Eindhoven University of Technology

MASTER

Efficient mapping of EEG algorithms

Heredia Cervantes, A.

Award date:
2019

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain

Department of Electrical Engineering
Electronic Systems Research Group

Efficient Mapping of EEG Algorithms

Master Thesis

Alejandro Heredia Cervantes
Student number: 1037414

Committee Members :
Jos Huisken
Barry de Bruin
Henk Corporaal
Rudolf Mak

Eindhoven, March 2019

Contents

Acronyms

ABU Accumulate-Branch Unit. 10, 29

AGU Address Generation Unit. 22

ALU Arithmetic and Logic Unit. 8, 10, 16, 17, 20–22, 27, 29, 34, 35, 37, 44, 45, 48

ApEn Approximate Entropy. 9

ASIC Application Specific Integrated Circuit. 4

ASIP Application Specific Instruction-Set Processor. 5

BTU Butterfly Unit. 22

cA Approximation Coefficients. 24, 25, 27

cD Detail Coefficients. 24, 25, 27

CGRA Coarse Grain Reconfigurable Array. ii, iii, 1, 3, 5–8, 10, 13, 14, 18, 19, 21, 22, 25, 27, 34–36, 40, 42–45, 47, 48

CORDIC Coordinate Rotation Digital Computer. 4

DFT Discrete Fourier Transform. 13

DIF Decimation In Frequency. 14, 18, 19, 22, 23

DIT Decimation In Time. 14, 18, 19

DSE Design Space Exploration. 6

DSP Digital Signal Processing. 38

DWS Dynamic Warping Similarity. 9

DWT Discrete Wavelet Transform. ii, iii, 2, 4, 9, 13, 24–26, 28, 30, 31, 41, 48

EDA Energy-Delay-Area. 1, 8

EDS Euclidean Distance Similarity. 9

EEG Electroencephalography. ii, 1–5, 8–12, 18, 23, 24, 26, 28, 29, 31, 36, 40, 41, 44, 48

FFT Fast Fourier Transform. ii, 4, 5, 13–15, 17–24, 41, 42, 44, 45, 48

FIR Finite Impulse Response. 4, 31

FPGA Field Programmable Gate Array. 22, 23, 30, 31, 38

FU Functional Units. 1, 5–8, 13, 15, 20, 23, 28–30, 36, 38, 41, 42, 44

FWT Filter-Based Discrete Wavelet Transform. ii, 13, 24, 25, 28, 29, 42, 44

ID Instruction Decoder. 42, 44, 48

IIR Infinite Impulse Response. iii, 4, 13, 31–34, 36–38, 44

IMM Immediate Unit. 10, 27, 29, 44, 47

ISA Instruction Set Architecture. 45, 47, 48

ISS Instruction-Set Simulator. 47

LSU Load-Store Unit. 10, 15–18, 20, 21, 23, 27–30, 36–38, 42, 45, 48

LWT Lifting-Based Discrete Wavelet Transform. ii, 13, 24–31, 40–42, 44, 45, 48, 53

MAC Multiply-Accumulate. 33

MUL Multiplier. 8, 10, 16, 17

PCA Principal Component Analysis. 4

RF Register File. 6, 8, 10, 20, 27, 29, 30, 44, 48

RISC Reduced Instruction Set Computer. 8

ROM Read-Only Memory. 22

SIMD Single Instruction Multiple Data. 10

SOS Second Order Sections. iii, 13, 32–41, 44, 45, 48

SP0 Smooth-Padding of order 0. 31

SVM Support Vector Machine. 4, 9

SWB Switch Boxes. 10

VLIW Very Long Instruction Word. 6, 10, 46, 47

Chapter 1

Introduction

EEG is a monitoring method to record electrical activity of the brain [43]. It is used in a variety of fields and application areas, such as *Brain computer interfaces (BCI)* for game development and wellness[20], and in the medical area as an aid to treat patients with brain-related diseases.

So far, the conventional EEG monitoring/recording devices are cumbersome due to the many connections needed from the electrodes attached to the scalp and a computer used to process the samples, this makes them far from ideal for every day use. Wireless battery-powered EEG monitoring systems that improve the patient EEG experience and make EEG devices least obtrusive are already available on the market [12, 15, 13, 16, 14, 11] but energy efficiency is still a challenge.

In conventional EEG processing platforms the processing and classification are done off-chip using machines that do not have energy constraints. However, off-chip processing would require to send big amounts of raw data and hence it is not suitable for wearable battery-powered EEG system as the energy required for the wireless data transmission is prohibitive. For an 8-channel EEG system, transmitting the raw EEG data by means of a low power radio for off-chip processing consumes around 1.32 mW, reducing the battery life to only a few hours. When feature extraction and classification[1] are done on-chip, the energy consumption is reduced by 13x and 80x respectively [1] and the battery life is prolonged up to a full day.

On-chip processing requires energy efficient processors however even current low-power general purpose processors (CPUs) cannot provide the efficiency required for wireless battery-powered EEG systems. On the other hand, specialized hardware provides high energy efficiency at the cost of flexibility[2]. This is an issue because the optimal EEG processing pipeline is application dependent [23]. This fact and the constant development of new EEG algorithms require efficient EEG systems to include programmable hardware solutions that balance the flexibility-efficiency problem. CGRA architectures can help to achieve efficient and flexible EEG platforms. However, sharing programmable hardware among a set of algorithms requires a detailed analysis for proper sizing of the reconfigurable fabric because it cannot be modified after fabrication.

The contributions of this work are:

- Analysis of the features of an EEG pipeline and how to map them to the *Blocks* CGRA identifying: ideal speed-up, possible optimization opportunities and limitations.
- Evaluation of the efficiency of a variety of EEG features running in a platform composed by a *RISC-V* core + CGRA.
- The proposition of an energy-efficient CGRA architecture that can be used to compute a set of EEG features. The main goal of this flexible architecture will be to minimize the Energy-Delay-Area (EDA) product, having as key metrics the Functional Units (FU) utilization and memory traffic.

[1]Training of the classifier is done off-chip and off-line.

[2]Research on Flexibility is been carried on in the Electronic Systems group at the TU/e, in this dissertation processor flexibility refers to the ability to run/modify different algorithms on the same hardware

The rest of the document is structured as follows. Chapter 2 starts by presenting a generic EEG pipeline, then related work on energy-efficient EEG platforms are revisited. The seizure detection application and the EEG platform used in this thesis are introduced in chapter 4. From this seizure detection application, the *Power per Band*, DWT and the Butterworth filter were selected for implementation and their analysis and the results of the mapping are shown in chapter 6. In chapter 7 a comparison of the mapped features in terms of cycle count, energy consumption and area is presented. Initially, the proposition of an energy model was intended as one of the contributions of this work. However, it was not carried out and chapter 9 explains the reasons. Chapter 8 explains the proposed size for the *Blocks* template and the encountered shortcomings. Finally, the conclusions and future work for this thesis are presented in chapter 10.

Chapter 2

Related Work

EEG monitoring/processing is used in applications that range from brain computer interfaces to seizure detection and classification in medical applications. It is obvious that different applications have different requirements. However, they all follow a common pipeline structure, presented in section 2.1. Energy efficiency and flexibility are the major problems in wearable EEG platforms. Therefore, some approaches taken in the literature that deals with the efficiency-flexibility problem are presented in sections 2.2 to 2.4 . The chapter finalizes discussing CGRA architectures in the context of wearable EEG platforms.

2.1 The generic EEG processing pipeline

The optimal EEG pipeline is application dependent, nonetheless, there are common features used in a generic EEG pipeline which is shown in figure 2.1. The features are organized into 3 stages:

- Signal preprocessing: can include filtering, trimming of the EEG data resampling, signal segmentation and signal selection. The objective is to remove noise and other artifacts such as motion or channel interference, in other words, the data is prepared for further stages.
- Feature extraction: in this stage, features in the frequency domain, time-frequency domain and spatio-temporal domain are extracted using various signal processing algorithms.
- Classification: In this stage, the data is classified using machine learning algorithms that take as input the features extracted from the feature extraction stage.

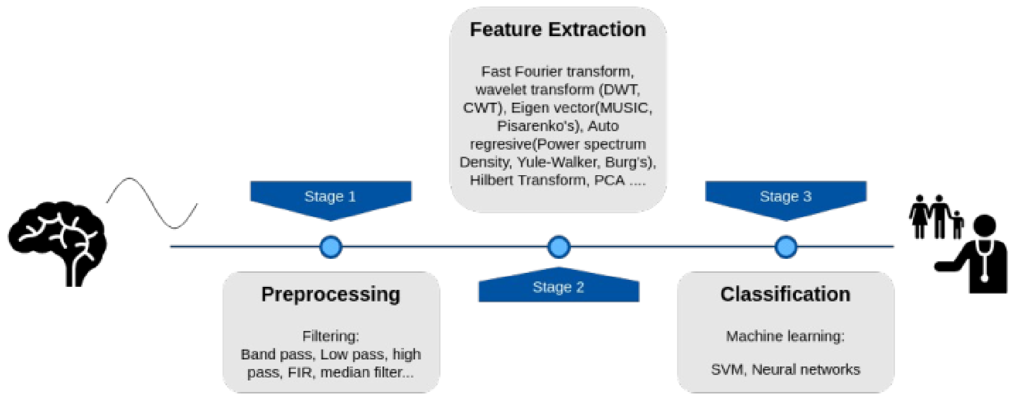

Feature Extraction
Fast Fourier transform, wavelet transform (DWT, CWT), Eigen vector(MUSIC, Pisarenko's), Auto regresive(Power spectrum Density, Yule-Walker, Burg's), Hilbert Transform, PCA ....
Stage 1
Stage 3
Preprocessing
Filtering:
Band pass, Low pass, high pass, FIR, median filter...
Stage 2
Classification
Machine learning:
SVM, Neural networks

Figure 2.1: Generic EEG pipeline

This generic EEG pipeline is customized depending on the target application. An example can be seen in [24], here the sampled signals enter a dimensionality reduction block (preprocessing stage) which computes the mean value, household reduction, accumulation and Principal Component Analysis (PCA) that reduces the input (channels) from 23 to 9 variables. The dimensionality reduction in the preprocessing stage reduces both the memory requirements as well as the computation needed in further stages of the EEG pipeline. In the feature extraction stage, DWT is used and the energy per band is estimated. Finally in the classification stage, a Support Vector Machine (SVM) classifier is implemented.

2.2 Energy efficiency in wearable EEG processing platforms

In conventional EEG processing platforms the processing and classification are done off-chip using processing platforms that do not have energy constraints. Off-chip processing is not suitable for wearable battery-powered EEG system as the energy required for the wireless data transmission is prohibitive. For and 8-channel EEG system, transmitting the raw EEG data by means of a low power radio for off-chip processing consumes around 1.32 mW. When feature extraction and classification[1] are done on-chip, the energy consumption is reduced by 13x and 80x respectively [1].

Using on-chip processing avoids the energy overhead of wireless communication of raw EEG data. As a consequence, the platform in which all the computations are carried out needs to be energy efficient as well. Low-power general purpose processors (CPU) offer flexibility as the algorithm can be easily modified or updated, on the other hand, CPUs are not energy efficient as the operations need to be done entirely by software, increasing the cycle count which in turn increases the energy consumption. On the other side of the spectrum we have the Application Specific Integrated Circuit (ASIC)s, which have high energy efficiency but at the cost of flexibility. An intermediate point is a combination of CPU plus hardware accelerators, which balance the flexibility-efficiency problem.

2.3 Energy efficient EEG platforms

To show the effects of hardware accelerators, in [21] accelerators for four common signal processing algorithms were designed, namely FFT, Coordinate Rotation Digital Computer (CORDIC), Finite Impulse Response (FIR) filter and median filter. As an experiment to measure the improvement, the algorithms were implemented in a 16-bit microcontroller with a hardware multiplier and compared against the same kernels executed by the accelerators. The platform was fabricated in the 13um CMOS technology. The results, presented in Table 2.1, indicate that the energy saved by using accelerators range from 133x to 215x compared to the corresponding microcontroller implementation.

