrics of performance, area usage, power and energy consumption as well as the energy efficiency.

5.1 Performance

Figure 5.1 illustrates the runtimes, when the hybrid version of the *STLmax* calculation algorithm is executed on a processor tile with Turbo-Amber as CPU (pure software implementation), and when it is executed on a modified tile that includes Turbo-Amber and an additional accelerator (mixed hardware-software implementation).

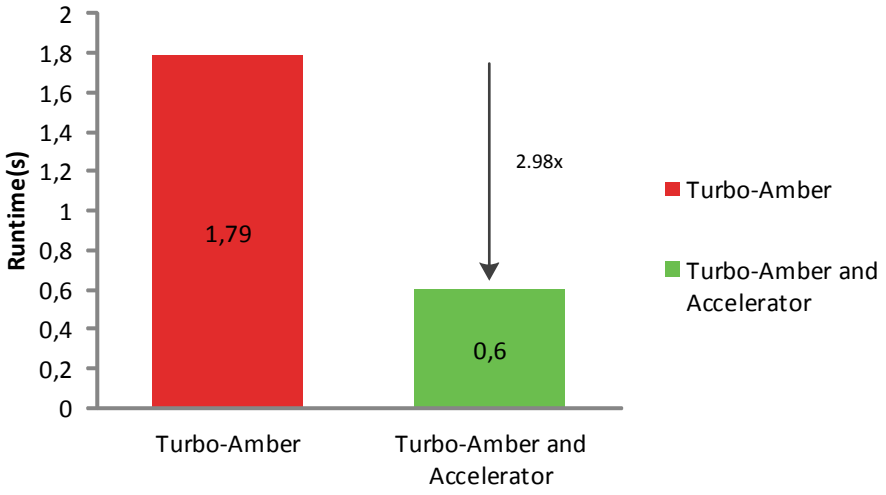


Figure 5.1: Performance results of SW and mixed HW/SW implementations.

In this figure, it is evident that the addition of the accelerator speeds the application up significantly (almost three times faster after the addition of the accelerator). The total execution time is reduced by 1.19 s or by 66%. Considering the fact that in software, the block *epsilon2*

takes 1.27 s and that the runtime reduction is a result of moving it into the accelerator, its total runtime in hardware is the difference 1.27 – 1.19 = 0.08 s. Despite the fact that the applications real-time requirements are still violated (the target runtime is 0.32s), a major step towards to meeting them has been performed.

5.2 Area Usage

After the addition of the accelerator it is reasonable that the overall system occupies more resources on the FPGA than without it. Table 5.1 contains the total number of occupied resources, while Table 5.2 the relative resource utilization on the Virtex-5 FPGA. Design 1 is the synthesis result of a SHMAC layout that includes an APB, a processor (Turbo-Amber) and a main memory tile, as illustrated in Figure 3.3. Design 2 is the synthesis result of the same layout, with the difference that the processor tile contains additionally the accelerator. The resources that were selected to be presented here are the logic slices (which contain the LUTs and the flip-flops of the FPGA [36]) and the DSP slices. The rest of the resources are the same for these two designs and thus they are omitted. Nevertheless the synthesis reports of both of them can be found in Appendix D.

Resource	Virtex-5	Design 1	Design 2
Logic Slices	51 850	9 720	10 796
DSP Slices	192	0	24

Table 5.1: Total number of occupied resources for the two designs.

Resource	Virtex-5	Design 1	Design 2
Logic Slices	51 850	19%	21%
DSP Slices	192	0%	13%

Table 5.2: Relative resource utilization with respect to the total available resources.

Design 2 contains 1076 extra logic slices and 24 DSP slices than Design 1. According to the numbers presented in Table 5.2, the resource utilization can be considered balanced; the FPGA resources are not drained for neither of the designs. This makes the synthesis of several tiles that include the accelerator feasible. Assuming that the real-time requirements could be met by using two tiles, each of which calculating the *STLmax* values of 16 EEG channels, the FPGA resources are still more than sufficient.

5.3 Power Consumption

Making accurate power measurements on SHMAC is a challenging task with the current infrastructure. As discussed in Section 2.3, SHMAC was instantiated on the ARM RealView

Versatile platform. This does not allow direct measurements of the power consumed by the FPGA (i.e. by measuring the current of a shunt resistor between the supply voltage and the FPGA). Two approaches were considered for the purpose of this task. The first one is to obtain power estimates according to Xilinx Power Estimator (XPE) [42]. XPE uses the map-report (*.mrp* file) that is created and updated throughout all the synthesis steps (synthesis, mapping, place and route etc.) of SHMAC, and according to its data, it estimates the power demands of the system. The user needs however to provide the tool the operating frequency (60 MHz for SHMAC) and the signal toggle rates. The latter parameter is difficult to be estimated, and therefore the default toggle rates (as defined by XPE) were used. Table 5.3 summarizes the results obtained by XPE.

	Design 1	**Design 2**	**Difference (%)**
Static Power (W)	3.081	3.081	0%
Dynamic Power (W)	0.323	0.360	+11%
Total Power (W)	3.404	3.441	+1%

Table 5.3: Power estimates obtained by XPE.

The static power of both the designs is the same, since they are implemented on the same FPGA. Design 2 requires 37mW or 11% more dynamic power than Design 1, for the needs of the accelerator (logic slices, DSP slices and signal/clock routing). The total power is however increased by only 1% because it is compensated by the contribution of the static power which is approximately nine times larger than the dynamic power in both designs. This is expected as the target Virtex-5 FPGA (XC5VLX330-ff1760-1) is one of the largest among the Virtex-5 family [36], so that it can support the design of MPSoC such as SHMAC.

The second approach is to measure the total power consumed by the ARM RealView Versatile platform. The available equipment at the Department of Computer and Information Science (IDI) was used for this purpose. This is a simple Watt-meter interposed between the power plug and the ARM RealView Versatile platform. The algorithm was executed continuously for several times (1000) so that it can create an as constant as possible power stream in the FPGA which stabilizes the power consumption of the whole system, so that it can be measured by the Watt-meter. It must be noted that the Watt-meter measures only the active power. This task was performed for the pure software and the mixed hardware-software implementations. Table 5.4 contains the results obtained from this procedure.

Platform State	**Measured Power (W)**
Idle	52.50
Algorithm executed in SW (Design 1)	51.10
Algorithm executed in HW-SW (Design 2)	51.60

Table 5.4: Power measurements on ARM RealView Versatile platform.

According to the numbers presented in Table 5.4, the ARM RealView Versatile platform seems to consume more power when SHMAC remains idle (no activity takes place within the FPGA). One would assume that at this state, the whole system should consume the minimum possible power, however the activity that takes place in the host system and the rest of the peripheral components during this state is something that needs further investigations. The execution of the pure software version of the algorithm makes the (overall) system consume 51.10W active power on average, while the mixed implementation (hardware-software) consumes 500mW (or 1%) more due to the accelerator. This is significantly different to the corresponding difference obtained by XPE (37mW), but it can be partially justified due to the signal toggle rates that were used in XPE. Nevertheless the accuracy of the actual power measurement (possible contributions of additional capacitances and inductions to reactive power) is something that should be considered for future work.

5.4 Energy Consumption

Measuring the energy consumption is as challenging as measuring the power consumption, since the energy consumed by an electronic system can be defined by the following equation:

$$E = \int_0^T P(t)dt \tag{14}$$

This means that the energy that needs to be consumed by the system (i.e. CPU) that executes an application, depends on the instantaneous power of the system *P(t)* and the total execution time of the application *T*. If the power *P(t)* is a constant function in time, then Equation (14) can be rewritten as:

$$E = P \times T \tag{15}$$

As discussed in the previous section, the power *P* that was measured during the power measurements of the ARM RealView Versatile platform was the average power (constant), therefore Equation (15) can be used for the calculation of the consumed energy. The numbers presented here correspond to the overall system and not only to the FPGA, nevertheless they can be used to evaluate the difference in energy consumption of the system with and without the accelerator (Design 2 and Design 1 respectively). For the Design 1, Equation (15) results in:

$$E_1 = P_1 \times T_1 = 51.1W \times 1.79s = 91.47J$$

While the corresponding result for the Design 2 is:

$$E_2 = P_2 \times T_2 = 51.6W \times 0.6s = 30.96J$$

It is evident that due to the fact that the addition of the accelerator decreases drastically the total execution time of the algorithm (by a factor of 66%) and increases slightly the power consumption (by a factor of 1%), the consumed energy will be less for the Design 2 (with the accelerator). More specifically, Design 2 consumes 60.51J or 66% less energy than Design 1 for the calculation of the same *STLmax* value.

5.5 Energy Efficiency

Evaluating the energy efficiency depends directly on the corresponding metric. Measuring energy efficiency can be achieved by defining a metric that balances power consumption and performance in an appropriate way [43]. Defining however the notation of 'appropriate way' is not straightforward and therefore there exist several metrics. Among them, the energy-delay product (EDP) is a widely used metric for comparing two designs at the processor level [43] and therefore it is adopted in this thesis for the energy efficiency evaluation of the *STLmax* calculation algorithm. More specifically, the EDPs of Designs 1 and 2 are:

$$EDP_1 = E_1 \times T_1 = 91.47J \times 1.79s = 163.73Js$$
$$EDP_2 = E_2 \times T_2 = 30.96J \times 0.60s = 18.58Js$$

According to the results presented in the above equations, Design 2 is 8.8 times (or 88%) more energy efficient than Design 1, as also illustrated in Figure 5.2. The advantage of utilizing the additional accelerator is clear in this case.

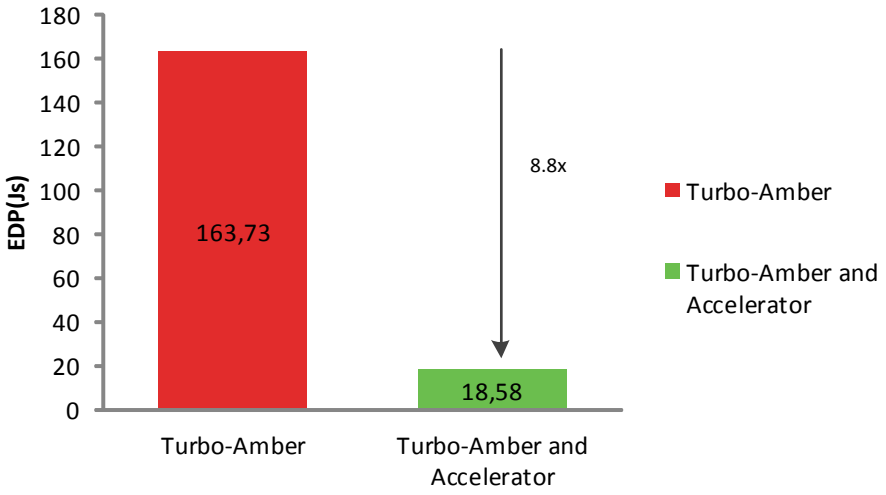


Figure 5.2: Energy efficiency results (in EDP terms) of SW and mixed HW/SW implementations.

Chapter 6

Conclusions and Suggestions for Future Work

This is the final chapter of the current thesis. It discusses the conclusions drawn after the software implementation of the algorithm in [26], along with conclusions regarding the selection of fixed/floating-point format for the current application, the accelerator design and the behavior of the system after the addition of the accelerator. Last but not least, some ideas about potential future work are discussed.

6.1 Conclusions

This thesis describes the implementation of the *STLmax* calculation algorithm on the SHMAC platform. Although that at its end, the application real-time requirements were not met, several useful insights can be concluded from the intermediate steps that have been performed. First of all, in any application mapping on a certain platform, the developer needs to take into account the nature of the platform and adapt the application accordingly. Table 3.3 highlights that the transition from a pure floating-point implementation of the current application into the hybrid (fixed-point/floating-point) version resulted in performance gain by 76% (just by adapting the software that performs the same task).