On-chip processing and the use of hardware accelerators are needed in order to build battery-powered EEG detection systems. An example of this is [21], the authors also implemented applications for EEG and ECG in the same platform, achieving energy savings of 10.2x and 11.5x respectively. In [36] a platform for epileptic seizure detection using a 32-bit microcontroller (ARM Cortex-M3) and a 16-bit FFT processor is proposed. In contrary to [21], here only a 256-FFT accelerator is designed. The pipeline has the following stages: sampling of the signal, conversion to the frequency domain using a 256-FFT, computation of the energy per band, median filter, IIR filter and comparison against a reference (prediction). All stages are executed by software except the FFT. To measure the gains, the pipeline was entirely implemented in the Cortex-M3 including the FFT, this approach uses 29.3μW at 1.0 V. The pipeline was also implemented using the Cortex-M3 and the FFT accelerator, the voltage was kept the same (1.0 V). The hardware FFT based implementation of seizure detection consumes 1.6μW at 1.0 V, achieving an improvement of 18x in the power consumption of the application.

[1]Training of the classifier is done off-chip.

Fully dedicated hardware solutions are also available. [45] presents an ultra low power scalable EEG platform, in which a certain degree of customization can be achieved by selecting some parameters according to the user needs, such as the number of channels (1, 2, 4, 8), system clock frequency (64, 128, 256, 512 kHz) and system operating modes (bipolar, referential or average referential). However, we are bounded to the same algorithm and the given operation modes since the algorithm can not be updated once the system is fabricated on silicon. This is perhaps the major disadvantage of fully dedicated hardware solutions.

Table 2.1: **Energy consumption of signal processing tasks: CPU + multiplier vs Hardware accelerator. Measurements @ 1V 10 MHz. Table taken from [21]**

Operation	**CPU & Multiplier Total Energy (nJ)**	**Accelerator Total Energy (nJ)**	**Reduction from Accelerators**	**Accel. Equivalent Gate Count**
32-tap FIR Filter	176	1.22	144.4x	11 k
512-pt FFT	82148	616	133.3x	24 k (logic)
sin x (CORDIC)	279	1.30	215.2x	9.3 k
65-pt Median Filter	114	0.79	149.9x	37 k

2.4 Flexibility in wearable EEG systems

From [1, 21, 36, 45] we have seen that a CPU in combination with (good) hardware processors can reduce both, the energy consumption and area utilization at the cost of flexibility. Flexibility, the ability to run different (or improved) algorithms, is an important characteristic desirable in consumer electronics which, due to the short time-to-market, usually require modification of the application after fabrication. In the context of EEG processing platforms, a flexible solution is needed as the features computed in the EEG pipeline may differ from patient to patient [41]. This is why an energy-efficient full-programmable solution that can cope with health monitoring related applications and algorithms is needed.

2.5 Coarse Grain Reconfigurable Architectures in wearable EEG systems

A promising solution for wearable energy efficient EEG processing systems is the use of CGRAs, composed of FU with a connection between them that can be reconfigured at run time. This means that a programmer can modify the interconnect and instantiate an Application Specific Instruction-Set Processor (ASIP) needed for the current computation, enabling the possibility of mapping different kernels to the same CGRA fabric.

However, CGRAs also share the disadvantages of the hardwired solutions, meaning that it is not possible to add more FU once it is fabricated on silicon. For this reason, an in-depth analysis of the properties of each kernel is required in order to obtain the optimal CGRA dimensions.

2.5.1 Architecture exploration in CGRAs

Architecture exploration in CGRAs is a challenge due to the many possible design choices such as FU, connections, placement and routing, etc. For this reason, even for a single application, finding the most optimal architecture is challenging and involves many trade-offs. Many works have been done that attempt to address the architecture exploration in CGRAs.

An interesting, yet limited approach is given in [38]. In this paper the authors propose a CGRA architecture that has as goals high performance and easy architecture exploration. They present table 2.2, in which they show that homogeneous FU arrays have the lowest Design Space Exploration (DSE) complexity compared to heterogeneous and hybrid arrays but at the cost of more area overhead.

In order to mitigate the DSE complexity of Heterogeneous arrays and to keep the area as low as possible they have grouped the FUs in the architecture in a sub-array creating what they call *mini cores*, which are arrangements of four different FUs of an arbitrary type. Using this approach they form *mini-cores* that implement the complete instruction set and some sort of homogeneous array and hence reducing the design space.

Table 2.2: Comparison distinct array types. Table taken from [38]

	Reusability	**Scalability**	**DSE Complexity**	**Area overhead**
Homogeneous FU array	High	High	Low	High
Heterogeneous FU array	Low	Medium	High	Medium
Hybrid MC array	High	High	Medium/ Low	Medium/ Low

A different approach is taken in [3]. Here the authors perform a systematic architecture exploration using a flexible architecture template (*Adres*) that includes a tightly coupled Very Long Instruction Word (VLIW) processor and a CGRA and integrate it into the same platform, they use their own toolchain (Compiler based on dataflow analysis and graph transformations).

In their experiments[2], they mapped various kernels from the multimedia and telecommunications domain onto different *Adres* instances using their toolchain while systematically changing parameters such as the number of FUs and Register File (RF)s, the interconnection topology, the operation set each FU supports, and the sizes of the distributed RFs. The goal of the authors is to see the effects of modifying the design parameters rather than to achieve a full design space exploration.

In general, due to all the possible combination of design choices and the complexity of current architectures, even with the help of a compiler, a full design space exploration is not yet possible.

2.6 Size a CGRA for a set of algorithms

As mentioned before, a CGRA will only have the Functional units given at design time. If we intend to use the same CGRA for a given set of algorithms, it is required to first map all the algorithms in the set to the CGRA architecture, the final CGRA size is given by the union of the sets of functional units required to run the desired algorithms.

However, the problem of optimally mapping an application (or set of applications) onto a CGRA is NP-complete [46] and still a research topic. The huge design space, the complexity of current CGRAs and the poor compiler support make the mapping problem a big drawback when using CGRA architectures. These factors make even more difficult to take advantage of the application parallelism using minimum hardware resources.

2.6.1 Mapping an algorithm onto a CGRA architecture

When the input algorithm is represented as a data-flow graph (DFG), and the CGRA target architecture, including blocks and their connectivity, is also represented as a graph, then the mapping problem is to embed the applications dataflow graph onto the device graph [5].

[2]To reduce the impact of the scheduling heuristic's inherent randomness, the authors scheduled each kernel five times with different random seeds and select the best result.

Several approaches have been taken in the literature when it comes to CGRA application mapping which range from novel techniques that try to map an application up to proposals that use a more regular Processing Element (PE) structure that makes the work of the compiler a bit easier.

In [46] the authors formulate the application mapping problem considering the routing of PEs the shared resource constraint and the interconnection. They also developed an Integer Linear Programming (ILP) solution and a graph drawing based approach to map the applications onto a CGRA. To measure their solution, the authors use two cost functions that involve: a) Utilizing less number of rows in the CGRA and b) Minimizing the total connection length between PEs. The limitation of this approach is that the proposed ILP model to obtain the application mapping is only applicable to moderate sizes of CGRAs due to high time complexity.

Another very interesting approach is presented in [27]. The paper presents a unified approach combining heuristics and an exact method for application mapping to various CGRA types by using backward simultaneous scheduling and binding (which usually is done sequentially) combined with dynamic graph transformations. The proposed method has as input a functional specification written in C/C++ and the targeted CGRA model, for which they also propose a modelling convention, then the application is compiled to obtain the Control Dataflow Graph (CDFG) using a GCC front-end. The CDFG and the CGRA model are used to generate the mappings using the proposed algorithm. According to the paper, the CGRA model is very flexible and can represent a CGRA with different characteristics such as homogeneous or heterogeneous tiles, with/without RF, regular/specific interconnect, multicycle operations among others. They have tested the proposed method by mapping nine algorithms used in signal processing applications, while varying the CGRA size, RF size and number of tiles. Metrics such as success rate, latency, diversity and efficiency are considered to measure the results of the mapping algorithm. They claim that the proposed method has the highest success rate, good latencies and a good solution space exploration compared with the state of the art.

The authors in [5] propose a more complete approach, they present a unified framework that includes a generic architecture description language, architecture modelling, application mapping algorithms and finally synthesizable logic for physical implementation. In other words it is a generic framework that can be applied to any CGRA, the only drawback is that the target CGRA needs to be modelled using their architecture description language. In this work, they define the cost of a mapping as the summation of all used routing nodes and all used functional unit nodes within the CGRA model (graph).

Although there are more approaches to the problem of the application mapping onto a CGRA, the vast majority need some sort of constraints to be used as a starting point, such as number of FUs, CGRA layout, number of connections/ports, etc. to then generate the dataflow graph and finally map the application.

Chapter 3

Problem statement

Because of the rapid development of new (computational intensive) EEG algorithms, energy efficiency and flexibility are vital characteristics in wireless battery-powered EEG platforms. CGRAs can enable low-power EEG platforms to compute and modify a set of algorithms running on them. However, CGRAs need to be properly instantiated to efficiently accommodate the application. This facts raises the research question of *How can we define a common reconfigurable architecture for the computation of a set of EEG features with a focus on energy efficiency.*

3.1 Contributions

The main contributions of this thesis are:

- Analysis of the EEG features and how to efficiently map them to the *Blocks* CGRA identifying: ideal speed up, possible parallelization opportunities and limitations.
- Evaluation of the efficiency of a variety of EEG features on a platform with a feature-specific CGRA + Reduced Instruction Set Computer (RISC) based processor architecture in terms of speed, area, and power.
- Additionally, we identify possible alternatives for features that cannot be efficiently computed in the CGRA.
- An energy model based on the previous results that incorporates the cost per Arithmetic and Logic Unit (ALU)/Multiplier (MUL) operation, RF and memory accesses, the interconnect, and the cost of idling (FU utilization) and reconfiguration.
- Using this model we propose an energy-efficient CGRA architecture that can be used to compute a set of EEG features. The main goal of this flexible architecture will be to minimize the EDA product, having as key metrics the FU utilization and memory traffic.

Chapter 4

Background

4.1 The seizure detection EEG pipeline

The EEG processing pipeline considered in this dissertation is based on [41]. In this paper a total of 26 features are computed on the platform explained in section 4.2. The sampling is done using 24 electrodes at a sampling frequency of 100 Hz and the signal is processed in epochs with a duration of 2 seconds each. In the filtering stage a 10^{th} order Butterworth band-pass filter (0.5 Hz - 45 Hz) is used to remove the slow drift artefacts and to suppress the interference of the power line (50 Hz), the extracted features are used for a seizure detection application.

The main difference of [41] compared to other EEG pipelines is in the features extracted from the EEG signal. They have proposed new features that improve the classification performance when combined with the common EEG features. These new features are based on synchronization methods and in the spatio-temporal domain methods. The most important features in this EEG pipeline are:

- Frequency domain

  -*Spectral analysis:* Estimates the strength of different frequency components (the power spectrum) of a signal.

- Non-linear features

  -*Approximate Entropy (ApEn):* Used to to measure regularity in the signal.

  -*Hurst exponent.*

- Time-Frequency domain

  - *DWT*: Computes both frequency and location information (location in time) of a signal.

  -Standard deviation: Quantifies the amount of variation or dispersion of a set of data values.

- Spatio-Temporal domain *Hilbert transform*: Measure the instantaneous phase of a signal.

- Synchronization based methods

  -*Maximum linear cross-correlation:* Measure for lag synchronization of two signals

  -*Dynamic Warping Similarity (DWS)*: used to characterize the similarity among EEG channels.

  -*Euclidean Distance Similarity (EDS)*: Used as a comparison with the DWS.

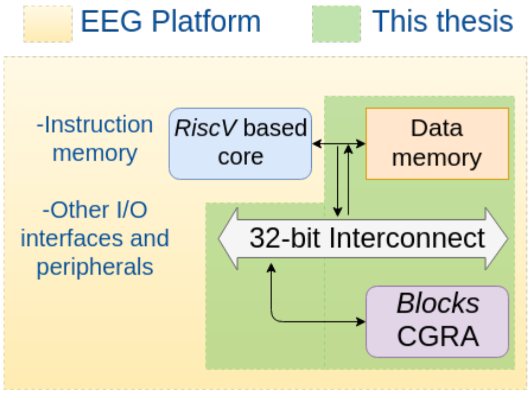
- Post-processing

  -*Moving average of EEG features, average feature on multichannel*

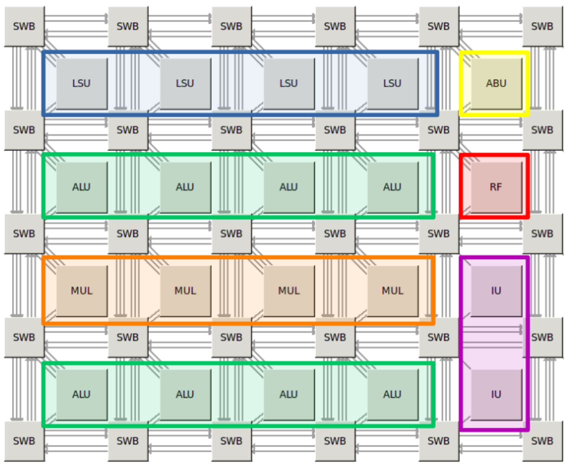
Finally, for the Classification stage, a SVM is used for a binary classification (seizure, no seizure). Recordings from 29 epilepsy patients with intellectual disability are used for training and validation of the model.

4.2 EEG processing platform overview

The block diagram of the EEG processing platform used in this thesis is shown in figure 4.1a, the yellow shaded rectangle represents the complete system [1]. The main components are a *RISC-V* based core, an instance of the *Blocks* CGRA, a data memory shared between the core and the *Blocks* instance, connected by a 32-bit bus. The green shaded rectangle contains the elements that are going to be considered in this thesis. Figure 4.1b shows the instantiation of the *Blocks* CGRA used as a starting point, it is further explained in section 4.3.



(a) EEG processing platform block diagram



(b) *Blocks* instance, taken from the Router tool [42]

Figure 4.1: EEG processing platform and *Blocks* CGRA

4.3 The *Blocks* architecture

The CGRA architecture is based on [42], the difference compared to other CGRA architectures is that the control and data path are independent from each other connected (with buses) to the functional units. The width of the bus (represented by the grey arrow 4.1a) that connects the memory arbiter to the shared data memory is a design parameter that can be selected between 8, 16 or 32 bit wide. In this thesis the platform has a 32-bit bus, which allows a complete word to be read from the data memory in each transaction, taking 3 cycles each.

Currently there are six kind of FU in the *Blocks* architecture: the Load-Store Unit (LSU), the RF, the ALU, the Immediate Unit (IMM), the Accumulate-Branch Unit (ABU) and the MUL. From which each LSU has access to the shared data memory and its own local memory of size 256 words by 32 bit.

Furthermore, the instruction fetch and instruction decoder units can be arbitrarily connected to any other functional unit. This makes it possible to instantiate VLIW-Single Instruction Multiple Data (SIMD) processors that tightly adapts to the application needs in terms of computation. It is also possible to bypass the register file and send directly the values that are required in other stages of the pipeline, reducing the data memory accesses and hence, the energy consumption. In contrary to other CGRA architectures, the *Blocks* architecture is designed with energy efficiency in mind, offering reconfigurability at a functional unit level (ALU, RF, etc.)

The *Blocks* instance used in this thesis is shown in figure 4.1b. It contains 4 LSU (their corresponding data memories are not shown in the figure), 8 ALU, 4 MUL, 1 ABU, 1 RF and 2 IMM. The architecture also contains Switch Boxes (SWB) that enables runtime reconfiguration. The instruction fetch/decoder units are not shown.

[1]Other peripherals and the core's instruction memory are not shown because are out of the scope of this work.

Chapter 5

Reference application benchmark

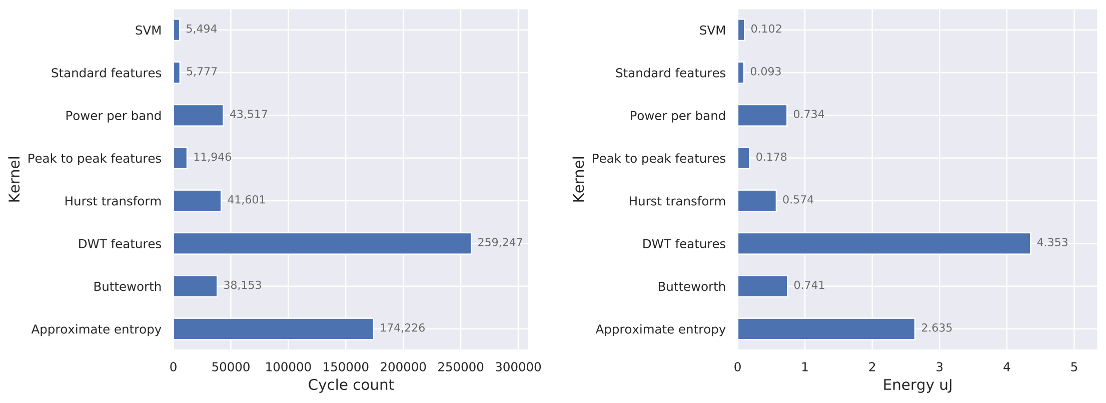
In this chapter the results of an initial benchmark of the EEG reference application is presented. The goal is to obtain the cycle count and energy consumption which will be used to identify the bottlenecks in the pipeline and also as a metric for the comparison of the mappings explained in chapter 6.

5.1 Reference EEG application benchmark

An initial benchmark of the (fixed point) EEG application [41] implemented in C [1] was performed. The cycle count and power consumption where taken from the netlist simulation reports and used to compute the energy consumption. The benchmark was run on the *RISC-V* core, which was synthesized for a maximum frequency of $100MHz$ in a commercial $40nm$ technology using the *Cadence Genus* compiler and simulated using the *Cadence Incisive* simulator.

The application contains the features explained in chapter 4.1. The runtime and energy breakdown of the application are presented in figures 5.1a and 5.1b.

Is clearly seen that the *DWT features* kernel has the largest cycle count, 1.48x, 5.9x and 6.7x larger when compared to the *Approximate entropy*, the *Power per band* and the *Butterworth* features respectively. As expected the features with the largest cycle count are also consuming the most energy.



(a) EEG application cycle count breakdown

(b) EEG application energy breakdown

Figure 5.1: EEG application benchmark results

[1] The C implementation of the algorithms were obtained from the *Brainwave* repository.

From this set, the *Power per band*, the *DWT features* and the *Butterworth* filter were selected. The choice was made based on the computation time and in the importance of the kernel in other EEG applications in the literature.

Finally, table 5.1 shows a detailed breakdown of the selected features. It is clearly seen that the calculation of the main components have the largest cycle count and therefore will be the target for the mapping.

Table 5.1: Detailed cycle count breakdown for selected features

	Feature calculation	**Main components calculation**	**Other calculations**
Power per band	3,246	FFT = 40,271	N/A

DWT features	10,101	DWT-Decomposition=23,388
		DWT-Reconstruction=213,779
			13,007

Butterworth	N/A	IIR = 38,153	N/A

Chapter 6

Algorithm mapping

This chapter presents the analysis of the expected performance and the explanation of the FFT, DWT and the Butterworth filter mappings into the *Blocks* CGRA. For the FFT two algorithms were mapped, the *Korn-Lambiotte* and the generic *Cooley-Tukey DIF*. In case of the DWT, the LWT algorithm was mapped and compared against the existing FWT(Mallat algorithm) version. For the Butterworth filter, a cascaded SOS filter was mapped and compared against the existing 10^{th} order IIR mapping.

The results of the energy consumption presented in each section correspond only to the FUs involved in the computation of each algorithm. The total energy consumption (which includes the energy consumed when loading a kernel to the *Blocks* CGRA), the delay and area are shown in a global comparison in chapter 7 for all the mappings. It is important to keep in mind that 16-bits are used to represent every input in the 256-sample epoch which is stored in the global memory of the platform. For the evaluation, the *Blocks* mappings were synthesized in a commercial $40nm$ technology using the *Cadence Genus* compiler and simulated using the *Cadence Incisive* simulator.

6.1 Fast Fourier Transform (FFT) mapping

This section starts with a brief explanation of the FFT algorithm, their classification and modifications for use in parallel computation. Then, the analysis of the expected performance for a 256-point FFT is shown. Next, the results of the mapped algorithm followed by a review of efficient FFT implementations in the literature are presented. The section concludes pointing to possible optimization opportunities.

6.1.1 The Cooley-Tukey FFT

The *Fast Fourier Transform* is an algorithm to efficiently compute the Discrete Fourier Transform (DFT) of an N-point length input sequence. Although there are many algorithms to compute the DFT, the FFT is still the widely used due to its simplicity and ease of implementation.

The FFT follows a split and conquer approach, which results in considerably savings in computation time. Specially if the input sequence has an even number of samples, i.e., $N = 2n$ samples. By using the FFT algorithm the time complexity is reduced to $N \bullet log_2(N)$, which is lower compared to the time complexity of the direct DFT of N^2 [6], figure 6.1a shows the comparison.

Intuitively the input vector x of length N is divided, into two smaller vectors of length $N/2$, containing the even and odd indexes, this operation is repeated until we obtain a length $N = 2$. This is the smallest DFT and it is called a *Butterfly*, which can be decomposed into 2 complex additions and 1 complex multiplication, the diagram is shown in figure 6.1b. There are two variations of this FFT algorithm, Decimation In Time (DIT) and Decimation In Frequency (DIF) they both have the same time complexity but differ in the way of dividing the input sequence, and in the butterfly structure[1].[19]

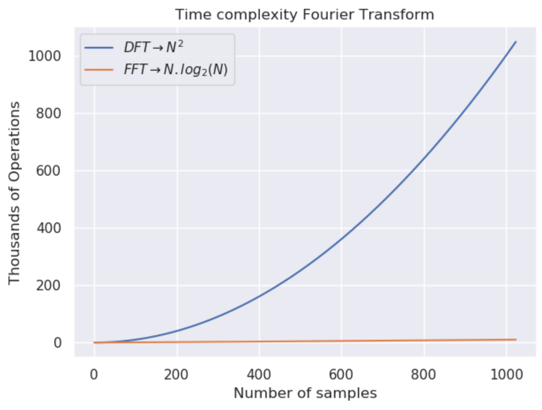
The equation 6.1 shows the expression for the Cooley-Tukey radix 2 DIT FFT:

$$Y[k] = \sum_{n=0}^{\frac{N}{2}-1} x[2n] \bullet \omega_N^{kn} + \omega_N^k \sum_{n=0}^{\frac{N}{2}-1} x[2n+1] \bullet \omega_N^{kn} \tag{6.1}$$

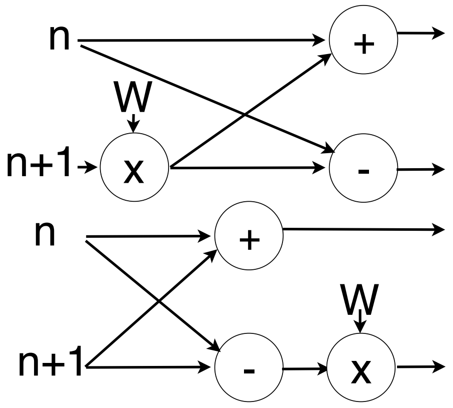
Where N is the sequence length, ω_N represents the twiddle factors for a N-point input sequence, x is the array containing the N elements, n is the index of a given sample, k is the phase angle and Y represents the array that contains the Fourier coefficients. The number of stages that compose an N-point FFT and the number of butterflies needed in each stage are given by equations 6.2 and 6.3 respectively.

$$N_{stages} = log_2(N_{points}) \tag{6.2}$$

$$N_{butterflies/stage} = \frac{N_{points}}{2} \tag{6.3}$$



(a) Time complexity comparison DFT vs FFT



(b) DIT(top) and DIF(bottom) butterfly diagrams

Figure 6.1: FFT time complexity plot and Butterfly diagram

6.1.2 FFT analysis and expected performance on the *Blocks* CGRA

In this thesis, the reference implementation is based on a radix 2 DIF algorithm and computes the FFT of a 256-point sequence length. The mappings on the CGRA are also based on a radix-2 DIF algorithm and its variations. The basic structure is the butterfly, for this reason the analysis starts by looking at the ideal execution of a single butterfly. Taking care that the dependencies between operations are not violated, and assuming an infinite bandwidth for the interconnect, and that all operations take 1 cycle, it is possible to perform a single butterfly in 5 cycles, figure 6.2 shows the timing diagram for a DIF butterfly. This model will be used as a base to analyse the ideal FFT in the *Blocks* architecture.

The number of stages is calculated using the equation 6.2 as $stages = log_2(256) = 8$ stages, each requiring $256/2 = 128$ butterflies.

[1]The DIT algorithm splits the input sequence into even and odd samples and perform the complex multiplication at the beginning of the butterfly whereas the DIF algorithm splits the sequence into a first and a second half and the complex multiplication is carried out at the and of the butterfly.