The methodology that was followed includes the profiling of the application during which the main bottlenecks were identified, and the implementation of the most important one in a hardware accelerator. This resulted in a final hardware-software solution. The addition of the accelerator turns out to have significant impact in many aspects (performance, area usage, power/energy consumption, energy efficiency). For the discussed application, the performance was improved by 66% compared to the already optimized software implementation, the energy efficiency was improved by 88% as well the FPGA resource utilization was kept at an as low as possible level. As long as the design of the accelerator is concerned, that depends on the software block that needs to be moved into the accelerator.

The hybrid approach was found to be very beneficial for this specific application, this however is due to the fact that the floating-point hardware components were taken as granted. If their manual design had been required, then this would have been an extremely time consuming task by itself (especially considering the manual design of a multiplier and a divider). The

accuracy demands in the block that was moved into the accelerator (*epsilon2*) justifies why the floating-point format in some operations was adopted.

Attempting to implement the same block only by using fixed-point arithmetic resulted in inefficient software (slower than the hybrid) as well as hardware (the fixed-point accelerator is a lot bigger than the hybrid). If the operand scaling could have been performed otherwise (i.e. by shifting out *n*-LSBs, regardless whether they are '0' or '1' and sacrificing the accuracy), then the fixed-point implementation would have probably been the most suitable one. Nevertheless the accuracy demands didn't allow such alternatives. Detecting the leading and trailing zeros of the fixed-point operands (64 and 32-bit) resulted in large combinatorial circuitries. In case were the operands would not have been that wide (i.e. 16-bit), then the utilization of the operand scaling might have been more efficient (this however needs to be investigated). In addition, the fixed-point 32-bit division was found to be more complicated than a corresponding floating-point, as a result of the fact than in floating-point format the sign and exponents segments of the number can be excluded from the actual division.

6.2 Suggestions for Future Work

Based on the conclusions drawn so far and looking forward, one can say that there are still many things that should be considered for future work. First of all, as the real-time requirements are not met, it would be interesting to utilize two processor tiles with accelerator in order to meet them. This would require some minimum synchronization mechanism between the two tiles so that each of which can process specific EEG channels (16 channels each). Implementing some other parts of the algorithm in hardware could also provide useful insights about the metrics discussed in Chapter 5 and about whether a single core hardware/software solution is more efficient than a multicore implementation.

Furthermore, the designed accelerator uses polling in order to communicate with the processor core. Replacing polling with interrupts could potentially increase the processor utilization (the processor is prohibited from doing nothing) along with the performance (since polling is dominated by the polling period) therefore this is something that needs to be investigated.

In addition, the Wishbone bus utilized on SHMAC is 128-bits wide, while the CPU is 32-bit. When the CPU needs to write data to some peripheral device (i.e. accelerator connected via a slave interface) it has to write 32-bits on each operation (the rest 86-bits of the bus cannot be used). Including CPU cores with SIMD features would allow the full utilization of the Wishbone bus and this could reduce the communication overhead between the CPU and the peripherals/accelerators. This would also increase the heterogeneity degree of SHMAC.

Regarding the heterogeneity, it would worth considering different clock domains on future accelerators. The current CPU frequency is 60MHz, and therefore clocking the accelerators at submultiples of this frequency (i.e. 30MHz, 15MHz etc.) can be performed

relatively easy. The impact of creating different clock domains on the overall SHMAC platform should be investigated. Can the power consumption be reduced by adopting this technique or the overhead for routing the extra clock eliminates this potential? If yes, then this could possibly allow the adoption of dynamic frequency scaling in certain applications. Dynamic voltage scaling can not be considered at the time being, as SHMAC is realized on an FPGA. Creating a reliable power measurement infrastructure is also something that is worth considering further investigations. At the time being, there is no way to measure accurately the power consumed by the FPGA; the only way is to perform measurements on the whole developing platform which includes several other components besides the FPGA that contains SHMAC.

Last but not least, further investigations about the operand scaling in fixed-point arithmetic could also be considered for future work. The available literature is very limited, and this makes the selection of an optimal technique that performs this task technically impossible. Considerations for instance regarding how the leading and trailing zeros can be detected as well as a completely new technique if it is feasible could result in less complicated fixed-point circuits.

Appendix A

Fixed-Point Mathematical Operations

- **Change of exponent**

Given that a number α is represented in $k \cdot 2^{-q_1}$ format, it is possible to change the exponent q_1 to another exponent q_2 by a proper shifting operation as follows:

$$\alpha = \begin{cases} \alpha << (q_2 - q_1) & if \quad q_2 > q_1 \\ \alpha << (q_2 - q_1) & if \quad q_2 < q_1 \end{cases}$$

The change of exponent must be handled very carefully because each shifting can lead to a possible overflow [16]. For example if there are 16bits available and the number 512 is represented in $Q_{10.5}$ format (exponent is equal to 5), it cannot be changed to the exponent 7, because the two additional bits that will be used for the fractional part will be taken from the integer part which in this case will end up having 8-bits, and with 8-bits the value 512 can't be represented.

- **Addition/Subtraction**

As it is discussed in [17] adding two numbers represented in $Q_{1.7}$ and $Q_{2.6}$ format respectively, will result in a number in $Q_{2.6}$ format (provided that there are 8 bits available). It is thus recommended to transform the numbers in the same format (by changing the exponent of one of them) [16] so that the result will also be in the same format. The addition is then done as in the following formula:

$$Q_{2.6} + Q_{2.6} = Q_{2.6}$$

Or equivalently (for the $k \cdot 2^{-q}$ representation [10]):

$$k_1 \cdot 2^{-q} + k_2 \cdot 2^{-q} = (k_1 + k_2) \cdot 2^{-q}$$

Special attention is also needed here for possible overflows. An overflow can easily be detected by checking if the carry-in bit into the addition of the MSBs is different than the carry-out bit of it after the addition, by saving the result of the addition of two *n*-bit binary variables in a *2n*-bits variable, and checking if it exceeds the available *n*-bits [17] (simply by checking if there are any "1"s after the *n*-th bit), or by checking the sing bit before and after the operation. If for instance we try to add two positive numbers (both of them have sign bit equal to zero) and the result is a negative number (sign bit equal to one), then we have overflow. Subtraction is performed in a similar manner.

- **Multiplication**

Multiplication can be performed as in the following equation:

$$Q_{n1.m1} \cdot Q_{n2.m2} = Q_{n1+n2.m1+m2}$$

Or equivalently:

$$(k_1 \cdot 2^{-q1}) \cdot (k_2 \cdot 2^{-q2}) = (k_1 \cdot k_2) \cdot 2^{-(q1+q2)}$$

The product of two *n*-bit variables requires *2n*-bits to be represented so that there is no precision loss. Such operations are very challenging because, in order to perform successfully the multiplication, a *2n*-bits variable is required or in another case the exponent of the multiplier and the multiplicand will have to be changed to $\frac{n}{2}$ (or some other equivalent values), so that their product will occupy *n*-bits. If for instance, someone tries to multiply the decimal numbers 3.25 and 2.75, and assume that both of them are represented in $Q_{4.4}$ format. The multiplication between them can be seen in Figure A.1.

3.25	0011.0100	$Q_{4.4}$
$\times$2.75	$\times$0010.1100	$\times Q_{4.4}$
8.9375	00001000.11110000	$Q_{8.8}$
(a)	(b)	

Figure A.1: (a) Multiplication in decimal system, **(b)** Multiplication in binary system.

As it can be seen in Figure A.1 (b), the product occupies twice the number of bits than the multiplier and the multiplicand. In order to avoid the necessity of a 16-bits variable, the multiplier and the multiplicand can be scaled (change the number of bits that represent the integer and the decimal part) before performing the multiplication (Figure A.2).

$$\begin{array}{rr} 0011.0100 & Q_{4.4} \\ \times 0010.1100 & \times Q_{4.4} \\ \hline 00001000.11110000 & Q_{8.8} \end{array} \rightarrow \begin{array}{rr} 11.01 & Q_{2.2} \\ \times 10.11 & \times Q_{2.2} \\ \hline 1000.1111 & Q_{4.4} \end{array}$$

Figure A.2: Operand scaling before the multiplication.

The scaling of the operands requires special care when used. Shifting out zeros from the beginning or the end of the initial operands (as in Figure A.2) is safe for the accuracy of the result. Otherwise (if it is necessary to shift out ones) there will be an accuracy penalty and thus the user must be very careful when using such techniques. Another hardware solution that is used by some processors, offers the utilization of *2n*-bits registers for the intermediate results [18]. When a multiplication has to be performed the processor can save temporally the *2n*-bits result, and then remove the *n* least significant bits, so that the final result will occupy *n*-bits. This of course leads to accuracy degradation, but for some specific applications it can be acceptable. Overflows are also very dangerous in this case since the format of the product must support values up to $k_1 \cdot k_2$ (in absolute values).

- **Division**

Given two numbers with exponents $q1$ and $q2$ respectively, their division can be performed by applying the following equation [16]:

$$\frac{k_1 \cdot 2^{-q1}}{k_2 \cdot 2^{-q2}} = \frac{k_1}{k_2} \cdot 2^{-q1+q2}$$, where k_1 and k_2 are integer variables

From the above equation, it is obvious that the result depends on the integer division of k_1 and k_2, and as a consequence there will be some loss of precision. Therefore it needs special attention that this operation will be done by such a manner that the loss of precision will be minimal [16][18]. This minimization of accuracy loss can be performed by proper shifting of the nominator and denominator. This will have an impact on the exponent of the quotient, especially in the case where $q1 = q2$ and the exponent of the quotient is zero (the quotient has only integer part and no fractional). Considering for instance the division between the decimal numbers 3.5 and 2.0, whereas both of them are in $Q_{4.4}$ format, 3.5 is represented as 0011.1000 and 2.0 as 0010.0000. The corresponding values of k_1 and k_2 (integer values by omitting the binary point) are 56 and 32 respectively. The integer division between these values will have only the integer part of the quotient as illustrated in Figure A.3.

3.5	56
÷2.0	÷32
1.75	1
(a)	(b)

Figure A.3: (a) Division in decimal system, **(b)** Integer division of k_1 and k_2 values.

From the previous example, it is evident that the accuracy penalty is significant. One way to improve the accuracy is by increasing the exponent of the dividend (simply by shifting the dividend by 8-bits to the left) but this means that the updated value of the dividend will now occupy 16-bits. The division in this case is:

$$\begin{array}{r} 56 \cdot 2^8 \\ \div\ 32 \\ \hline 448 \end{array}$$

The quotient now has 8-bits in the decimal part because of the 8-positions left shifting of the dividend. The decimal value that number 448 represents can be found by shifting 448 8-positions to the right. This gives the value 1.75 which is the exact value. It is important to keep in mind that the computer keeps only the value 448 which is an integer value. It is the user who has to select the shifting amount of the dividend and remember it so that he/she knows the corresponding decimal values. If both the dividend and the divisor are represented by *n*-bits, it turns out that full precision can be achieved by shifting the dividend by *n*-bits on the left, and then perform the division. If the dividend can't be shifted by *n*-bits but by a smaller number there will be some accuracy loss. Another technique to keep accuracy as high as possible is similar to what was described in the multiplication operation. Zeros can be shifted out from the beginning of the dividend (which will increase its exponent q_1) and from the end of the divisor (which will decrease its exponent q_2) . The goal in this case is to keep the quantity $-q_1 + q_2$ as high as possible (in absolute value), because it determines the number of fractional bits of the quotient and thus its accuracy. If this quantity is equal to *n* then the division can be performed with no loss of accuracy.