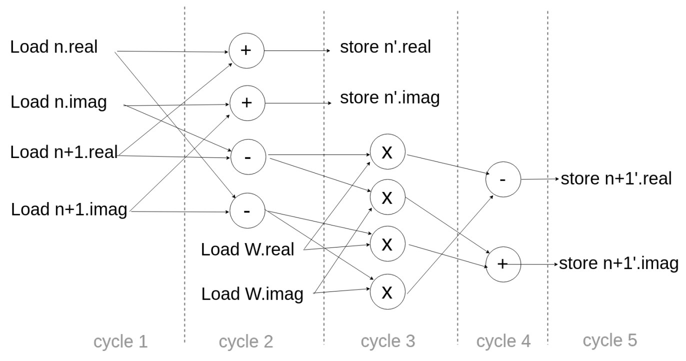


Figure 6.2: Diagram of an ideal Butterfly with parallel execution of operations.

6.1.3 Single Butterfly analysis

The analysis starts by making the following assumptions:

- The number of availbale FU are shown in figure 4.1b.
- The hardware is utilized 100% of the time.

Because of the fact that the input data is read from the global memory and the intermediate results are stored in local memories in each LSU, it is possible to group the 8 stages of the FFT, based on the location of their reading/writing operations, into three (imaginary) categories. Group 1: containing stage 1, group 2: containing stages 2-7 and group 3: containing stage 8. The idea behind this is that, in the ideal case the input data has to be read from the global memory only in the first FFT stage, and has to be written back to the global memory only in the last FFT stage, using the local memories of the LSUs to store the results from intermediate stages, figure 6.3 depicts the idea.

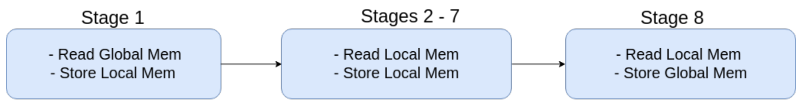


Figure 6.3: Grouping the stages of a 256-point FFT.

Next we assume that the butterflies in all the stages are perfectly pipelined and that we don't have any loop overheads or other index calculations. Then the cycle estimation is done using following formula:

$$\left\lceil \frac{operations_{type}}{resource_{type}} \right\rceil = cycles_{type} \tag{6.4}$$

Where $operations_{type}$ is the total number of operations of the same type and $resource_{type}$ is the number of available resources for executing a given operation type. The result $cycles_{type}$ indicate the number of cycles needed to perform an arbitrary number of operations in the available resources for that type.

Taking the store/load cost equal to 3 cycles the formula 6.4 is applied on the model of the ideal butterfly yielding the results in table 6.1, it shows the breakdown of the latency for a single butterfly in the different stages of the FFT. This information is used to build the time diagram and butterfly breakdown shown in figures 6.4, 6.5 and 6.6 for the first, second and third phases respectively.

After obtaining the timing diagrams the cycle count for each phase is calculated using the following expression:

$$Stage_n = ((stall_{cycles} \times (Butterflies_{stage} - 1)) + Latency_{cycles}) \tag{6.5}$$

Table 6.1: Butterfly breakdown per stage in cycles for a single butterfly

Stage	LSU operations	ALU operations	MUL	Load stall	Store stall	Latency single butterfly
1	$\lceil \frac{6}{4} \rceil = \mathbf{2}$	$\lceil \frac{6}{8} \rceil = \mathbf{1}$	$\lceil \frac{4}{4} \rceil = \mathbf{1}$	$\lceil \frac{6}{2} \rceil = \mathbf{3}$	implicit	7
2 to 7	**2**	**1**	**1**	$\lceil \frac{2}{1} \rceil = \mathbf{2}$	implicit	6
8	**2**	**1**	**0**	implicit	$\lceil \frac{4}{2} \rceil = \mathbf{2}$	5

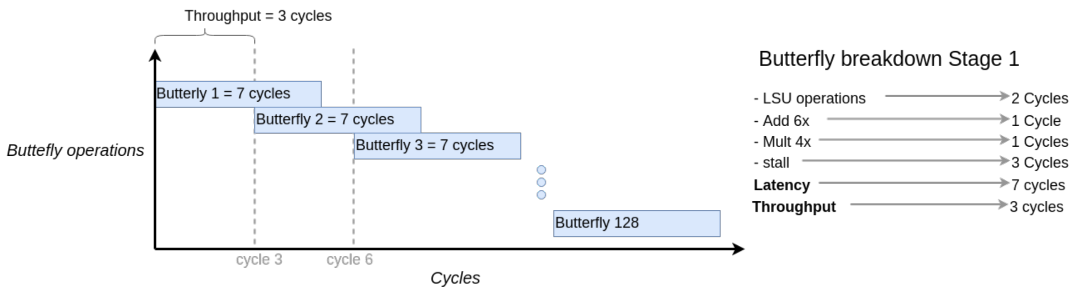


Figure 6.4: Ideal Butterfly pipeline $1^{st} stage$ (left), butterfly breakdown (right)

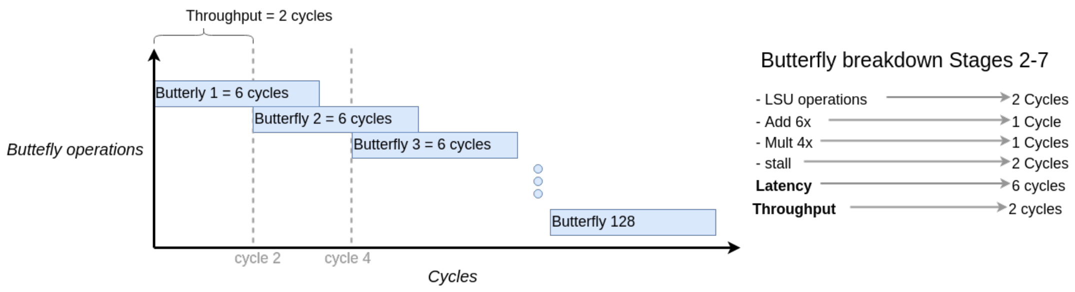


Figure 6.5: Ideal Butterfly pipeline for stages $2 - 7$ (left), butterfly breakdown (right)

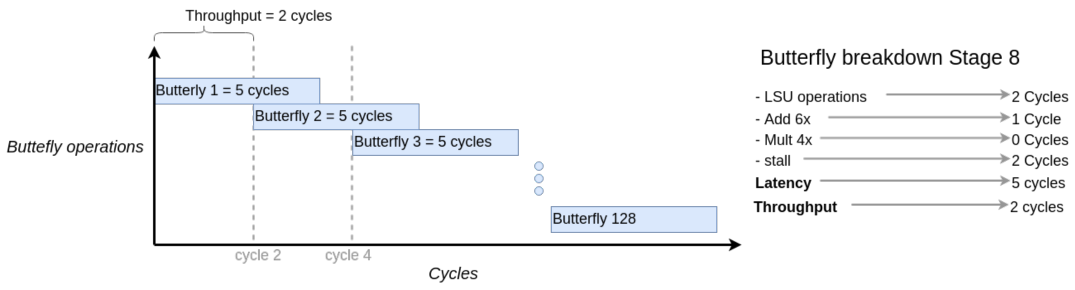


Figure 6.6: Ideal Butterfly pipeline $8^{st} stage$ (left), butterfly breakdown (right)

Where $stall_{cycles}$ is the cost in cycles for loading two numbers from the global/local memory, $Butterflies_{stage}$ is the number of butterflies per stage, $Butterfly_{cycles}$ is the number of cycles for a butterfly operation in the stage and num_{stages} is the number of stages contained in the group. After substitution of the values in the equation 6.5 we obtain an estimated cycle count of 338, 1560 and 259 cycles for the first stage, the second to seventh and for the last stage respectively. The calculations are shown in equation 6.6.

$$\begin{aligned} Stage_1 &: (3 \times 127) + 7 = 388\ cycles \\ Stage_{2-7} &: ((2 \times 127) + 6) \times 6 = 1560\ cycles \\ Stage_8 &: (2 \times 127) + 5\ = 259\ cycles \end{aligned} \tag{6.6}$$

Adding the partial results per stage yields a total of $EstimatedFFT_{cycles} = 2,207$ cycles for the complete 256-point FFT. Finally the ideal speed up, given the hardware shown in figure 4.1b, can be calculated as:

$$Ideal\ speed\ up = \frac{referenceFFT_{cycles}}{EstimatedFFT_{cycles}} = \frac{43,517}{2,207} = 19.71 \tag{6.7}$$

It is clear that this ideal speed up of 19.71 is optimistic, because the analysis is not considering loop overheads, address calculations or other steps needed to prepare the operands in the butterflies (such as sign extension and shifting) that are most of the times present in a physical implementation. However, it can be used as an upper bound that further implementations should try to approximate.

6.1.4 Parallel Butterfly analysis

The possibility of computing two butterfly operations in parallel was explored by following the same analysis explained in section 6.1.3, this time assuming that twice the amount of resources are available. The results are presented in table 6.2. It is possible to see that the latency of the fist and the last stages increases due to the stall cycles generated by the conflicting memory operations.

Table 6.2: Butterfly breakdown per stage in cycles for x2 butterflies assuming double resources

Stage	LSU operations	ALU operations	MUL	Load stall	Store stall	Latency single butterfly
1	$\lceil \frac{12}{8} \rceil = \mathbf{2}$	$\lceil \frac{12}{16} \rceil = \mathbf{1}$	$\lceil \frac{8}{8} \rceil = \mathbf{1}$	$\lceil \frac{12}{2} \rceil = \mathbf{6}$	implicit	10
2 to 7	**2**	**1**	**1**	$\lceil \frac{2}{1} \rceil = \mathbf{2}$	implicit	6
8	**2**	**1**	**0**	implicit	$\lceil \frac{8}{2} \rceil = \mathbf{4}$	7

Using the equation 6.5 and the latency per stage shown in the table above, an estimated cycle count per stage equal to 772, 1560 and 515 for the first, second to seventh and for the last stage respectively is obtained. The calculations are shown in equation 6.8.

$$\begin{aligned} Stage_1 &: (6 \times 127) + 10 = 772\ cycles \\ Stages_{2-7} &: ((2 \times 127) + 6) \times 6 = 1560\ cycles \\ Stage_8 &: (4 \times 127) + 7 = 515\ cycles \end{aligned} \tag{6.8}$$

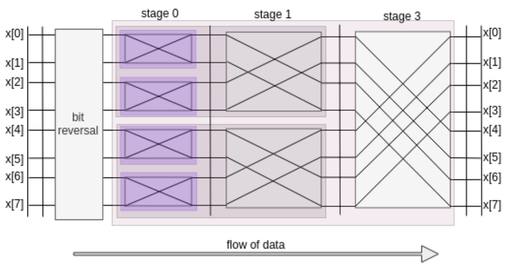
Adding the partial results per stage gives a total of 2,847 cycles for the complete 256-point FFT, and a speed-up of 15.28 as shown in equation 6.9 below. This shows that computing two butterfly operations in parallel decreases the potential speed-up compared to processing a single butterfly operation.

$$Ideal\ speed\ up = \frac{referenceFFT_{cycles}}{EstimatedFFT_{cycles}} = \frac{43,517}{2,847} = 15.288 \tag{6.9}$$

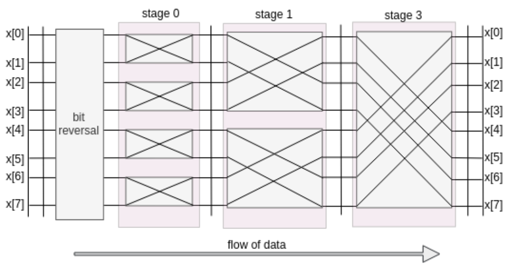
6.1.5 Efficient FFT algorithms

Several variations based on the Cooley-Tukey algorithm exist that intend to efficiently fit parallel architectures, examples of these algorithms are the *Pease*and the *Korn-Lambiotte* adaptations for vector computers which are revisited in [40]. Although the algorithms compute the same result and have similar number of operations, the performance depends on how well the structure of the data flow (load/store patters) of a given algorithm fits the target architecture. In [9] the authors present a good overview of some FFT radix-2 variants. The discussion is presented by classifying the algorithms into two categories:

- Recursive algorithms, which are based on the principle of locality and hence perform better on systems with memory hierarchy, fig. 6.7a.
- Iterative algorithms, which perform the transform stage by stage, fig. 6.7b



(a) Recursive algorithm. The shaded blocks indicate the order of computation, darker shade blocks are computed first.



(b) Iterative algorithm. Each stage is completed before the next starts.