Appendix B

VHDL Code

B.1 Hybrid Accelerator

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
----------------------------------------------------

entity epsilon2 is
port(   clk          :  in  std_logic;
        rst          :  in  std_logic;
        i_start      :  in  std_logic;
        diffpoint0   :  in  signed(31 downto 0);
        diffpoint1   :  in  signed(31 downto 0);
        diffpoint2   :  in  signed(31 downto 0);
        diffpoint3   :  in  signed(31 downto 0);
        diffpoint4   :  in  signed(31 downto 0);
        diffpoint5   :  in  signed(31 downto 0);
        diffpoint6   :  in  signed(31 downto 0);
        xV20         :  in  signed(31 downto 0);
        xV21         :  in  signed(31 downto 0);
        xV22         :  in  signed(31 downto 0);
        xV23         :  in  signed(31 downto 0);
        xV24         :  in  signed(31 downto 0);
        xV25         :  in  signed(31 downto 0);
        xV26         :  in  signed(31 downto 0);
        mulpoint     :  in  signed(31 downto 0);
        d1           :  in  std_logic_vector(31 downto 0);
        df           :  in  std_logic_vector(31 downto 0);
        o_rdy        :  out std_logic;
        fraction     :  out std_logic_vector(31 downto 0)
);
end epsilon2;

----------------------------------------------------

architecture behavior of epsilon2 is

signal result1   : signed(31 downto 0) := (others => '0');
signal resultTemp: signed(63 downto 0) := (others => '0'); --intermediate
multiplication result 32x32=>64 bits
type state is (idleS, S1, S2, S3, S4, S5, S6, S7);
signal presentState : state;


------------Components generated with CoreGenerator----------------------
```

```
COMPONENT fixedTofloat
PORT (
     clk             : IN STD_LOGIC;
     a               : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
     operation_nd    : IN STD_LOGIC;
     operation_rfd   : OUT STD_LOGIC;
     result          : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
     rdy             : OUT STD_LOGIC
);
END COMPONENT;


COMPONENT floatMultiplier
PORT (
     a               : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
     b               : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
     operation_nd    : IN STD_LOGIC;
     operation_rfd   : OUT STD_LOGIC;
     clk             : IN STD_LOGIC;
     result          : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
     rdy             : OUT STD_LOGIC
);
END COMPONENT;

COMPONENT floatComparator
 PORT (
     a               : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
     b               : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
     operation_nd    : IN STD_LOGIC;
     operation_rfd   : OUT STD_LOGIC;
     clk             : IN STD_LOGIC;
     result          : OUT STD_LOGIC_VECTOR(0 DOWNTO 0);
     rdy             : OUT STD_LOGIC
  );
END COMPONENT;

COMPONENT floatDivider
  PORT (
     a               : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
     b               : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
     operation_nd    : IN STD_LOGIC;
     operation_rfd   : OUT STD_LOGIC;
     clk             : IN STD_LOGIC;
     result          : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
     rdy             : OUT STD_LOGIC
  );
END COMPONENT;

signal nomtemp       : std_logic_vector(31 downto 0) := (others => '0');
signal d1float       : std_logic_vector(31 downto 0) := (others => '0');
signal dffloat       : std_logic_vector(31 downto 0) := (others => '0');
signal dftemp        : std_logic_vector(31 downto 0) := (others => '0');
signal nomfloat      : std_logic_vector(31 downto 0);
signal denomfloat    : std_logic_vector(31 downto 0);
signal nomfloat2     : std_logic_vector(31 downto 0);

--Inputs - Outputs   fixed2floatConverter
```

```
    signal converterIN  : std_logic_vector(31 downto 0);
    signal converterOUT : std_logic_vector(31 downto 0);
    signal rdy1         : std_logic;
    signal nd1          : std_logic;
    signal rfd1         : std_logic := '0';

    --Inputs - Outputs   floatMultiplier
    signal multiplierA  : std_logic_vector(31 downto 0);
    signal multiplierB  : std_logic_vector(31 downto 0);
    signal multiplierRes: std_logic_vector(31 downto 0);
    signal rdy2         : std_logic;
    signal rfd2         : std_logic;
    signal nd2          : std_logic := '0';

    --Inputs - Outputs   floatComparator
    signal comparatorA  : std_logic_vector(31 downto 0):= (others => '0');
    signal comparatorB  : std_logic_vector(31 downto 0):= (others => '0');
    signal comparatorRes: std_logic_vector(0 DOWNTO 0):= (others => '0');
    signal rdy3         : std_logic;
    signal rfd3         : std_logic;
    signal nd3          : std_logic := '0';

    --Inputs - Outputs   floatDivider
    signal dividerA     : std_logic_vector(31 downto 0):= (others => '0');
    signal dividerB     : std_logic_vector(31 downto 0):= (others => '0');
    signal dividerRes   : std_logic_vector(31 downto 0):= (others => '0');
    signal rdy4         : std_logic;
    signal rfd4         : std_logic;
    signal nd4          : std_logic := '0';

    signal temp         : std_logic_vector(3 downto 0)  := (others => '0');
    signal temp1        : std_logic_vector(31 downto 0) := (others => '0');
    signal temp2        : std_logic_vector(31 downto 0) := (others => '1');

    --Signals used in the FSM to define the state transitions
    signal holdS1       : std_logic_vector(1 downto 0):= (others => '0');
    signal holdS2       : std_logic_vector(1 downto 0):= (others => '0');
    signal holdS3       : std_logic_vector(2 downto 0):= (others => '0');
    signal holdS4       : std_logic_vector(1 downto 0):= (others => '0');
    signal holdS5       : std_logic_vector(1 downto 0):= (others => '0');
    signal holdS6       : std_logic_vector(1 downto 0):= (others => '0');
    signal holdS7       : std_logic;
--------------------------------------------------------------------------------
----------Registered signals according to the 'two-process' method----------

     type registers is record
        converterIN    :  std_logic_vector(31 downto 0);
        converterOUT   :  std_logic_vector(31 downto 0);
        rfd1           :  std_logic;
        nd1            :  std_logic;
        multiplierA    :  std_logic_vector(31 downto 0);
        multiplierB    :  std_logic_vector(31 downto 0);
        multiplierRes  :  std_logic_vector(31 downto 0);
        rfd2           :  std_logic;
        nd2            :  std_logic;
        comparatorA    :  std_logic_vector(31 downto 0);
        comparatorB    :  std_logic_vector(31 downto 0);
```

```
        comparatorRes   :  std_logic_vector(31 downto 0);
        rfd3            :  std_logic;
        nd3             :  std_logic;
        dividerA        :  std_logic_vector(31 downto 0);
        dividerB        :  std_logic_vector(31 downto 0);
        dividerRes      :  std_logic_vector(31 downto 0);
        rfd4            :  std_logic;
        nd4             :  std_logic;
        o_rdy           :  std_logic;
        temp            :  std_logic_vector(3 downto 0);
        resultTemp      :  signed(63 downto 0);
        dftemp          :  std_logic_vector(31 downto 0);
        holdS2          :  std_logic_vector(1 downto 0);
        result1         :  signed(31 downto 0);
        d1float         :  std_logic_vector(31 downto 0);
        holdS3          :  std_logic_vector(2 downto 0);
        dffloat         :  std_logic_vector(31 downto 0);
        holdS4          :  std_logic_vector(2 downto 0);
        holdS5          :  std_logic_vector(1 downto 0);
        holdS6          :  std_logic_vector(1 downto 0);
        holdS7          :  std_logic;
        nomfloat        :  std_logic_vector(31 downto 0);
        denomfloat      :  std_logic_vector(31 downto 0);
        nomfloat2       :  std_logic_vector(31 downto 0);
        fraction        :  std_logic_vector(31 downto 0);
        e2              :  std_logic_vector(31 downto 0);
        presentState    :  state;
    end record;
    signal r, rin : registers;
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------

begin

    Converter: fixedTofloat
    PORT MAP (
     a              => converterIN,
     operation_nd   => nd1,
     operation_rfd  => rfd1,
     clk            => clk,
     result         => converterOUT,
     rdy            => rdy1
    );

    floatMul : floatMultiplier
    PORT MAP (
     a              => multiplierA,
     b              => multiplierB,
     operation_nd   => nd2,
     operation_rfd  => rfd2,
     clk            => clk,
     result         => multiplierRes,
     rdy            => rdy2
    );

  floatDiv : floatDivider
  PORT MAP (
```

```
    a               => dividerA,
    b               => dividerB,
    operation_nd    => nd4,
    operation_rfd   => rfd4,
    clk             => clk,
    result          => dividerRes,
    rdy             => rdy4
   );

  floatComp : floatComparator
  PORT MAP (
    a               => comparatorA,
    b               => comparatorB,
    operation_nd    => nd3,
    operation_rfd   => rfd3,
    clk             => clk,
    result          => comparatorRes,
    rdy             => rdy3
   );


   -- Calculate the values (outputs, internal signals)
   -- in two processes: sequential + combinatorial

   sequential : process(clk)
   begin
       if rising_edge(clk) then r <= rin; end if;
   end process;


   combinatorial : process(rst, i_start, rfd1, rdy1, rfd2, rdy2, rfd3, rdy3,
rfd4, rdy4, r)
       variable v  : registers;
   begin
         v := r;
           if(rst='1') then
               v.presentState := idleS;
           end if;
           if(i_start='1') then
               v.presentState := S1;
           end if;

           case r.presentState is
           when idleS =>
               v.nd1       := '0';
               v.nd2       := '0';
               v.nd3       := '0';
               v.nd4       := '0';
               v.o_rdy     := '0';
               if(i_start = '1') then
                   v.presentState := S1;
               else
                   v.presentState := idleS;
               end if;


           when S1 =>
```

```
                v.o_rdy     := '0';
                v.nd1       := '0';
                v.nd2       := '0';
                v.nd3       := '0';
                v.nd4       := '0';
                if(rfd1='1') then
                    v.resultTemp
:=diffpoint0*xV20+diffpoint1*xV21+diffpoint2*xV22+diffpoint3*xV23+diffpoint4*
xV24+diffpoint5*xV25+diffpoint6*xV26;
                    v.dftemp        := "00" & df(31 downto 2);
                    v.converterIN   := d1;
                    v.nd1           := '1';
                    v.presentState  := S2;
                else
                    v.nd1           := '0';
                    v.presentState  := S1;
                end if;
                v.o_rdy     := '0';
                v.holdS2    := "00";


            when S2 =>
                v.o_rdy     := '0';
                v.nd1       := '0';
                v.nd2       := '0';
                v.nd3       := '0';
                v.nd4       := '0';
                v.result1   :=
r.resultTemp(31)&r.resultTemp(31)&r.resultTemp(31 downto 2);
                if(rdy1='1') then
                    v.d1float   := converterOUT;
                    v.holdS2(0) :='1';
                end if;
                if(rfd1='1') then
                    v.converterIN := r.dftemp;
                    v.nd1         := '1';
                    v.holdS2(1)   := '1';
                end if;

                if(r.holdS2="11") then
                    v.presentState  := S3;
                else
                    v.presentState  := S2;
                end if;
                v.holdS3    :="000";


            when S3 =>
                v.o_rdy     := '0';
                v.nd1       := '0';
                v.nd2       := '0';
                v.nd3       := '0';
                v.nd4       := '0';
                if(rdy1='1') then
                    v.dffloat   := r.converterOUT;
                    v.holdS3(0) :='1';
                end if;
```

```
    if(rfd2='1') then
        v.multiplierA := r.d1float;
        v.multiplierB := converterOUT;
        v.nd2         := '1';
        v.holdS3(1)   := '1';
    end if;
    if(rfd1='1') then
        v.converterIN := mulpoint-v.result1;
        v.nd1         := '1';
        v.holdS3(2)   := '1';
    end if;
    if(r.holdS3="111") then
        v.presentState  := S4;
    else
        v.presentState  := S3;
    end if;
    v.holdS4    :="000";


when S4 =>
    v.o_rdy     := '0';
    v.nd1       := '0';
    v.nd2       := '0';
    v.nd3       := '0';
    v.nd4       := '0';
    if(rdy2='1') then
        v.denomfloat    := multiplierRes;
        v.holdS4(0)     := '1';
    end if;

    if(rdy1='1') then
        v.nomfloat2     := converterOUT;
        v.holdS4(1)     := '1';
    end if;

    if(rfd2='1') then
        v.multiplierA   := converterOUT;
        v.multiplierB   := converterOUT;
        v.nd2           := '1';
        v.holdS4(2)     := '1';
    end if;

    if(r.holdS4="111") then
        v.presentState  := S5;
    else
        v.presentState  := S4;
    end if;
    v.holdS5    :="00";


when S5 =>
    v.o_rdy     := '0';
    v.nd1       := '0';
    v.nd2       := '0';
    v.nd3       := '0';
    v.nd4       := '0';
```

```
        if(rdy2='1') then
            v.nomfloat  := multiplierRes;
            v.holdS5(0) := '1';
        end if;

        if(rfd3='1')and(v.holdS5(0) = '1') then
            v.comparatorA := r.denomfloat;
            v.comparatorB := v.nomfloat;
            v.nd3         := '1';
            v.holdS5(1)   := '1';
        end if;

        if(r.holdS5="11") then
            v.presentState  := S6;
            v.o_rdy         := '0';
        else
            v.presentState  := S5;
        end if;
        v.holdS6    :="00";

    when S6 =>
        v.o_rdy     := '0';
        v.nd1       := '0';
        v.nd2       := '0';
        v.nd3       := '0';
        v.nd4       := '0';
        if(rdy3='1') then
            if(comparatorRes="0") then
                if(rfd4='1') then
                    v.dividerA := r.nomfloat;
                    v.dividerB := r.denomfloat;
                    v.nd4      := '1';
                    v.holdS6   := "01";
                else
                    v.holdS6   := "00";
                end if;
            else
                v.holdS6       := "11";
            end if;
        end if;
        if(r.holdS6="01") then
            v.presentState  := S7;
        elsif(r.holdS6="11") then
            v.fraction      := x"3f800000";
            v.o_rdy         := '1';
            v.presentState  := idleS;
        else
            v.presentState  := S6;
        end if;
        v.holdS7    :='0';


    when S7 =>
        v.o_rdy     := '0';
        v.nd1       := '0';
        v.nd2       := '0';
        v.nd3       := '0';
```

```
                v.nd4       := '0';
                if(rdy4='1') then
                    v.fraction      := r.dividerRes;
                    v.o_rdy         := '1';
                    v.presentState  := idleS;
                    v.holdS7        :='1';
                else
                    v.presentState  := S7;
                end if;
            end case;


            v.multiplierRes := multiplierRes;
            v.converterOUT  := converterOUT;
            v.dividerRes    := dividerRes;
            nd1             <= v.nd1;
            nd2             <= v.nd2;
            nd3             <= v.nd3;
            nd4             <= v.nd4;
            converterIN     <= r.converterIN;
            multiplierA     <= r.multiplierA;
            multiplierB     <= r.multiplierB;
            comparatorA     <= r.comparatorA;
            comparatorB     <= r.comparatorB;
            dividerA        <= r.dividerA;
            dividerB        <= r.dividerB;
            rin             <= v;
            fraction        <= r.fraction;
            o_rdy           <= r.o_rdy;
    end process;
end behavior;
```

B.2 Fixed-Point Accelerator

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;
library work;
use work.fixed_point_pkg.all;

entity epsilon2FixedPoint is
port (clk           : in std_logic;
        i_start     :  in std_logic;
        rst         :  in std_logic;
        diffpoint0  :  in signed(31 downto 0);
        diffpoint1  :  in signed(31 downto 0);
        diffpoint2  :  in signed(31 downto 0);
        diffpoint3  :  in signed(31 downto 0);
        diffpoint4  :  in signed(31 downto 0);
        diffpoint5  :  in signed(31 downto 0);
```

```
        diffpoint6  :  in signed(31 downto 0);
        xV20        :  in signed(31 downto 0);
        xV21        :  in signed(31 downto 0);
        xV22        :  in signed(31 downto 0);
        xV23        :  in signed(31 downto 0);
        xV24        :  in signed(31 downto 0);
        xV25        :  in signed(31 downto 0);
        xV26        :  in signed(31 downto 0);
        mulpoint    :  in signed(31 downto 0);
        d1          :  in signed(31 downto 0);
        df          :  in signed(31 downto 0);
        fractionIN  :  in std_logic_vector(31 downto 0);
        fractionOUT :  out std_logic_vector(31 downto 0);
        o_rdy       :  out std_logic := '0'
    );
end epsilon2FixedPoint;

architecture Behavioral of epsilon2FixedPoint is
signal nom64            : signed(63 downto 0) := (others => '0');
signal nom32            : std_logic_vector(31 downto 0);
signal denom64          : signed(63 downto 0) := (others => '0');
signal denom32          : std_logic_vector(31 downto 0);
signal fraction         : std_logic_vector(31 downto 0);
signal fractionTemp     : signed(31 downto 0);
signal nd               : std_logic:= '0';
signal rdy              : std_logic;
signal rfd              : std_logic;


shared variable denomTrail64    : integer range 0 to 64 :=0;
shared variable denomLead64     : integer range 0 to 64 :=0;
shared variable nomTrail64      : integer range 0 to 64 :=0;
shared variable nomLead64       : integer range 0 to 64 :=0;
shared variable nomTrail32      : integer range 0 to 32 :=0;
shared variable nomLead32       : integer range 0 to 32 :=0;
shared variable denomTrail32    : integer range 0 to 32 :=0;
shared variable denomLead32     : integer range 0 to 32 :=0;
signal ignoreFlag               : std_logic:= '0';
signal signDenom                : std_logic:= '0';

signal temp64   : signed(63 downto 0) := (others => '0'); --intermediate
multiplication result 32x32=>64 bits
signal temp32   : signed(31 downto 0) := (others => '0'); --final result
signal temp1    : std_logic_vector(31 downto 0) := (others => '0');
type state is (idleS, S1, S2, S3, S4, S5, S6, S7);
signal presentState, nextState : state;

component fixedPointDivider
    port (
    clk     : in std_logic;
    nd      : in std_logic;
    rdy     : out std_logic;
    rfd     : out std_logic;
    dividend: in std_logic_vector(31 downto 0);
    divisor : in std_logic_vector(31 downto 0);
    quotient: out std_logic_vector(31 downto 0));
end component;
```

```
begin

    fractionOUT <= temp1;

    divider : fixedPointDivider
        port map (
            clk      => clk,
            nd       => nd,
            rdy      => rdy,
            rfd      => rfd,
            dividend => nom32,
            divisor  => denom32,
            quotient => fraction
        );


--  Processes for zeros detection
    denom64Zeros: process(denom64) is
    begin
        denomLead64 :=leading64(std_logic_vector(denom64));
        denomTrail64:=trailing64(std_logic_vector(denom64));
    end process denom64Zeros;

    nom64Zeros: process(nom64) is
    begin
        nomLead64   :=leading64(std_logic_vector(nom64));
        nomTrail64  :=trailing64(std_logic_vector(nom64));
    end process nom64Zeros;

    denom32Zeros: process(denom32) is
    begin
        denomLead32 :=leading32(denom32);
        denomTrail32:=trailing32(denom32);
    end process denom32Zeros;

    nom32Zeros: process(nom32) is
    begin
        nomLead32   :=leading32(nom32);
        nomTrail32  :=trailing32(nom32);
    end process nom32Zeros;

--  Rest of the code (sequential+combinatorial)
    seq: process (rst, clk) is
    begin
        if(rst='1') then
            presentState <= idleS;
        elsif (rising_edge(clk)) then
            presentState <=nextState;
        end if;
    end process seq;

    comb: process(i_start, presentState, rdy, nd, rfd) is
    variable Qdenom      : integer := 0;
    variable Qnom        : integer := 0;
    variable Qfraction : integer := 0;
    begin
        case presentState is
```

```
            when idleS =>
                nd          <= '0';
                o_rdy       <= '0';
                signDenom   <= '0';
                ignoreFlag  <= '0';
                if(i_start = '1') then
                    nextState <= S1;
                else
                    nextState <= idleS;
                end if;


            when S1 =>
                nd          <= '0';
                o_rdy       <= '0';
                temp1       <= fractionIN;
                temp64
<=diffpoint0*xV20+diffpoint1*xV21+diffpoint2*xV22+diffpoint3*xV23+diffpoint4*
xV24+diffpoint5*xV25+diffpoint6*xV26;
                if((d1*df)<x"0000000000000000") then
                    denom64     <= -d1*df;
                    signDenom   <= '1';
                else
                    denom64     <= d1*df;
                    signDenom   <= '0';
                end if;
                Qdenom      := 38;
                ignoreFlag  <= '0';
                nextState   <= S2;


            when S2 =>
                nd          <= '0';
                o_rdy       <= '0';
                temp32      <= temp64(31)&temp64(31)&temp64(31 downto 2);
                if(denomLead64+denomTrail64>=32) then
                    if(denomLead64<26) then
                        denom32     <= std_logic_vector(denom64(63-
denomLead64 downto 32-denomLead64));
                        Qdenom      := 6+denomLead64;
                        ignoreFlag  <= '0';
                        nextState   <= S3;
                    elsif(denomLead64>=26)and(denomTrail64>=6) then
                        denom32     <=std_logic_vector(denom64(37 downto 6));
                        Qdenom      := 32;
                        ignoreFlag  <= '0';
                        nextState   <= S3;
                    else
                        ignoreFlag  <= '1';
                        nextState   <= S7;
                    end if;
                else
                    ignoreFlag  <= '1';
                    nextState   <= S7;
                end if;
```

```
            when S3 =>
                nd          <= '0';
                o_rdy       <= '0';
                nom64       <=(mulpoint-temp32)*(mulpoint-temp32);
                Qnom        := 40;
                ignoreFlag  <= '0';
                nextState   <= S4;


            when S4 =>
                nd      <= '0';
                o_rdy   <= '0';
                if(nomLead64+nomTrail64>=32) then
                    if(nomLead64<24) then
                        nom32       <=std_logic_vector(nom64(63-nomLead64
downto 32-nomLead64));
                        Qnom        := 8+nomLead64;
                        ignoreFlag  <= '0';
                        nextState   <= S5;
                    elsif(nomLead64>=24)and(nomTrail64>=8) then
                        nom32       <=std_logic_vector(nom64(39 downto 8));
                        Qnom        := 32;
                        ignoreFlag  <= '0';
                        nextState   <= S5;
                    else
                        ignoreFlag  <= '1';
                        nextState   <= S7;
                    end if;
                else
                    ignoreFlag  <= '1';
                    nextState   <= S7;
                end if;


            when S5 =>
                nd      <= '0';
                o_rdy   <= '0';
                if(Qnom>Qdenom)and(nomTrail32>=Qnom-Qdenom) then
                    nom32       <=
std_logic_vector(shift_right(unsigned(nom32),Qnom-Qdenom));
                    Qnom        := Qdenom;
                    ignoreFlag  <= '0';
                    nextState   <= S6;
                elsif(Qnom>Qdenom)and(denomLead32>=Qnom-Qdenom) then
                    denom32     <=
std_logic_vector(shift_left(unsigned(denom32),Qnom-Qdenom));
                    Qdenom      := Qnom;
                    ignoreFlag  <= '0';
                    nextState   <= S6;

                elsif(Qdenom>Qnom)and(denomTrail32>=Qdenom-Qnom) then
                    denom32     <=
std_logic_vector(shift_right(unsigned(denom32),Qdenom-Qnom));
                    Qdenom      := Qnom;
                    ignoreFlag  <= '0';
                    nextState   <= S6;
```

```
                elsif(Qdenom>Qnom)and(nomLead32>=Qdenom-Qnom) then
                    nom32       <=
std_logic_vector(shift_left(unsigned(nom32),Qdenom-Qnom));
                    Qnom        := Qdenom;
                    ignoreFlag  <= '0';
                    nextState   <= S6;
                else
                    ignoreFlag  <= '1';
                    nextState   <= S7;
                end if;


            when S6 =>
                nd      <= '0';
                o_rdy   <= '0';
                if(denom32>nom32) then -- no need to invert denom32, cause
nom32 is always positive
                    --division, keep this state until the divider is ready to
receive data
                    if(rfd='1') then
                        if((nomLead32>0)and(denomTrail32>0)) then
                            nom32
<=std_logic_vector(shift_left(unsigned(nom32),7));
                            denom32
<=std_logic_vector(shift_right(unsigned(denom32),7));
                            Qfraction   := 14;
                            nd          <= '1';
                            ignoreFlag  <= '0';
                            nextState   <= S7;
                        else
                            ignoreFlag  <='1';
                            nextState   <= S7;
                        end if;
                    else
                        nd              <= '0';
                        ignoreFlag      <= '0';
                        nextState       <= S6;
                    end if;
                else
                    ignoreFlag      <= '0';
                    o_rdy           <= '1';
                    temp1           <= "00000000000000000100000000000000"; --
"1" in Q14 format
                    nextState       <= idleS;
                end if;


            when S7 =>
                nd      <= '0';
                o_rdy   <= '0';
                temp1   <= fractionIN;
                --wait until the divider has finished
                if(rdy='0') then
                    if(ignoreFlag='0') then
                        nd          <= '0';
                        nextState   <= S7;
                    else
```

```
                        o_rdy       <= '1';
                        temp1       <= fractionIN;
                        nextState   <= idleS;
                    end if;
                else
                    o_rdy <= '1';
                    if(ignoreFlag='0') then
                        if(signDenom='0') then
                            temp1   <=  fraction;
                        else
                            temp1   <=  std_logic_vector(unsigned (not
fraction) + 1);
                        end if;
                    else
                        temp1   <= fractionIN;
                    end if;
                    nextState   <= idleS;
                end if;
        end case;
    end process comb;

end Behavioral;
```