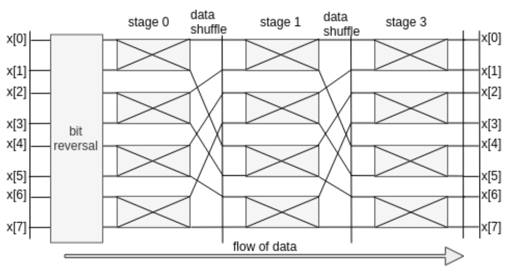
Figure 6.7: FFT algorithm classification

Recursive algorithms are not suited for this thesis due to the fact that the *Blocks* CGRA does not count with a data cache, instead it has a local memory per LSU that can fit the entire input data. Furthermore, implementing complex control flow in *Blocks* is expensive and for this reason only iterative algorithms with simple control flow and high regularity are further considered.

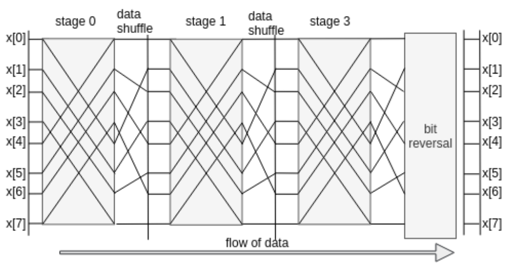
Three iterative algorithms are selected based on their constant geometry and the regularity between stages, i.e., the number of blocks and their vectors length are the same in all the stages of the transform. These algorithms are better suited to parallel architectures since the butterfly operations can be performed as vector operations or as parallel butterflies, furthermore, the constant geometry facilitates the control flow in a CGRA implementation. The selected algorithms are explained next and summarized in table 6.3, the iterative Cooley-Tukey algorithm is included as a mean of comparison.

- Iterative Cooley-Tukey: Can be computed in place, requiring N complex locations for the input data, the geometry changes between stages and a bit reversal phase is needed.
- *Pease* and *Korn-Lambiotte*: Cannot be performed in place, hence 2N complex locations are required. The geometry is constant and the control flow independent of the stage, a bit reversal phase is needed. The *Pease* algorithm performs the complex multiplication at the beginning of each butterfly (DIT Butterfly), and the bit reversal phase at the end of the transform and vice versa for the *Korn-Lambiotte* algorithm (DIF Butterfly).
- *Stockham*: cannot be done in place and requires 2N complex locations, the geometry is constant between stages but it requires a vector shuffle that depends on the stage. The advantage of this algorithm is that no explicit bit reversal phase is required.

From these three algorithms, the *Korn-Lambiotte* is the best fit for the platform. This is because, as mentioned before, the *Korn-Lambiotte* algorithm is based on the DIF Butterfly. This



(a) Example of an 8-point Pease FFT



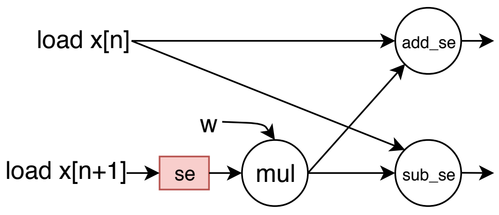
(b) Example of an 8-point Korn-Lambiotte FFT

Figure 6.8: Flow of data in the *Pease* and *Korn-Lambiotte* algorithms

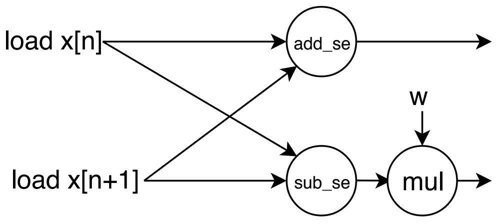
Table 6.3: FFT algorithms summary

Algorithm	**In-place**	**Memory Requirements**	**Geometry between stages**	**Bit-reversal phase**
Cooley-Tukey (Iterative)	yes	N complex locations	Vector length and geometry vary	yes, at the beginning for DIT, at the end for DIF
Pease	no	2N complex locations	Fixed geometry and permutations	Yes, at the beginning
Korn-Lambiotte	no	2N complex locations	Fixed geometry and permutations	Yes, at the end
Stockham	no	2N complex locations	Vector shuffle vary	No, self sorting

is illustrated in figure 6.9 that shows the instructions required to compute both Butterfly structures in the *Blocks* CGRA. The nodes marked with `add_se` and `sub_se` represent the instructions "sign-extend the inputs and add/subtract" currently available in the *Blocks* Instruction Set Architecture (ISA), the nodes marked `mul` represent a multiplication. In figure 6.9a a red shaded rectangle represent the extra sign-extension needed in the DIT Butterfly before the multiplication.



(a) Instructions DIT Butterfly.



(b) Instructions DIF Butterfly.

Figure 6.9: *Blocks* instructions for computing the DIT and DIF Butterfly operations.

6.1.6 Mapping results

The results for the *Korn-Lambiotte* and the *Cooley-Tukey DIF* mappings are shown in table 6.4. The obtained speed-up compared to the FFT running in the *RISC-V* core was 6.83x and 6.02x for the *Korn-Lambiotte* and the *Cooley-Tukey DIF* mappings respectively. It is important to note that the cycle count includes the stall cycles caused by conflicting memory accesses and the extra stall cycles caused by the *AXI* bridge that is used in the bus of the platform, which stalls 3 cycles in every memory accesses.

The achieved energy consumption was 0.34uJ and 0.41uJ for the *Korn-Lambiotte* and for the *Cooley-Tukey DIF*, which corresponds to an improve of 2.16x and 1.8x respectively.

Table 6.4: Simulation results

	Cycle count	**Speed-up**	**Energy (nJ)**
Cooley-Tukey DIT	7,228	6.02	0.40
Korn-Lambiotte	6,055	6.83	0.33
RISC-V	43,517	1	0.73

The energy breakdown for both mappings is shown in figure 6.10. The horizontal axis represents the energy in nano Joules, and the labels on the vertical axis represents the instruction decoders used in the design. The energy of the FUs are included in their respective instruction decoder, i.e. The label '*id_abu*' represent the energy used by the instruction decoder '*id_abu*' plus the energy of the FU '*abu*'.

Differences in the energy consumption

There are three main differences in the energy consumption of both FFT versions:

- In the RF and ALU used for the loop calculations: As expected, it is possible to see that in the *Korn-Lambiotte* version the RF and the ALU responsible for the loop calculations (labeled as *'id_alu_aux'* in figure 6.10) consume less energy compared to the *Cooley-Tukey DIF* version. This is due to the regular structure of the *Korn-Lambiotte* algorithm in which all the stages are exactly the same, avoiding the need of loop calculations and the storage of their control variables in the RF.

- In the twiddle factor LSUs: The structure of the *Korn-Lambiotte* allows a maximum reuse of the twiddle factors in every stage, this is clearly seen in the energy consumption of the LSUs responsible to load the twiddle factors, labeled as '*id_lsu_w*' in figure 6.10, on which the energy consumption is less than half in the *Korn-Lambiotte* mapping when compared to the *Cooley-Tukey DIF*.

- In the input data LSUs: Due to the regularity of the *Korn-Lambiotte* algorithm, the LSUs responsible for loading the input data can efficiently use their automatic address generation capabilities without the need to reconfigure them after every block[2]. This is seen in figure 6.10, where the '*id_lsu*' uses less energy in the *Korn-Lambiotte* mapping.

Furthermore, there is a small difference in the energy used by the multipliers ('*id_mult*'), however, this is because a small implementation optimization in the *Korn-Lambiotte* version on which the multiplications in the fist stage where optimized away. This was done in order to prove that because of the regularity of the *Korn-Lambiotte* algorithm is it possible to further reduce the energy consumption of the multipliers without significant overhead.

The rest of the instruction decoders and their corresponding FUs consume a similar amount of energy in both versions, which is expected as the computational complexity is the same in both algorithms.

Similarities in the energy consumption

The similarities shared by both versions are:

- It is possible to see that in both the *Korn-Lambiotte* and the *Cooley-Tukey DIF* mappings, the LSUs and the multipliers are consuming the largest amount of energy when compared to the other units.

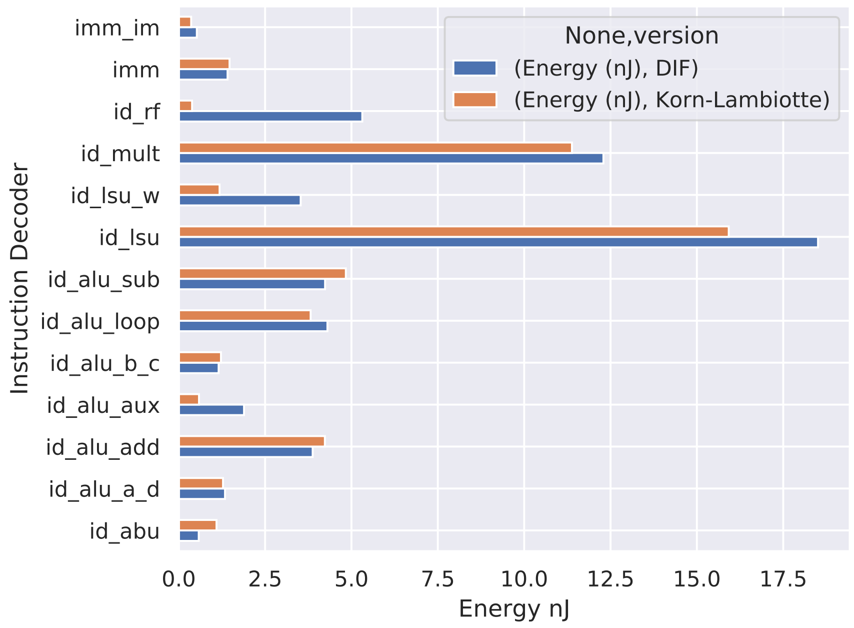
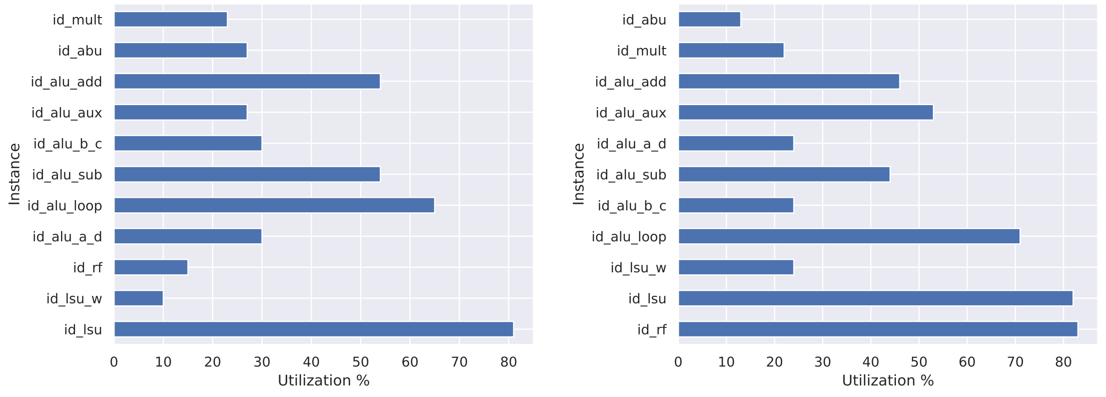


Figure 6.10: *Korn-Lambiotte* and *Cooley-Tukey DIF* energy breakdown.



(a) Utilization breakdown *Korn-Lambiotte* mapping. (b) Utilization breakdown *Cooley-Tukey DIF* mapping.

Figure 6.11: Utilization comparison of the *Korn-Lambiotte* and *Cooley-Tukey DIF* mappings.

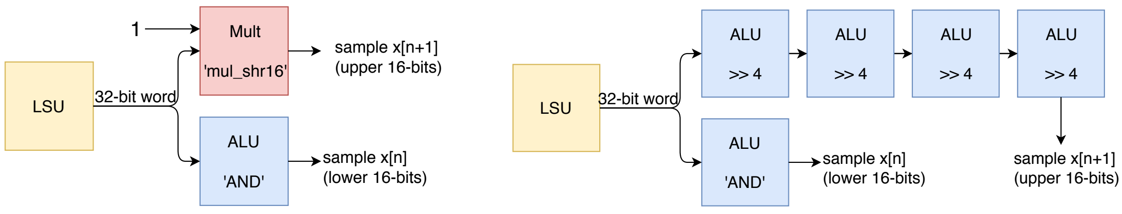
Furthermore, figure 6.11 shows the utilization of both mappings where it is possible to see that in the *Blocks* CGRA, the LSUs responsible for loading the input data are the bottleneck in a FFT. The reason for this is that, although currently the data-path used in the *Blocks* instantiation is 32-bit wide, each LSU has to load two 16-bit values sequentially. It is possible to load both 16-bit values in a single cycle if they are aligned in memory, however, splitting the upper and lower halves of this word is expensive. Figure 6.12a shows the first option, which uses an ALU to obtain the lower 16-bits, and a multiplier[3] to obtain the upper 16-bits. The second option is shown in figure

[2]Every LSU can be configured to compute internally the address for the next load operation. This is done by writing the stride into a configuration register which is added to the address counter after every load operation.