B.3 Zeros Detection VHDL Package

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

package fixed_point_pkg is
     type inputVector is array(0 to 6) of signed (31 downto 0);
     subtype zeros32  is integer range 0 to 32;
     subtype zeros64  is integer range 0 to 64;
     --function declaration.
     function trailing32(x  : std_logic_vector(31 downto 0)) return zeros32;
     function leading32(x   : std_logic_vector(31 downto 0)) return zeros32;
     function trailing64(x  : std_logic_vector(63 downto 0)) return zeros64;
     function leading64(x   : std_logic_vector(63 downto 0)) return zeros64;
end fixed_point_pkg;



package body fixed_point_pkg is

    function trailing32(x : std_logic_vector(31 downto 0)) return zeros32 is
    variable trailingZeros : zeros32;
    begin
        if(x(31 downto 0)=x"00000000") then
            trailingZeros :=32;
        elsif (x(30 downto 0)="00000000000000000000000000000000") then
            trailingZeros :=31;
```

```
elsif (x(29 downto 0)="00000000000000000000000000000000") then
    trailingZeros :=30;
elsif (x(28 downto 0)="00000000000000000000000000000000") then
    trailingZeros :=29;
elsif (x(27 downto 0)="00000000000000000000000000000000") then
    trailingZeros :=28;
elsif (x(26 downto 0)="00000000000000000000000000000000") then
    trailingZeros :=27;
elsif (x(25 downto 0)="00000000000000000000000000000000") then
    trailingZeros :=26;
elsif (x(24 downto 0)="00000000000000000000000000000000") then
    trailingZeros :=25;
elsif (x(23 downto 0)="00000000000000000000000000000000") then
    trailingZeros :=24;
elsif (x(22 downto 0)="00000000000000000000000000000000") then
    trailingZeros :=23;
elsif (x(21 downto 0)="00000000000000000000000000000000") then
    trailingZeros :=22;
elsif (x(20 downto 0)="00000000000000000000000000000000") then
    trailingZeros :=21;
elsif (x(19 downto 0)="00000000000000000000000000000000") then
    trailingZeros :=20;
elsif (x(18 downto 0)="00000000000000000000000000000000") then
    trailingZeros :=19;
elsif (x(17 downto 0)="00000000000000000000000000000000") then
    trailingZeros :=18;
elsif (x(16 downto 0)="00000000000000000000000000000000") then
    trailingZeros :=17;
elsif (x(15 downto 0)="00000000000000000000000000000000") then
    trailingZeros :=16;
elsif (x(14 downto 0)="00000000000000000000000000000000") then
    trailingZeros :=15;
elsif (x(13 downto 0)="00000000000000000000000000000000") then
    trailingZeros :=14;
elsif (x(12 downto 0)="00000000000000000000000000000000") then
    trailingZeros :=13;
elsif (x(11 downto 0)="00000000000000000000000000000000") then
    trailingZeros :=12;
elsif (x(10 downto 0)="00000000000000000000000000000000") then
    trailingZeros :=11;
elsif (x(9 downto 0)="00000000000000000000000000000000") then
    trailingZeros :=10;
elsif (x(8 downto 0)="00000000000000000000000000000000") then
    trailingZeros :=9;
elsif (x(7 downto 0)="00000000000000000000000000000000") then
    trailingZeros :=8;
elsif (x(6 downto 0)="00000000000000000000000000000000") then
    trailingZeros :=7;
elsif (x(5 downto 0)="00000000000000000000000000000000") then
    trailingZeros :=6;
elsif (x(4 downto 0)="00000000000000000000000000000000") then
    trailingZeros :=5;
elsif (x(3 downto 0)="00000000000000000000000000000000") then
    trailingZeros :=4;
elsif (x(2 downto 0)="00000000000000000000000000000000") then
    trailingZeros :=3;
elsif (x(1 downto 0)="00000000000000000000000000000000") then
```

```
        trailingZeros :=2;
    elsif (x(0)='0') then
        trailingZeros :=1;
    else
        trailingZeros :=0;
    end if;
  return trailingZeros;
end trailing32;


function leading32 (x : std_logic_vector(31 downto 0)) return zeros32 is
variable leadingZeros : zeros32;
begin
    if(x(31 downto 0)=x"00000000") then
        leadingZeros := 32;
    elsif (x(31 downto 1)="00000000000000000000000000000000") then
        leadingZeros := 31;
    elsif (x(31 downto 2)="00000000000000000000000000000000") then
        leadingZeros := 30;
    elsif (x(31 downto 3)="00000000000000000000000000000000") then
        leadingZeros := 29;
    elsif (x(31 downto 4)="00000000000000000000000000000000") then
        leadingZeros := 28;
    elsif (x(31 downto 5)="00000000000000000000000000000000") then
        leadingZeros := 27;
    elsif (x(31 downto 6)="00000000000000000000000000000000") then
        leadingZeros := 26;
    elsif (x(31 downto 7)="00000000000000000000000000000000") then
        leadingZeros := 25;
    elsif (x(31 downto 8)="00000000000000000000000000000000") then
        leadingZeros := 24;
    elsif (x(31 downto 9)="00000000000000000000000000000000") then
        leadingZeros := 23;
    elsif (x(31 downto 10)="00000000000000000000000000000000") then
        leadingZeros := 22;
    elsif (x(31 downto 11)="00000000000000000000000000000000") then
        leadingZeros := 21;
    elsif (x(31 downto 12)="00000000000000000000000000000000") then
        leadingZeros := 20;
    elsif (x(31 downto 13)="00000000000000000000000000000000") then
        leadingZeros := 19;
    elsif (x(31 downto 14)="00000000000000000000000000000000") then
        leadingZeros := 18;
    elsif (x(31 downto 15)="00000000000000000000000000000000") then
        leadingZeros := 17;
    elsif (x(31 downto 16)="00000000000000000000000000000000") then
        leadingZeros := 16;
    elsif (x(31 downto 17)="00000000000000000000000000000000") then
        leadingZeros := 15;
    elsif (x(31 downto 18)="00000000000000000000000000000000") then
        leadingZeros := 14;
    elsif (x(31 downto 19)="00000000000000000000000000000000") then
        leadingZeros := 13;
    elsif (x(31 downto 20)="00000000000000000000000000000000") then
        leadingZeros := 12;
    elsif (x(31 downto 21)="00000000000000000000000000000000") then
        leadingZeros := 11;
```

```
        elsif (x(31 downto 22)="0000000000") then
            leadingZeros := 10;
        elsif (x(31 downto 23)="000000000") then
            leadingZeros := 9;
        elsif (x(31 downto 24)="00000000") then
            leadingZeros := 8;
        elsif (x(31 downto 25)="0000000") then
            leadingZeros := 7;
        elsif (x(31 downto 26)="000000") then
            leadingZeros := 6;
        elsif (x(31 downto 27)="00000") then
            leadingZeros := 5;
        elsif (x(31 downto 28)="0000") then
            leadingZeros := 4;
        elsif (x(31 downto 29)="000") then
            leadingZeros := 3;
        elsif (x(31 downto 30)="00") then
            leadingZeros := 2;
        elsif (x(31)='0') then
            leadingZeros := 1;
        else
            leadingZeros := 0;
        end if;
      return leadingZeros;
    end leading32;


    --trailing64
    function trailing64 (x : std_logic_vector(63 downto 0)) return zeros64 is
    variable trailingZeros : zeros64;
    begin
        if    (x(63 downto 0)=x"0000000000000000") then
            trailingZeros:=64;
        elsif (x(62 downto
0)="00000000000000000000000000000000000000000000000000000000000000") then
            trailingZeros:=63;
        elsif (x(61 downto
0)="000000000000000000000000000000000000000000000000000000000000000") then
            trailingZeros:=62;
        elsif (x(60 downto
0)="0000000000000000000000000000000000000000000000000000000000000000") then
            trailingZeros:=61;
        elsif (x(59 downto
0)="0000000000000000000000000000000000000000000000000000000000000000") then
            trailingZeros:=60;
        elsif (x(58 downto
0)="0000000000000000000000000000000000000000000000000000000000000000") then
            trailingZeros:=59;
        elsif (x(57 downto
0)="0000000000000000000000000000000000000000000000000000000000000000") then
            trailingZeros:=58;
        elsif (x(56 downto
0)="0000000000000000000000000000000000000000000000000000000000000000") then
            trailingZeros:=57;
        elsif (x(55 downto
0)="0000000000000000000000000000000000000000000000000000000000000000") then
            trailingZeros:=56;
```

```
        elsif (x(54 downto
0)="00000000000000000000000000000000000000000000000000000000") then
            trailingZeros:=55;
        elsif (x(53 downto
0)="00000000000000000000000000000000000000000000000000000000") then
            trailingZeros:=54;
        elsif (x(52 downto
0)="00000000000000000000000000000000000000000000000000000000") then
            trailingZeros:=53;
        elsif (x(51 downto
0)="00000000000000000000000000000000000000000000000000000000") then
            trailingZeros:=52;
        elsif (x(50 downto
0)="00000000000000000000000000000000000000000000000000000000") then
            trailingZeros:=51;
        elsif (x(49 downto
0)="00000000000000000000000000000000000000000000000000000000") then
            trailingZeros:=50;
        elsif (x(48 downto
0)="00000000000000000000000000000000000000000000000000000000") then
            trailingZeros:=49;
        elsif (x(47 downto
0)="00000000000000000000000000000000000000000000000000000000") then
            trailingZeros:=48;
        elsif (x(46 downto
0)="00000000000000000000000000000000000000000000000000000000") then
            trailingZeros:=47;
        elsif (x(45 downto
0)="00000000000000000000000000000000000000000000000000000000") then
            trailingZeros:=46;
        elsif (x(44 downto
0)="00000000000000000000000000000000000000000000000000000000") then
            trailingZeros:=45;
        elsif (x(43 downto 0)="00000000000000000000000000000000000000000000000000000000")
then
            trailingZeros:=44;
        elsif (x(42 downto 0)="00000000000000000000000000000000000000000000000000000000")
then
            trailingZeros:=43;
        elsif (x(41 downto 0)="00000000000000000000000000000000000000000000000000000000")
then
            trailingZeros:=42;
        elsif (x(40 downto 0)="00000000000000000000000000000000000000000000000000000000")
then
            trailingZeros:=41;
        elsif (x(39 downto 0)="00000000000000000000000000000000000000000000000000000000")
then
            trailingZeros:=40;
        elsif (x(38 downto 0)="00000000000000000000000000000000000000000000000000000000") then
            trailingZeros:=39;
        elsif (x(37 downto 0)="00000000000000000000000000000000000000000000000000000000") then
            trailingZeros:=38;
        elsif (x(36 downto 0)="00000000000000000000000000000000000000000000000000000000") then
            trailingZeros:=37;
        elsif (x(35 downto 0)="00000000000000000000000000000000000000000000000000000000") then
            trailingZeros:=36;
        elsif (x(34 downto 0)="00000000000000000000000000000000000000000000000000000000") then
```

```
    trailingZeros:=35;
elsif (x(33 downto 0)="0000000000000000000000000000000000") then
    trailingZeros:=34;
elsif (x(32 downto 0)="000000000000000000000000000000000") then
    trailingZeros:=33;
elsif(x(31 downto 0)=x"00000000") then
    trailingZeros:=32;
elsif (x(30 downto 0)="00000000000000000000000000000000") then
    trailingZeros:=31;
elsif (x(29 downto 0)="0000000000000000000000000000000") then
    trailingZeros:=30;
elsif (x(28 downto 0)="000000000000000000000000000000") then
    trailingZeros:=29;
elsif (x(27 downto 0)="00000000000000000000000000000") then
    trailingZeros:=28;
elsif (x(26 downto 0)="0000000000000000000000000000") then
    trailingZeros:=27;
elsif (x(25 downto 0)="000000000000000000000000000") then
    trailingZeros:=26;
elsif (x(24 downto 0)="00000000000000000000000000") then
    trailingZeros:=25;
elsif (x(23 downto 0)="0000000000000000000000000") then
    trailingZeros:=24;
elsif (x(22 downto 0)="000000000000000000000000") then
    trailingZeros:=23;
elsif (x(21 downto 0)="00000000000000000000000") then
    trailingZeros:=22;
elsif (x(20 downto 0)="0000000000000000000000") then
    trailingZeros:=21;
elsif (x(19 downto 0)="000000000000000000000") then
    trailingZeros:=20;
elsif (x(18 downto 0)="00000000000000000000") then
    trailingZeros:=19;
elsif (x(17 downto 0)="0000000000000000000") then
    trailingZeros:=18;
elsif (x(16 downto 0)="000000000000000000") then
    trailingZeros:=17;
elsif (x(15 downto 0)="00000000000000000") then
    trailingZeros:=16;
elsif (x(14 downto 0)="0000000000000000") then
    trailingZeros:=15;
elsif (x(13 downto 0)="000000000000000") then
    trailingZeros:=14;
elsif (x(12 downto 0)="00000000000000") then
    trailingZeros:=13;
elsif (x(11 downto 0)="0000000000000") then
    trailingZeros:=12;
elsif (x(10 downto 0)="000000000000") then
    trailingZeros:=11;
elsif (x(9 downto 0)="00000000000") then
    trailingZeros:=10;
elsif (x(8 downto 0)="0000000000") then
    trailingZeros:=9;
elsif (x(7 downto 0)="000000000") then
    trailingZeros:=8;
elsif (x(6 downto 0)="00000000") then
    trailingZeros:=7;
```

```
        elsif (x(5 downto 0)="000000") then
            trailingZeros:=6;
        elsif (x(4 downto 0)="00000") then
            trailingZeros:=5;
        elsif (x(3 downto 0)="0000") then
            trailingZeros:=4;
        elsif (x(2 downto 0)="000") then
            trailingZeros:=3;
        elsif (x(1 downto 0)="00") then
            trailingZeros:=2;
        elsif (x(0)='0') then
            trailingZeros:=1;
        else
            trailingZeros:=0;
        end if;
        return trailingZeros;
    end trailing64;

    --trailing64
    function leading64 (x : std_logic_vector(63 downto 0)) return zeros64 is
    variable leadingZeros : zeros64;
    begin
        if    (x(63 downto 0)=x"0000000000000000") then
            leadingZeros := 64;
        elsif (x(63 downto
1)="00000000000000000000000000000000000000000000000000000000000000000") then
            leadingZeros := 63;
        elsif (x(63 downto
2)="00000000000000000000000000000000000000000000000000000000000000000") then
            leadingZeros := 62;
        elsif (x(63 downto
3)="00000000000000000000000000000000000000000000000000000000000000000") then
            leadingZeros := 61;
        elsif (x(63 downto
4)="00000000000000000000000000000000000000000000000000000000000000000") then
            leadingZeros := 60;
        elsif (x(63 downto
5)="00000000000000000000000000000000000000000000000000000000000000000") then
            leadingZeros := 59;
        elsif (x(63 downto
6)="00000000000000000000000000000000000000000000000000000000000000000") then
            leadingZeros := 58;
        elsif (x(63 downto
7)="00000000000000000000000000000000000000000000000000000000000000000") then
            leadingZeros := 57;
        elsif (x(63 downto
8)="00000000000000000000000000000000000000000000000000000000000000000") then
            leadingZeros := 56;
        elsif (x(63 downto
9)="00000000000000000000000000000000000000000000000000000000000000000") then
            leadingZeros := 55;
        elsif (x(63 downto
10)="00000000000000000000000000000000000000000000000000000000000000000") then
            leadingZeros := 54;
        elsif (x(63 downto
11)="00000000000000000000000000000000000000000000000000000000000000000") then
            leadingZeros := 53;
```

```
        elsif (x(63 downto
12)="0000000000000000000000000000000000000000000000000000") then
            leadingZeros := 52;
        elsif (x(63 downto
13)="0000000000000000000000000000000000000000000000000000") then
            leadingZeros := 51;
        elsif (x(63 downto
14)="0000000000000000000000000000000000000000000000000000") then
            leadingZeros := 50;
        elsif (x(63 downto
15)="0000000000000000000000000000000000000000000000000000") then
            leadingZeros := 49;
        elsif (x(63 downto
16)="0000000000000000000000000000000000000000000000000000") then
            leadingZeros := 48;
        elsif (x(63 downto
17)="0000000000000000000000000000000000000000000000000000") then
            leadingZeros := 47;
        elsif (x(63 downto
18)="0000000000000000000000000000000000000000000000000000") then
            leadingZeros := 46;
        elsif (x(63 downto
19)="0000000000000000000000000000000000000000000000000000") then
            leadingZeros := 45;
        elsif (x(63 downto
20)="0000000000000000000000000000000000000000000000000000") then
            leadingZeros := 44;
        elsif (x(63 downto 21)="0000000000000000000000000000000000000000000000000000")
then
            leadingZeros := 43;
        elsif (x(63 downto 22)="0000000000000000000000000000000000000000000000000000")
then
            leadingZeros := 42;
        elsif (x(63 downto 23)="0000000000000000000000000000000000000000000000000000")
then
            leadingZeros := 41;
        elsif (x(63 downto 24)="0000000000000000000000000000000000000000000000000000")
then
            leadingZeros := 40;
        elsif (x(63 downto 25)="0000000000000000000000000000000000000000000000000000")
then
            leadingZeros := 39;
        elsif (x(63 downto 26)="0000000000000000000000000000000000000000000000000000") then
            leadingZeros := 38;
        elsif (x(63 downto 27)="0000000000000000000000000000000000000000000000000000") then
            leadingZeros := 37;
        elsif (x(63 downto 28)="0000000000000000000000000000000000000000000000000000") then
            leadingZeros := 36;
        elsif (x(63 downto 29)="0000000000000000000000000000000000000000000000000000") then
            leadingZeros := 35;
        elsif (x(63 downto 30)="0000000000000000000000000000000000000000000000000000") then
            leadingZeros := 34;
        elsif (x(63 downto 31)="0000000000000000000000000000000000000000000000000000") then
            leadingZeros := 33;
        elsif (x(63 downto 32)=x"00000000") then
            leadingZeros := 32;
        elsif (x(63 downto 33)="0000000000000000000000000000000000000000000000000000") then
```

```
    leadingZeros := 31;
elsif (x(63 downto 34)="00000000000000000000000000000000") then
    leadingZeros := 30;
elsif (x(63 downto 35)="00000000000000000000000000000000") then
    leadingZeros := 29;
elsif (x(63 downto 36)="00000000000000000000000000000000") then
    leadingZeros := 28;
elsif (x(63 downto 37)="00000000000000000000000000000000") then
    leadingZeros := 27;
elsif (x(63 downto 38)="00000000000000000000000000000000") then
    leadingZeros := 26;
elsif (x(63 downto 39)="00000000000000000000000000000000") then
    leadingZeros := 25;
elsif (x(63 downto 40)="00000000000000000000000000000000") then
    leadingZeros := 24;
elsif (x(63 downto 41)="00000000000000000000000000000000") then
    leadingZeros := 23;
elsif (x(63 downto 42)="00000000000000000000000000000000") then
    leadingZeros := 22;
elsif (x(63 downto 43)="00000000000000000000000000000000") then
    leadingZeros := 21;
elsif (x(63 downto 44)="00000000000000000000000000000000") then
    leadingZeros := 20;
elsif (x(63 downto 45)="00000000000000000000000000000000") then
    leadingZeros := 19;
elsif (x(63 downto 46)="00000000000000000000000000000000") then
    leadingZeros := 18;
elsif (x(63 downto 47)="00000000000000000000000000000000") then
    leadingZeros := 17;
elsif (x(63 downto 48)="00000000000000000000000000000000") then
    leadingZeros := 16;
elsif (x(63 downto 49)="00000000000000000000000000000000") then
    leadingZeros := 15;
elsif (x(63 downto 50)="00000000000000000000000000000000") then
    leadingZeros := 14;
elsif (x(63 downto 51)="00000000000000000000000000000000") then
    leadingZeros := 13;
elsif (x(63 downto 52)="00000000000000000000000000000000") then
    leadingZeros := 12;
elsif (x(63 downto 53)="00000000000000000000000000000000") then
    leadingZeros := 11;
elsif (x(63 downto 54)="00000000000000000000000000000000") then
    leadingZeros := 10;
elsif (x(63 downto 55)="00000000000000000000000000000000") then
    leadingZeros := 9;
elsif (x(63 downto 56)="00000000000000000000000000000000") then
    leadingZeros := 8;
elsif (x(63 downto 57)="00000000000000000000000000000000") then
    leadingZeros := 7;
elsif (x(63 downto 58)="00000000000000000000000000000000") then
    leadingZeros := 6;
elsif (x(63 downto 59)="00000000000000000000000000000000") then
    leadingZeros := 5;
elsif (x(63 downto 60)="00000000000000000000000000000000") then
    leadingZeros := 4;
elsif (x(63 downto 61)="00000000000000000000000000000000") then
    leadingZeros := 3;
```

```
        elsif (x(63 downto 62)="00") then
            leadingZeros := 2;
        elsif (x(63)='0') then
            leadingZeros := 1;
        else
            leadingZeros := 0;
        end if;
        return leadingZeros;
    end leading64;

end fixed_point_pkg;
```

Appendix C

Accelerator Interface

```
`include "common_defs.v"

module accelerator1
  #(
    parameter WB_DWIDTH  = 32,
    parameter WB_SWIDTH  = 4
    )
   (
    input                    i_clk,
    input                    i_rst,

    input [31:0]             i_wb_adr,
    input [WB_SWIDTH-1:0]    i_wb_sel,
    input                    i_wb_we,
    output [WB_DWIDTH-1:0]   o_wb_dat,
    input [WB_DWIDTH-1:0]    i_wb_dat,
    input                    i_wb_cyc,
    input                    i_wb_stb,
    output                   o_wb_ack,
    output                   o_wb_err
    );