[3]The multiplier in the *Blocks* CGRA counts with the instruction '*mul_shr*16', which multiply two values and shift-right the output result by 16 bits to obtain the upper 16-bits.

6.12b, in which only ALUs[4] are used to split a 32-bit word.

Both splitting alternatives are expensive, the first one would require 2048 extra multiplications to split the 32-bit word, and the second one would decrease the utilization of the hardware when software pipelining is applied due to the chain of shift operations. For these reasons, the sequential load of the input values was chosen as the best option.



(a) Using an ALU and a multiplier.   (b) Using only ALUs.

Figure 6.12: Options for splitting a 32-bit word in the *Blocks* CGRA.

6.1.7 Energy efficient FFT architectures in the literature

Because of the fact that the FFT is widely used there are many energy efficient FFT implementations. However the efficiency is relative to the application domain. This section presents some state of the art implementations, organized in two subsections depending on the approach: custom-hardware based and custom-algorithm/mapping based. It is important to note that although the approach is different the common factor is that a trade-off between area, energy, flexibility and delay is done on most of the architectures.

Based on custom-hardware

An example of this if given in [25]. Although the design aims to be low power, It is implemented in a Field Programmable Gate Array (FPGA) and focused on low latency applications, the energy efficiency comes from the parallelism of the implementation which is based on a custom 64-point 8-parallel FFT architecture, composed of an Address Generation Unit (AGU) and a custom Butterfly Unit (BTU)s. Here a trade-off between energy efficiency and area was done.

An efficient and scalable implementation is given in [28]. Here the goal is to have a configurable BTU able to process radix-2 or radix-4 operations for 8 to 1024 points FFTs. A dedicated Read-Only Memory (ROM) is used to store the twiddle factors near the BTU, the proposed design also uses an AGU and double buffering for latency hiding of input/output data transfers. The results were compared against the *xilinx logicore IP FFT*, the proposed design can operate at 217 MHz compared to the 189 MHz of the *Xilinx IP* at a lower area cost (1/3), the latency however is 1.5 times larger than that of the *Xilinx IP*. In this case there is a trade-off between area and latency.

Another example of the area and energy trade-off is done in [4]. They propose a low power variable length FFT processor. The main features of this work are the combination of scaling factors for the input data and a tailored constant multiplier array in the butterfly unit that use a mechanism to decide whether to use the multiplier array or complex multipliers during the computation. This method saves about 20% of the power consumption compared to the pure complex multiplier implementation. Furthermore, a simple trounding strategy is used to shorten the word length after a multiplication. Although rounding is more accurate, the authors claim that it increases the critical path delay and the energy consumption. The proposed FFT processor is composed of an AGU, a custom DIF BTU, a twiddle factor ROM and a scaling factor generator unit.

[4]The ALU in the *Blocks* CGRA has shift-right by 1-bit and by 4-bit instructions

Based on custom algorithms/mapping

Efficient implementations based on stock resources are also available in the literature, an example is [17], which proposes an efficient mapping of the pipeline single-path delay feedback FFT to FPGAs. They mention that it is not only the FFT architecture that affects the results but also the mapping to the hardware. To prove it, they optimized the radix-2 FFT algorithm for the *Virtex-4* and *Virtex-6* FPGAs, performing transformations that improve the resource usage and resource utilization, specially in multipliers and memories. This was combined with pipelining in order to improve the delay. As a result they obtained a reduced area and an increased throughput per slice of 350% and 400% for the *virtex-4* and *Virtex-6* respectively compared to the literature. Due to the tailored mapping, in this case the trade-off is between energy efficiency-area and portability.

6.1.8 Possible optimizations

This chapter concludes by proposing the following optimizations based on the observations made during the state-of-the-art literature review and on the results of the *Korn-Lambiotte* algorithm mapping.

- **Reduced precision for complex numbers:** Limited bandwidth is one of the bottlenecks in this kernel. The first situation is during the first and the last stages of the FFT in which two complex values, currently represented by 32-bit each, need to be read/written using the 32-bit bus that connects the LSUs to the data memory, meaning that 62 bits are scheduled to be transferred in the same cycle. In *Blocks*, when two memory transactions whose added bandwidth is bigger than the bus bandwidth are scheduled in the same cycle will stall other FU until the memory transaction is complete. It is not possible to reduce the number of operands needed in the butterflies to avoid the stall cycles, however it is possible to reduce the number of bits used to represent the complex values. Using 16 bits to represent each complex number (8-bits for each, the real and imaginary parts) would allow to transfer the two complex inputs needed in the first and the last stages without stall cycles. On the other hand, using 8-bits for each component of a complex number means that samples with values outside the range -128 to 127 would need to be clipped and hence the accuracy could drop.

  Figure 6.13 shows the distribution of the EEG input data[5] used to test and develop the algorithms, the dashed lines represent the minimum and the maximum signed values that can be represented by 8-bits. This data set is composed of about 1.5 million samples from which 92.7% fit in 8-bits. Although the value of most of the samples are in the 8-bit range, rigorous checking is still needed to verify that the results are still accurate enough for the EEG application.

- **Preloaded Twiddle factors:** Currently, the twiddle factors are copied from the global memory and stored in the local memories of the corresponding LSU. Since the twiddle factors do not change, a possible improvement is to have dedicated memory locations in a local memory to store the twiddle factors (pre-loaded values).

- **Real Korn-Lambiotte FFT algorithm:** The implemented algorithms assume complex input values. However, when the input data is composed of real values only the first half of the transform is used, i.e. The second half is a mirrored version of the first half and it is therefore discarded. The EEG dataset is composed of real values only, so by using a real FFT algorithm the required computations could be reduced by half. In [35] the authors show how to implement a real-valued transform using some of the most popular FFT algorithms. The *Korn-Lambiotte* algorithm could be modified to perform a real-value transform.

  A working reference implementation is provided in [33], this is a good starting point for further improvements because it is based on the general radix-2 DIF algorithm that suits the *Blocks* architecture.

[5]See section 4.1 for more details.

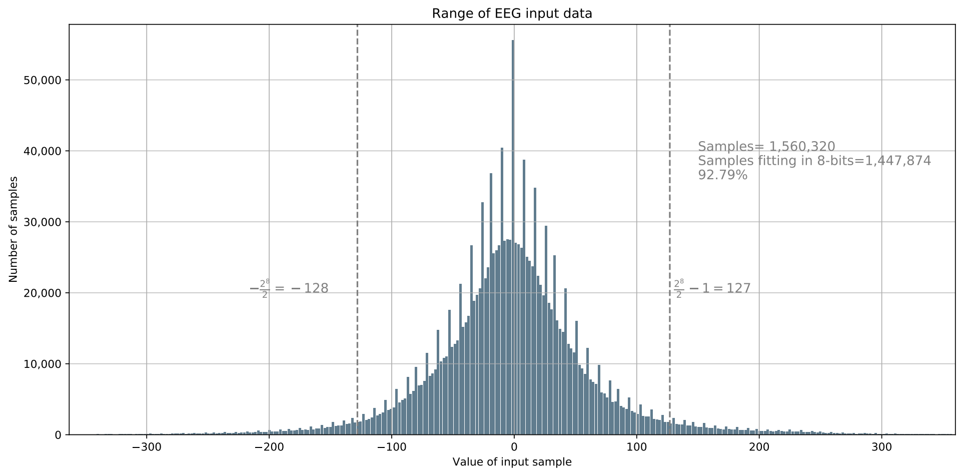


Figure 6.13: Data distribution of the EEG dataset.

6.2 DWT mapping

This section presents the DWT mapping and it is structured as follows, first a brief overview of both methods the FWT (Mallat algorithm) and LWT is presented. Note that this brief explanation is to provide an intuitive idea of how the DWT works, [39] provides an in-depth mathematical explanation and [18] gives a practical approach focussed on the LWT. An explanation of how to derive the lifting equations is also provided in this chapter. The chapter concludes presenting the expected performance analysis followed by the mapping results and possible optimizations.

6.2.1 DWT Introduction

The main feature of the *Wavelet Transform* is that it provides good time and frequency localization. For practical implementation the DWT id used [39]. The intuitive idea is that a signal can be considered as composed of two components, the low and high frequency components, which are obtained from the signal using low-pass and high-pass filters with the same cut-off frequency. The high-frequency component is the difference between the original signal and the low-frequency component. The high-frequency component represents the rapidly varying component of the signal that is complementary to the low-frequency component. After the corresponding filtering, the signals are downscaled completing one level of the transform. This process is repeated for n levels operating on the low-pass filtered signal in the next iteration. At the end, a series of approximations of a signal are created. The detail components contain the difference between adjacent approximation. This process is called multi-resolution analysis. The advantage of this transform is that it is suitable to non-stationary signals on which other methods such as the FFT have troubles. The fields of applications of the DWT are vast from analysis of seismic activity to data compression.

In this thesis the FWT and LWT algorithms are used for the computation of the DWT. A brief explanation is presented next.

6.2.2 FWT

In the FWT, one decomposition level is achieved by passing the signal $x[n]$ through a set of analysis filters, $g[n]$ and $h[n]$ which corresponds to the low-pass and high-pass filters respectively, followed by a decimation step. The output of the high-pass filter $h[n]$ contains the Detail Coefficients (cD) and the output of the low-pass filter has the Approximation Coefficients (cA). The process is repeated using the cA for the next iteration. Figure 6.14 shows 3 decomposition levels using the FWT.

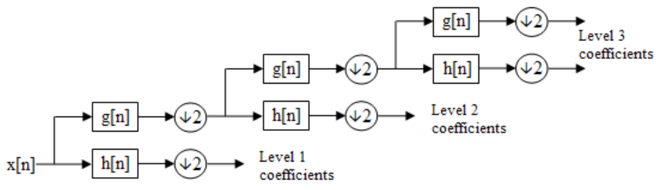


Figure 6.14: FWT with 3 decomposition levels.

6.2.3 LWT

The LWT computes the same coefficients as the FWT with the advantage of half its computational complexity. However the algorithm requires to first factor the wavelet transform into the corresponding lifting steps[6] which is a set of equations that implement the transform. In [7] it is shown how any wavelet transform can be factorized into a finite sequence of lifting steps.

The main principle is to exploit the correlation present in real-life signals to build an approximation. The correlation is local in space and frequency, this means that contiguous/neighbouring samples are more correlated than the samples far apart.

Although the lifting steps vary depending on the number of filter coefficients, the lifting algorithm is the same for any filter. It starts by splitting the signal into even and odd samples[7], because this sets are closely related it is possible to construct a good *Prediction (P)* and the difference or *Detail (d)*. The prediction step handles some of the spatial correlation. However, because the even-sample vector is made by subsampling the signal, a second lifting step is needed to correct it, this step is called *Update (U)* which updates the even samples with smoothed values. The *Predict* and *Update* steps are repeated according to the lifting equations and followed by a scaling step k. At the end of this process we obtain the cA and cD which correspond to one level of the wavelet transform. This procedure can be repeated as many times as required decomposition levels. The input for a new level is the cA of the previous level. Figure 6.15 shows the block diagram of the lifting algorithm. The reduced computational complexity comes from avoiding the computation of the samples that will be subsampled immediately.

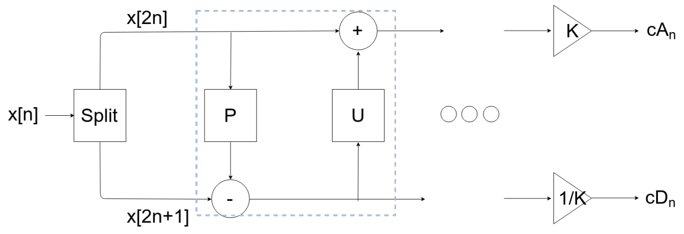


Figure 6.15: Lifting algorithm. The dashed blue box contains the *Predict* and *Update* steps that are repeated depending on the lifting equations, at the end the coefficients are scaled.

The LWT is a good candidate to compute the DWT in the *Blocks* CGRA because of the following reasons:

- The lifting-based transform has half the computational complexity compared to the filter bank-based transform.
- In place computation is possible (no extra memory required).

[6]Lifting Step: The operation of computing a prediction and recording the detail is called lifting step.
[7]This is called in the literature the polyphase components

- Theoretically, all operations within a single lifting step can be done in parallel, the only sequential part is the order of the lifting steps.
- It is possible to perform a DWT that map integers to integers which is important for efficient hardware implementations.[8]

It is important to mention that the LWT has also a negative side effect, namely the factorization of the lifting equations which is based on the euclidean algorithm operating on Laurent polynomials. This process is non-unique, meaning that for an arbitrary filter there exists several factorizations that lead to slightly different lifting equations with different coefficients. According to [7], it is still a research topic the number of different factorizations possible, their difference and a good methodology for choosing the best factorization.

6.2.4 Factorization of the DB4 wavelet into lifting steps

The EEG pipeline explained in section 4.1 uses the *DB4* wavelet for the transform which is composed by 8 coefficients. This section shows (a summary on) how to derive the lifting equations for this wavelet, a detailed explanation of the general algorithm is explained in [18].

The factorization algorithm is as follows:

1. Select the wavelet to use and find the coefficients of the high-pass filter $h(z)$. In this case the *DB4* wavelet has 8 coefficients so we have:

$$h(z) = h_0 z + h_1 z^{-1} + h_2 z^{-2} + h_3 z^{-3} + h_4 z^{-4} + h_5 z^{-5} + h_6 z^{-6} + h_7 z^{-7} \tag{6.10}$$

2. Form the polyphase components as:

$$\begin{aligned} h_{even} &= h_0 z + h_2 z^{-2} + h_4 z^{-4} + h_6 z^{-6} \\ h_{odd} &= h_1 z^{-1} + h_3 z^{-3} + h_5 z^{-5} + h_7 z^{-7} \end{aligned} \tag{6.11}$$

3. Use the euclidean algorithm in the set of equations 6.11 to obtain the polyphase matrix shown in appendix A

4. Evaluate to obtain the lifting equations. Multiply the polyphase matrix with the following evaluation matrix (starting from the right) to obtain the set of lifting equations.

$$\begin{bmatrix} x[2n] \\ x[2n+1] \end{bmatrix} = \begin{bmatrix} cA[n] \\ cD[n] \end{bmatrix} \tag{6.12}$$

   Where $cA[n]$ and $cD[n]$ represents the even and odd components of the signal x.

5. After the evaluation, the resulting lifting equations for one decomposition level are given by:

$$\begin{aligned} cD_1[n] &= a * cA[n+1] + cD[n] \\ cA_1[n] &= cA[n] + b * cD[n] + c * cD[n-1] \\ cD_2[n] &= d * cA[n+2] + e * cA[n+1] + cD[n] \\ cA_2[n] &= cA[n] + f * cD[n] + g * cD[n-1] \\ cD_3[n] &= h * cA[n] + i * cA[n-1] + j * cA[n-2] + cD[n] \\ cA_{scaling}[n] &= l * cA[n] \\ cD_{scaling}[n] &= k * cD[n] \end{aligned} \tag{6.13}$$

[8]This approach is called the integer DWT, it performs addition on integers. However floating point multiplications are still required after which the result is rounded to get an integer value.

This set of equations are sequentially implemented, each running in a loop $N/2$ times, where N is the length of the input signal in that decomposition level[9]. It is important to note that some equations in the set 6.13 have dependencies at the boundaries of the signal, this means that they need previous or future samples at the start and at the end of the signal respectively. This is because we are operating on a finite signal, to cope with this is it necessary to use a method for signal extension at the signal boundaries. In this thesis the *Periodization* method is used for the signal extension at the boundaries.

6.2.5 LWT analysis and expected performance on the *Blocks* CGRA

In this section the expected performance of the LWT is presented.

Due to the similar structure of the lifting equations, it is possible to reuse most of the architecture in all the lifting steps. Figure 6.16 shows the dataflow diagram for the set of lifting equations (excluding eq.cD_2) where white, yellow, blue, purple and red nodes represent memory operations, immediate values, multiplications, additions and register operations respectively. The dashed yellow nodes mean that the immediate value is already present from a previous cycle. The dataflow of eq.cD_2 has the same structure compared to eq.cA_1 and cA_2 but use different operands.

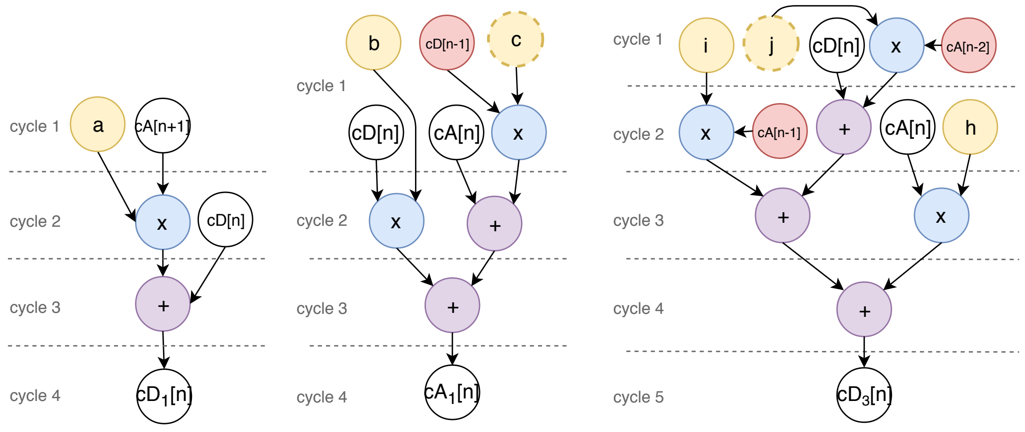


Figure 6.16: Lifting dataflow diagrams. Equation $cD_1[n]$ (left), equations cA_1, and cA_2 (center) and equation cD_3 (right).

The expected cycle count for the LWT is calculated using the following expressions:

$$N_2 = \frac{epoch\_lenght}{2}$$
$$cycles_{lwt} = \sum_{level=0}^{5} \Big(\sum_{eq=1}^{6} throughput_{eq} * \Big(\frac{N_2}{2^{level}} - 1\Big) + latency_{eq}\Big) \tag{6.14}$$

Where $cycles_{lwt}$ represents the total cycle count for the complete 256-point LWT, $level$ stands for the number of decompositions levels of the LWT, eq represents the equations in the set 6.13[10], $throughput_{eq}$ and $latency_{eq}$ are the throughput and latency of the corresponding equation derived from the dataflow graphs shown in figure 6.16.

Table 6.5 provides a breakdown of the estimated cycle count per decomposition level for every lifting equation. The second and third columns of the table represent the latency and throughput. The latency values are obtained by counting the number of cycles needed for an iteration of the corresponding dataflow graph. The throughput values are derived after binding the dataflow graphs from figure 6.16 to 1x ALU, 2x LSUs, 1x RF and 2x IMMs.

[9]Note that N is the length of the input signal for that level, and it is halved for subsequent levels
[10]The scaling of cA and cD is accounted as a single equation

Generally, adding more FUs to the pipeline would benefit the throughput of the graphs. However, because of the hardware reuse and because of the fact that every lifting step alternates the LSU it Load-Store values from, the number of input ports to the FUs, specially in the LSUs, become a limiting factor when more FUs are added.

Table 6.5: Breakdown of the expected LWT cycle count

Eq.	**L**	**T**	**Lvl 0**	**Lvl 1**	**Lvl 2**	**Lvl 3**	**Lvl 4**	**Lvl 5**	**Total (eq)**
read+SE	4	1	259						259
cD_1	4	1	131	67	35	19	11	7	270
cA_1	4	2	258	130	66	34	18	10	516
cD_2	4	2	258	130	66	34	18	10	516
cA_2	4	2	258	130	66	34	18	10	516
cD_3	5	4	513	257	129	65	33	17	1014
scaling	4	2	258	130	66	34	18	10	516
stall			512						512
								Total:	**4119**

The first row of the table shows the cycles required for reading and sign extending the input data, which is done only in the first decomposition level. It is also possible to see that the computation of the eq.cD_3 takes almost twice as much cycles compared to the other equations, this is expected as it also has the most complex dataflow graph in the set. The last row of the table shows the number of compulsory stall cycles caused by the accessing the global memory via the system bus. The total cycle count equals 4,119 cycles and is obtained by adding all values in the last column of the table.

Finally, the LWT is compared against both the reference running in the *RISC-V* and the existing FWT which take 23,388 and 6,555 cycles respectively for the computation of six wavelet decomposition levels.

The expected speed-up for both cases is calculated as follows:

$$Ideal\ speed\ up_{RISC-V} = \frac{referenceDWT_{cycles}}{EstimatedLWT_{cycles}} = \frac{23,388}{4,119} = 5.67 \tag{6.15}$$

$$Ideal\ speed\ up = \frac{referenceFWT_{cycles}}{EstimatedLWT_{cycles}} = \frac{6,555}{4,119} = 1.6 \tag{6.16}$$

The expression 6.15 shows that a speed-up of 5.67x can be achieved by the LWT when compared to the wavelet decomposition running on the *RISC-V* core. This improvement might not seem significant, however it was shown previously in table 5.1 that the reference implementation running on the *RISC-V* also computes the wavelet reconstruction of the signal taking 213,779 cycles, which correspond to 82.5% of the cycle count in the *DWT feature* of the EEG application. Since the only requirement to perform the signal reconstruction using the LWT is to invert the operations[11] in the lifting equations [7], theoretically it would be possible to reuse the LWT and compute the reconstruction of the signal in the same amount of cycles. This means that using the LWT the wavelet decomposition and reconstruction of an EEG channel could be done in 8,238 cycles which is 28.8x faster than performing the same computation in the *RISC-V*[12].

Finally, the analysis suggest that the LWT could also achieve a speed-up of 1.6x when compared to the FWT. The calculation is shown in the expression 6.16.

6.2.6 LWT analysis parallel channels

The analysis for the computation of two EEG channels in parallel is done following the same approach as in section 6.2.5. Due to the *Periodization* method used for the signal extension, the

[11] Additions are replaced by subtractions and vice versa

[12] The wavelet reconstruction using the LWT was not tested during this thesis.

only requirement is to add an extra RF needed in the second channel to temporarily store the samples used for the computation near the boundaries of the signal. This means that the hardware cost for this vector implementation is 2x RF, 2x multipliers, 4x LSU, 3x ALU, 2x IMM and the ABU that keeps the program counter.

Regarding the stall cycles for accessing the global memory, it is possible to keep the count the same as in the scalar LWT version. This is done by using the layout shown figure 6.17 for the storage of the input/output data. In this way, the first and second channel use contiguous half-words in the memory and can be efficiently transferred to/from the LSU's local memories.

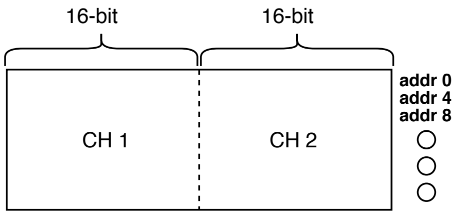


Figure 6.17: Global memory layout

In section 6.2.5, it was shown that a speed-up of 5.67x is already achieved when processing a single EEG channel, therefore in this section the speed-up comparison will be against the reference FWT mapping which also has a vector implementation that compute the transform in 7,323 cycles. This means that, theoretically, a speed-up of 1.78 is achieved by computing two EEG channels using the LWT. The calculation is shown in eq.6.17 below.

$$Ideal\ speed\ up = \frac{referenceFWT_{cycles}}{EstimatedLWT_{cycles}} = \frac{7,323}{4,119} = 1.78 \tag{6.17}$$

6.2.7 Mapping results

The results of the mappings are shown in table 6.6. It is important to note that two EEG channels are processed in the LWT and three channels in the FWT, this is shown in the second column of the table. Furthermore, note that only the cycle count for the wavelet decomposition in the *RISC-V* is used for the comparison of the results.

For the LWT, 3.7x and 0.24uJ were obtained for the speed-up and energy consumption respectively. These numbers correspond to the parallel processing of two EEG channels. In order to compare the results against the reference *RISC-V* implementation it is necessary to normalize both speed-up and energy consumption. This is done by a multiplication by the number of channels in case of the speed-up, and by a division in case of the energy consumption. The normalized speed-up and energy consumption per channel equals 7.4x and 0.12uJ respectively.

The results for the simulation of the FWT mapping are also presented in table 6.6 showing a speed-up of 2.93x and an energy consumption of 0.58uJ for the parallel computation of three EEG channels. After the normalization, the speed-up and energy consumption per channel in the FWT are 8.7x and 0.19uJ respectively.