    //Registers
    reg [31:0] i_opt_reg = 'd0;
    reg [31:0] acc_in_0_reg = 'd0;
    reg [31:0] acc_in_1_reg = 'd0;
    reg [31:0] acc_in_2_reg = 'd0;
    reg [31:0] acc_in_3_reg = 'd0;
    reg [31:0] acc_in_4_reg = 'd0;
    reg [31:0] acc_in_5_reg = 'd0;
    reg [31:0] acc_in_6_reg = 'd0;
    reg [31:0] acc_in_7_reg = 'd0;
    reg [31:0] acc_in_8_reg = 'd0;
    reg [31:0] acc_in_9_reg = 'd0;
    reg [31:0] acc_in_10_reg = 'd0;
    reg [31:0] acc_in_11_reg = 'd0;
    reg [31:0] acc_in_12_reg = 'd0;
    reg [31:0] acc_in_13_reg = 'd0;
    reg [31:0] acc_in_14_reg = 'd0;
    reg [31:0] acc_in_15_reg = 'd0;
    reg [31:0] acc_in_16_reg = 'd0;
    reg [31:0] acc_out_0_reg = 'd0;
    reg [31:0] status_reg    = 32'h00000000;
   // Wishbone interface
   reg [31:0]                                    wb_rdata32 = 'd0;
   wire                                          wb_start_write;
   wire                                          wb_start_read;
   reg                                           wb_start_read_d1 = 'd0;
```

```
   wire [31:0]                                  wb_wdata32;

   reg acc_start = 'd0;
   wire out_rdy;
   wire [31:0]                                  acc1_out_0;
   wire [31:0]                                  acc1_out_1;


   // ======================================
   // Instantiate accelerator core
   // ======================================
    epsilon2Hybrid u_epsilon2 (
        .clk                    ( i_clk             ),
        .rst                    ( i_rst             ),
        .i_start                ( acc_start         ),
        .o_rdy                  ( out_rdy           ),

        .diffpoint0             ( acc_in_0_reg      ),
        .diffpoint1             ( acc_in_1_reg      ),
        .diffpoint2             ( acc_in_2_reg      ),
        .diffpoint3             ( acc_in_3_reg      ),
        .diffpoint4             ( acc_in_4_reg      ),
        .diffpoint5             ( acc_in_5_reg      ),
        .diffpoint6             ( acc_in_6_reg      ),
        .xV20                   ( acc_in_7_reg      ),
        .xV21                   ( acc_in_8_reg      ),
        .xV22                   ( acc_in_9_reg      ),
        .xV23                   ( acc_in_10_reg     ),
        .xV24                   ( acc_in_11_reg     ),
        .xV25                   ( acc_in_12_reg     ),
        .xV26                   ( acc_in_13_reg     ),
        .mulpoint               ( acc_in_14_reg     ),
        .d1                     ( acc_in_15_reg     ),
        .df                     ( acc_in_16_reg     ),

        .fraction               ( acc1_out_0        )
    );


   //=====================================================================
   //Assignments
   // Can't start a write while a read is completing. The ack for the read
cycle
   // needs to be sent first
   assign wb_start_write = i_wb_stb && i_wb_we && !wb_start_read_d1;
   assign wb_start_read  = i_wb_stb && !i_wb_we && !o_wb_ack;

   always @( posedge i_clk or posedge i_rst) begin
      if(i_rst)
        wb_start_read_d1 <= 1'b0;
      else
        wb_start_read_d1 <= wb_start_read;
   end

   assign o_wb_err = 1'd0;
   assign o_wb_ack = i_wb_stb && ( wb_start_write || wb_start_read_d1 );
```

```
generate
   if (WB_DWIDTH == 128)
     begin : wb128
        assign wb_wdata32   = i_wb_adr[3:2] == 2'd3 ? i_wb_dat[127:96] :
                              i_wb_adr[3:2] == 2'd2 ? i_wb_dat[ 95:64] :
                              i_wb_adr[3:2] == 2'd1 ? i_wb_dat[ 63:32] :
                              i_wb_dat[ 31: 0] ;

        assign o_wb_dat     = {4{wb_rdata32}};
     end
   else
     begin : wb32
        assign wb_wdata32  = i_wb_dat;
        assign o_wb_dat    = wb_rdata32;
     end
endgenerate
// ========================================================
// Register Writes
// ========================================================
 always @(posedge acc_start or posedge out_rdy)
 begin
     if(out_rdy)
     begin
         acc_out_0_reg <= acc1_out_0;
         status_reg    <= 32'h11111111;
     end

     if(acc_start)
         status_reg    <= 32'h00000000;
 end

always @( posedge i_clk or posedge i_rst)
begin
     if(i_rst)
         begin
         i_opt_reg       <= 32'h00000000;
         acc_in_0_reg    <= 32'h00000000;
         acc_in_1_reg    <= 32'h00000000;
         acc_in_2_reg    <= 32'h00000000;
         acc_in_3_reg    <= 32'h00000000;
         acc_in_4_reg    <= 32'h00000000;
         acc_in_5_reg    <= 32'h00000000;
         acc_in_6_reg    <= 32'h00000000;
         acc_in_7_reg    <= 32'h00000000;
         acc_in_8_reg    <= 32'h00000000;
         acc_in_9_reg    <= 32'h00000000;
         acc_in_10_reg   <= 32'h00000000;
         acc_in_11_reg   <= 32'h00000000;
         acc_in_12_reg   <= 32'h00000000;
         acc_in_13_reg   <= 32'h00000000;
         acc_in_14_reg   <= 32'h00000000;
         acc_in_15_reg   <= 32'h00000000;
         acc_in_16_reg   <= 32'h00000000;
         acc_start       <= 1'b0;
         end
     else
     begin
```

```
        if(acc_start)
            begin
            acc_start       <= 1'b0;
            i_opt_reg       <= 32'h00000000;
            end
        if ( wb_start_write )
        begin
            case (i_wb_adr[11:0])
                `ACC1_OPTIONS: begin
                                    i_opt_reg       <= i_wb_dat[ 31: 0];
                                    acc_start       <= 1'b1;
                 end
                `ACC1_INPUT0:       acc_in_0_reg    <= i_wb_dat[ 31: 0];
                `ACC1_INPUT1:       acc_in_1_reg    <= i_wb_dat[ 31: 0];
                `ACC1_INPUT2:       acc_in_2_reg    <= i_wb_dat[ 31: 0];
                `ACC1_INPUT3:       acc_in_3_reg    <= i_wb_dat[ 31: 0];
                `ACC1_INPUT4:       acc_in_4_reg    <= i_wb_dat[ 31: 0];
                `ACC1_INPUT5:       acc_in_5_reg    <= i_wb_dat[ 31: 0];
                `ACC1_INPUT6:       acc_in_6_reg    <= i_wb_dat[ 31: 0];
                `ACC1_INPUT7:       acc_in_7_reg    <= i_wb_dat[ 31: 0];
                `ACC1_INPUT8:       acc_in_8_reg    <= i_wb_dat[ 31: 0];
                `ACC1_INPUT9:       acc_in_9_reg    <= i_wb_dat[ 31: 0];
                `ACC1_INPUT10:      acc_in_10_reg   <= i_wb_dat[ 31: 0];
                `ACC1_INPUT11:      acc_in_11_reg   <= i_wb_dat[ 31: 0];
                `ACC1_INPUT12:      acc_in_12_reg   <= i_wb_dat[ 31: 0];
                `ACC1_INPUT13:      acc_in_13_reg   <= i_wb_dat[ 31: 0];
                `ACC1_INPUT14:      acc_in_14_reg   <= i_wb_dat[ 31: 0];
                `ACC1_INPUT15:      acc_in_15_reg   <= i_wb_dat[ 31: 0];
                `ACC1_INPUT16:      acc_in_16_reg   <= i_wb_dat[ 31: 0];
            endcase
        end
    end
 end
// ============================================================
// Register Reads
// ============================================================
always @( posedge i_clk or posedge i_rst )
begin
   if(i_rst)
     begin
     wb_rdata32 <= 32'h10101010;
     end
   else
     begin
     if ( wb_start_read )
         case ( i_wb_adr[11:0] )
             `ACC1_OPTIONS: wb_rdata32 <= i_opt_reg[ 31: 0];
             `ACC1_OUTPUT0: wb_rdata32 <= acc_out_0_reg[ 31: 0];
             `ACC1_STATUS:  wb_rdata32 <= status_reg[ 31: 0];

             default: wb_rdata32 <= 32'h33333333;
         endcase
     end
end
endmodule
```

Appendix D

Synthesis Reports

D.1 Design 1 Map Report File

Tiles: APB, Turbo-Amber and Main Memory (SHMAC “VTZ” layout).

```
Release 14.5 Map P.58f (lin64)
Xilinx Mapping Report File for Design 'AXILTEx'

Design Information
------------------
Command Line   : map -mt 4 -p XC5VLX330-ff1760-1 -timing -ol high
-register duplication -t 1 -cm speed -pr b -c 100 -tx on -o shmac map.ncd
-intstyle xflow -w -detail shmac.ngd shmac.pcf
Target Device  : xc5vlx330
Target Package : ff1760
Target Speed   : -1
Mapper Version : virtex5 -- $Revision: 1.55 $
Mapped Date    : Tue Jan 27 19:35:41 2015

Design Summary
--------------
Number of errors:      0
Number of warnings:   21
Slice Logic Utilization:
  Number of Slice Registers:                17,933 out of 207,360    8%
    Number used as Flip Flops:              17,932
    Number used as Latch-thrus:                  1
  Number of Slice LUTs:                     28,806 out of 207,360   13%
    Number used as logic:                   28,774 out of 207,360   13%
      Number using O6 output only:          27,741
      Number using O5 output only:             251
      Number using O5 and O6:                  782
    Number used as Memory:                      20 out of  54,720    1%
      Number used as Dual Port RAM:             20
        Number using O5 and O6:                 20
    Number used as exclusive route-thru:        12
  Number of route-thrus:                       257
    Number using O6 output only:               256
    Number using O5 and O6:                      1

Slice Logic Distribution:
  Number of occupied Slices:                 9,720 out of  51,840   18%
  Number of LUT Flip Flop pairs used:       30,774
    Number with an unused Flip Flop:        12,841 out of  30,774   41%
    Number with an unused LUT:               1,968 out of  30,774    6%
    Number of fully used LUT-FF pairs:      15,965 out of  30,774   51%
    Number of unique control sets:             389
```

```
    Number of slice register sites lost
      to control set restrictions:             508 out of 207,360    1%

  A LUT Flip Flop pair for this architecture represents one LUT paired with
  one Flip Flop within a slice.  A control set is a unique combination of
  clock, reset, set, and enable signals for a registered element.
  The Slice Logic Distribution report is not meaningful if the design is
  over-mapped for a non-slice resource or if Placement fails.
  OVERMAPPING of BRAM resources should be ignored if the design is
  over-mapped for a non-BRAM resource or if placement fails.

IO Utilization:
  Number of bonded IOBs:                     1,110 out of   1,200   92%
    Number of LOCed IOBs:                    1,110 out of   1,110  100%
    IOB Flip Flops:                            401

Specific Feature Utilization:
  Number of BlockRAM/FIFO:                      26 out of     288    9%
    Number using BlockRAM only:                 26
    Total primitives used:
      Number of 36k BlockRAM used:              22
      Number of 18k BlockRAM used:               5
    Total Memory used (KB):                   882 out of  10,368    8%
  Number of BUFG/BUFGCTRLs:                     3 out of      32    9%
    Number used as BUFGs:                       3
  Number of DCM_ADVs:                           1 out of      12    8%

Average Fanout of Non-Clock Nets:               4.94
Release 14.5 Map P.58f (lin64)
Xilinx Mapping Report File for Design 'AXILTEx'

Design Information
------------------
Command Line   : map -mt 4 -p XC5VLX330-ff1760-1 -timing -ol high
-register_duplication -t 1 -cm speed -pr b -c 100 -tx on -o shmac_map.ncd
-intstyle xflow -w -detail shmac.ngd shmac.pcf
Target Device  : xc5vlx330
Target Package : ff1760
Target Speed   : -1
Mapper Version : virtex5 -- $Revision: 1.55 $
Mapped Date    : Tue Jan 27 19:35:41 2015

Design Summary
--------------
Number of errors:      0
Number of warnings:   21
Slice Logic Utilization:
  Number of Slice Registers:                17,933 out of 207,360    8%
    Number used as Flip Flops:              17,932
    Number used as Latch-thrus:                  1
  Number of Slice LUTs:                     28,806 out of 207,360   13%
    Number used as logic:                   28,774 out of 207,360   13%
      Number using O6 output only:          27,741
      Number using O5 output only:             251
      Number using O5 and O6:                  782
    Number used as Memory:                      20 out of  54,720    1%
      Number used as Dual Port RAM:             20
```

```
        Number using O5 and O6:                20
    Number used as exclusive route-thru:        12
  Number of route-thrus:                       257
    Number using O6 output only:               256
    Number using O5 and O6:                      1

Slice Logic Distribution:
  Number of occupied Slices:                 9,720 out of  51,840   18%
  Number of LUT Flip Flop pairs used:       30,774
    Number with an unused Flip Flop:        12,841 out of  30,774   41%
    Number with an unused LUT:               1,968 out of  30,774    6%
    Number of fully used LUT-FF pairs:      15,965 out of  30,774   51%
    Number of unique control sets:             389
    Number of slice register sites lost
      to control set restrictions:             508 out of 207,360    1%

  A LUT Flip Flop pair for this architecture represents one LUT paired with
  one Flip Flop within a slice.  A control set is a unique combination of
  clock, reset, set, and enable signals for a registered element.
  The Slice Logic Distribution report is not meaningful if the design is
  over-mapped for a non-slice resource or if Placement fails.
  OVERMAPPING of BRAM resources should be ignored if the design is
  over-mapped for a non-BRAM resource or if placement fails.

IO Utilization:
  Number of bonded IOBs:                     1,110 out of   1,200   92%
    Number of LOCed IOBs:                    1,110 out of   1,110  100%
    IOB Flip Flops:                            401

Specific Feature Utilization:
  Number of BlockRAM/FIFO:                      26 out of     288    9%
    Number using BlockRAM only:                 26
    Total primitives used:
      Number of 36k BlockRAM used:              22
      Number of 18k BlockRAM used:               5
    Total Memory used (KB):                   882 out of  10,368    8%
  Number of BUFG/BUFGCTRLs:                     3 out of      32    9%
    Number used as BUFGs:                       3
  Number of DCM_ADVs:                           1 out of      12    8%

Average Fanout of Non-Clock Nets:               4.94
```

D.2 Design 2 Map Report File

Tiles: APB, Turbo-Amber with additional accelerator and Main Memory.

```
Release 14.5 Map P.58f (lin64)
Xilinx Mapping Report File for Design 'AXILTEx'
```

```
Design Information
------------------
Command Line   : map -mt 4 -p XC5VLX330-ff1760-1 -timing -ol high
-register_duplication -t 1 -cm speed -pr b -c 100 -tx on -o shmac_map.ncd
-intstyle xflow -w -detail shmac.ngd shmac.pcf
Target Device  : xc5vlx330
Target Package : ff1760
Target Speed   : -1
Mapper Version : virtex5 -- $Revision: 1.55 $
Mapped Date    : Tue May 19 09:58:21 2015

Design Summary
--------------
Number of errors:      0
Number of warnings:   21
Slice Logic Utilization:
  Number of Slice Registers:                21,025 out of 207,360   10%
    Number used as Flip Flops:              20,989
    Number used as Latches:                     34
    Number used as Latch-thrus:                  2
  Number of Slice LUTs:                     30,790 out of 207,360   14%
    Number used as logic:                   30,717 out of 207,360   14%
      Number using O6 output only:          29,535
      Number using O5 output only:             254
      Number using O5 and O6:                  928
    Number used as Memory:                      59 out of  54,720    1%
      Number used as Dual Port RAM:             20
        Number using O5 and O6:                 20
      Number used as Shift Register:            39
        Number using O6 output only:            39
    Number used as exclusive route-thru:        14
  Number of route-thrus:                       309
    Number using O6 output only:               259
    Number using O5 output only:                49
    Number using O5 and O6:                      1

Slice Logic Distribution:
  Number of occupied Slices:                10,796 out of  51,840   20%
  Number of LUT Flip Flop pairs used:       34,205
    Number with an unused Flip Flop:        13,180 out of  34,205   38%
    Number with an unused LUT:               3,415 out of  34,205    9%
    Number of fully used LUT-FF pairs:      17,610 out of  34,205   51%
    Number of unique control sets:             427
    Number of slice register sites lost
      to control set restrictions:             586 out of 207,360    1%

  A LUT Flip Flop pair for this architecture represents one LUT paired with
  one Flip Flop within a slice.  A control set is a unique combination of
  clock, reset, set, and enable signals for a registered element.
  The Slice Logic Distribution report is not meaningful if the design is
  over-mapped for a non-slice resource or if Placement fails.
  OVERMAPPING of BRAM resources should be ignored if the design is
  over-mapped for a non-BRAM resource or if placement fails.

IO Utilization:
  Number of bonded IOBs:                     1,110 out of   1,200   92%
    Number of LOCed IOBs:                    1,110 out of   1,110  100%
```

```
    IOB Flip Flops:                          401

Specific Feature Utilization:
  Number of BlockRAM/FIFO:                    25 out of     288    8%
    Number using BlockRAM only:               25
    Total primitives used:
      Number of 36k BlockRAM used:            22
      Number of 18k BlockRAM used:             5
    Total Memory used (KB):                  882 out of  10,368    8%
  Number of BUFG/BUFGCTRLs:                    4 out of      32   12%
    Number used as BUFGs:                      4
  Number of DCM_ADVs:                          1 out of      12    8%
  Number of DSP48Es:                          24 out of     192   12%

Average Fanout of Non-Clock Nets:              4.65
```

Bibliography

[1] L. D. Iasemidis, D. S. Shiau, P. M. Pardalos, W. A. Chaovalitwongse, K. Narayanan, A. Prasad, K. Tsakalis, P. R. Carney, and J. C. Sackellares. *Long-Term Prospective On-Line Real-Time Seizure Prediction*. Clinical Neurophysiology, 116(3), 532-544, 2005.

[2] L. D. Iasemidis, D.S. Shiau, W. A. Chaovalitwongse, J. C. Sackellares, P. M. Pardalos, J. C. Principe, P. R. Carney, A. Prasad, B. Veeramani, and K. Tsakalis. *Adaptive Epileptic Seizure Prediction System*. IEEE Transactions on Biomedical Engineering, 50(5), 616-627, 2003.

[3] EECS. *Single-ISA Heterogeneous MAny-core Computer project plan*, NTNU, September 2014.

[4] J.G. Koomey, S. Berard, M. Sanchez, and H. Wong. *Implications of historical trends in the electrical efficiency of computing*, Annals of the History of Computing, IEEE, 33(3), 46–54, 2011.

[5] D.A. Patterson, and J.L. Hennessy. *Computer organization and design: the hardware/software interface*, 4th Edition, Morgan Kaufmann, 2011.

[6] H. Esmaeilzadeh, E. Blem, R. St Amant, K. Sankaralingam,and D. Burger. *Dark Silicon and the End of Multicore Scaling*, Proc. ISCA 38th Annual International Symposium on Computer Architecture, San Jose, USA, pp. 365-376, 2011.

[7] E.S. Chung, P.A. Milder, J.C. Hoe, and K. Mai. *Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs?*, Proc. MICRO 43rd Annual IEEE/ACM International Symposium on Microarchitecture, Atlanta, USA, pp. 225–236, 2010.

[8] Energy Efficient Computing Systems Initiative, http://www.ntnu.edu/ime/eecs

[9] A. Wolf, J. B. Swift, H. L. Swinney, and J. A. Vastano. *Determining Lyapunov Exponents From a Time Series*. Physica D: Nonlinear Phenomena, 16(3), 285-317, 1985.

[10] L. D. Iasemidis. *Seizure Prediction and its Applications*. Neurosurgery Clinics of North America, pp. 489-506, October 2011.

[11] Open cores. *Amber 2 Core Specification*. April 2013.

[12] A. D. Booth. *A signed binary multiplication technique*. The Quarterly Journal of Mechanics and Applied Mathematics, 4(2), 236-240, 1951.

[13] A. N. Sloss, D. Symes and C. Wright. *ARM Systems Developers's Guide*. Morgan Kaufmann, 2004.

[14] A. T. Akre and S. Bøe. *Turbo Amber: A high-performance processor core for SHMAC*, June 2014.

[15] Silicon Labs. *Digital Signal Processing with the EFM32AN0051 - Application Note*, September 2013.

[16] ARM Holdings. *Fixed Point Arithmetic on the ARM. Application Note 33*, September 1996.

[17] E. L. Oberstar. Fixed-point representation & fractional math. (White paper), August 2007.

[18] A.G.M. Cilio and H. Corporaal. *Floating point to fixed point conversion of ccode*, 1999.

[19] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*, 4th Edition, Morgan Kaufmann, 2006.

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

[21] G. De Michelli and R.K. Gupta. *Hardware/software co-design*. Proceedings of the IEEE, 85(3), 349-365, 1997.

[22] P. Schaumont. *A practical introduction to hardware/software codesign*. Springer Science & Business Media, 2012.
[23] J. Teich. *Hardware/software codesign: The past, the present, and predicting the future*. Proceedings of the IEEE, 100(Special Centennial Issue), 1411-1430, 2012
[24] M. Wolf. *Computers as components: principles of embedded computing system design*. Elsevier, 2012.
[25] ST Microelectronics. *RM0090 Reference manual*, October 2014.
[26] I. Lousis. *Implementation of Biomedical Algorithm on the SHMAC platform*, December 2015.
[27] S.N. Berg. *Implementation of the Epileptic Seizure Prediction Algorithm on the SHMAC Platform*, June 2014.
[28] ARM Holdings. *ARM Software Development Toolkit Version 2.50 Reference Guide*, October 2009.
[29] J. D. Knutsen. *Extending Amber with a Hardware FPU*, June 2014.
[30] ARM Holdings. *Floating-Point Performance. Application Note 55*, January 1998.
[31] H. O. Wikene. *Benchmarking SHMAC*, October 2014.
[32] A. L. Indegaard. *Configurable Floating-Point Unit for the SHMAC Platform*, June 2014.
[33] M.L. Teilgård. *Integration of hardware accelerators on the SHMAC platform*. June 2014.
[34] Arakawa, Fumio, Hironori Kasahara, Tohru Nojiri, Hideyuki Noda, Yasuhiro Tawara, Akio Idehara, Kenichi Iwata, and Hiroaki Shikano. *Heterogeneous Multicore Processor Technologies for Embedded Systems*. Springer, 2012.
[35] Taylor, Michael B. *Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse*. In Proceedings of the 49th Annual Design Automation Conference, ACM, pp. 1131-1136, 2012.
[36] Xilinx, *Virtex-5 Family Overview*, 5th edition, February 2009.
[37] Xilinx, *LogiCORE IP Floating-Point Operator v.5.0*, March 2011
[38] OpenCores. *Wishbone B4 - Wishbone System-on-chip (SoC) Interconnection Architecture for Portable IP Cores*, 2011.
[39] J. Gaisler. *A structured VHDL design method*, Fault-tolerant microprocessors for space applications, 41-50, 2011.
[40] M. Arora, Mohit. *The art of hardware architecture: Design methods and techniques for digital circuits*, Springer Science & Business Media, 2011.
[41] Xilinx. *LogiCORE IP Divider Generator v.3.0*, March 2011
[42] Xilinx. *Xilinx Power Estimator, User Guide*. April 2014.
[43] S. Rivoire, A.S. Mehul, R. Ranganathan, C. Kozyrakis, and J. Meza. *Models and metrics to enable energy-efficiency optimizations*. IEEE Computer, 40(12), 39-48, 2007.