When the energy consumption per channel is taken into account, an improvement of 3.25x and 2.05x is obtained for the LWT and for the FWT mappings respectively when compared to the reference running in the *RISC-V* core. These results might seem contradictory at the first glance, i.e. Smaller computation time is generally related to higher energy efficiency. However, in this case the computational complexity of the LWT is half when compared to the computational complexity of the FWT, thus it makes sense to have less energy consumption for processing a single channel. A detailed energy breakdown and utilization percentage at the FU level for the LWT is presented next.

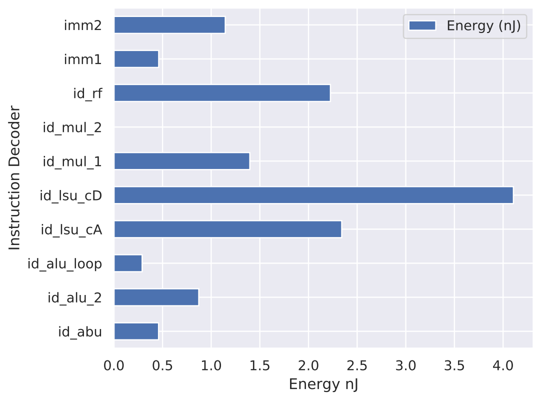
Table 6.6: Simulation results

	Channels	Cycle count	Speed-up	Energy (uJ)	Speed-up per channel	Energy per channel (uJ)
RISC-V (DWT-Dec)	1	23,388			1	0.39
LWT	2	6,308	3.7	0.242	7.4	0.12
FWT	3	7,978	2.93	0.579	8.7	0.19

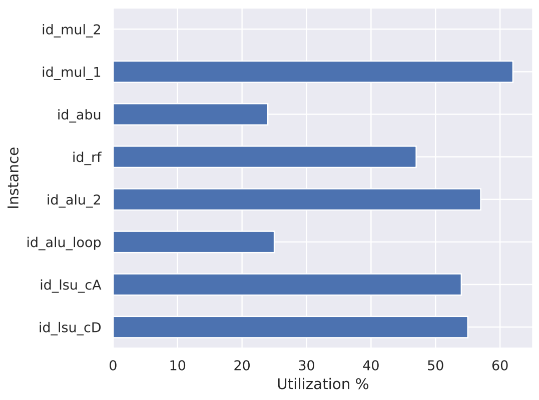
Energy breakdown and utilization

The energy breakdown for the LWT mapping is shown in figure 6.18a. The horizontal axis represent the energy in nano Joules, and the labels on the vertical axis represents the instruction decoders used in the design. The energy of the FUs are included in their respective instruction decoder, i.e. The label '*id_abu*' represent the energy used by the instruction decoder '*id_abu*' plus the energy of the FU '*abu*'. It is possible to see that most of the energy is spend at the LSUs and RFs. This is because the current LWT mapping uses the RF as a shift register in order to cope with the dependencies between previous/future samples in the lifting equations.

Figure 6.18b shows that the utilization percentage of most of the FUs is between 50-60 percent, this is due to the difficulties for processing the samples near the signal boundaries caused by the use of the *Periodization* scheme.



(a) Energy breakdown.



(b) Utilization percentage

Figure 6.18: Energy breakdown and utilization percentage of the LWT

6.2.8 Efficient DWT architectures in the literature

Most of the works regarding energy efficient DWT implementations in the literature use the LWT, this is obvious since, as mentioned before, it has half the computational complexity compared to the filter-bank approach. However, it has been shown in section 6.2.7 that in the conventional lifting scheme the serial processing of the lifting steps decreases data reuse and limits parallelism. Furthermore, it was shown in section 6.2.5 that the similarity of the lifting steps lead to a relatively simple hardware. These are the reasons why the LWT implementations in the literature are based on dataflow optimization of the conventional LWT, hardware multiplexing techniques and optimized multiplication operations. Examples of the state-of-the-art are presented next.

Based on dataflow optimization

In [34] a folded architecture for the LWT is proposed. The design is implemented in a FPGA. At the algorithmic level, the design is based on the optimization of the dataflow of a conventional lifting scheme. The goal of this optimization is to reduce the time needed to calculate the intermediate data and to reuse it efficiently. This approach allows to parallelize and pipeline the lifting stages resulting in 2x speed up compared to the conventional lifting scheme.

At the hardware level, multiplexing was done to reuse resources, using only two adders and two multipliers. The design was tested in the *Altera Stratix II* FPGA using a 12-bit width data bus, and 12-bit quantized coefficients. Their results were compared against other architectures in terms of delay, control complexity, throughput and hardware complexity, achieving the lowest resource requirements.

Based on Hardware optimization

In [10], the authors present three lifting schemes for the DB4 wavelet derived from an efficient factorization of the polyphase matrix. The three designs are implemented using fixed-point representation and 8-bit precision for the coefficients to reduce the hardware cost. However, the main features of this work are the elimination of the scaling stage that cause computation errors and the use of shift-add operations instead of multiplications.

The proposed approach uses from 1.46x to 6.7x less FPGA logic cells compared to other DB4 architectures in the literature.

6.2.9 Possible optimizations

This section concludes by proposing the use of different signal border extension method as an optimization for the LWT. As mentioned in section 6.2.4, *Periodization* is used for the signal extension. The main issue of using this signal extension method in the current LWT implementation is that it can not be efficiently software pipelined generating an overhead when the samples near the signal boundaries are processed. To solve this issue, a different signal extension method can be used. In [26] the influences of the signal border extension in the DWT for EEG applications are investigated. Here it is shown how the selection of an unsuitable border extension method can degrade an EEG spike detector up to 44%. They experimented with nine border extension methods in order to determine in a practical way good signal extension methods for different wavelet families with different number of coefficients. Their results show that for wavelets having up to 8 coefficients (in a spike detector application) the Smooth-Padding of order 0 (SP0)[13] border extension method provides the best results.

6.3 Butterworth Mapping

This section deals with the Butterworth band-pass filter used in the preprocessing stage of the EEG pipeline. The goal of this filter is to remove the low frequency components (< 1 Hz) and the interference from the power line (50/60 Hz). The desired frequency response is represented by the black line in figure 6.19, it corresponds to a 10^{th} order Butterworth band-pass (cut-off at 1Hz and 45Hz) filter that uses double-precision for the coefficients. The blue and orange curves on the figure will be explained later in this chapter.

Two filter structures could be used to implement the filter, namely IIR and FIR. Due to the fact that IIR filters are calculated using the delayed input and output values they need less filter taps in order to achieve the desired frequency response when compared to FIR filters, which use only input values for the computations [32].

Taking into consideration that generally a low computational complexity is related to higher energy efficiency, an IIR is a better candidate and therefore it is used to implement the filter. Figure 6.20 shows the diagram of an IIR filter, it is composed by a non-recursive section that corresponds to the delayed input values and a recursive section that corresponds to the delayed output values, shown at the left and at the right of the figure respectively.

[13]Replicates the first and last samples of the event.

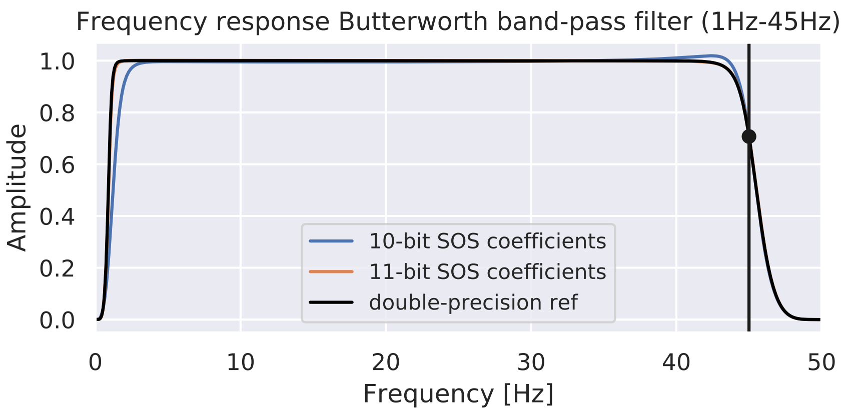


Figure 6.19: Frequency response of a Butterworth band-pass 1Hz-45Hz.

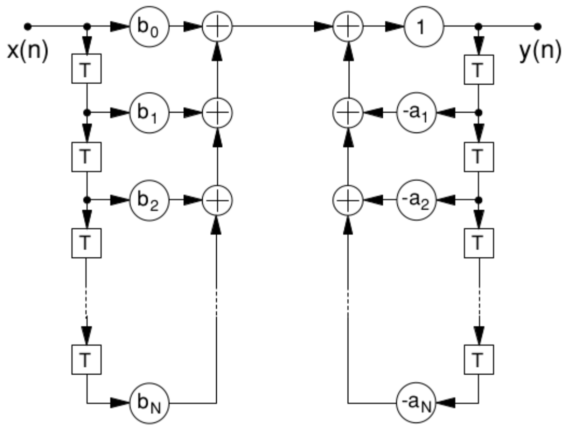


Figure 6.20: Generic IIR filter structure

Next, two arrangements of the IIR structure that implement the frequency response shown in figure 6.19 are analyzed, the first is a direct implementation of a 10^{th} order filter and the second a cascade of SOS. After which it will be explained why this last option is a more efficient and robust alternative.

6.3.1 Direct 10^{th} order IIR Butterworth filter

The transfer function of a generic M_{th} order IIR filter is shown in equation 6.18

$$H(z) = \frac{Y(z)}{X(z)} = \frac{b_0 + b_1 z^{-1} + ... + b_M z^{-M}}{1 + a_1 z^{-1} + ... + a_M z^{-M}} \tag{6.18}$$

Where b and a are the set of filter coefficients and M is the filter order. The first implementation is a 10^{th} order Butterworth filter, which means there are 11 taps[14]. For practical implementation, the filter structure shown in figure 6.20 can be rearranged to form the *Direct form I* or *Direct form II* structures (figure 6.21) while conserving the frequency response characteristics [32].

Direct filters have a straight forward implementation. However, they suffer from negative effects caused by the finite precision of the quantized coefficients and signal representation such as:

[14]$taps = filter_order + 1$

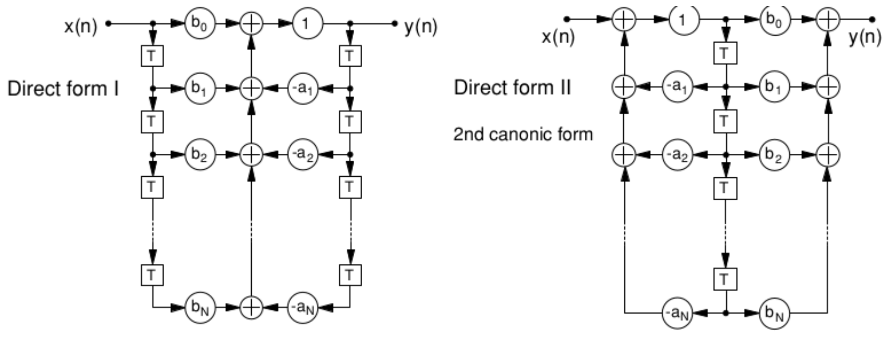


Figure 6.21: Modified IIR filter structures

- Variations in the frequency response due to coefficient inaccuracy.
- Quantization noise because of truncation after multiplication
- Unstable behaviour when not properly designed/quantized

Computational complexity

The Multiply-Accumulate (MAC)s required per sample in an IIR filter using a direct form is calculated as:

$$\begin{aligned} mult_{sample} &= 2M + 1 \\ add_{sample} &= 2M \end{aligned} \tag{6.19}$$

Where M is the filter order, *add* is the number of additions and *mult* is the number of multiplications. In this case the epoch length N is equal to 256 samples, so the computational complexity of the band-pass filter is calculated as follows:

$$\begin{aligned} mult_{filter} &= (2M + 1) * N \\ add_{filter} &= (2M) * N \end{aligned} \tag{6.20}$$

This means that we need to perform $5,376$ multiplications and $5,120$ additions for a 256-length signal.

6.3.2 Cascaded SOS

The transfer function shown in eq. 6.18 could be realized using a cascade of lower order IIR structures. This structures are usually of second order and hence the name SOS [29].

The number i of SOS depends on the filter order M that we try to mimic, and is calculated as: $i = M/2$. This means that in order to achieve the same frequency response of a 10^{th} order IIR filter, a cascade of 5-SOS is needed. Then, it is possible to rewrite the transfer function for a 10^{th} order IIR filter in terms of a cascade of SOS as follows:

$$\begin{aligned} H(z) &= H_1(z)H_2(z)H_3(z)H_4(z)H_5(z) \\ H(z) &= \frac{(b_{01} + b_{11}z^{-1} + b_{21}z^{-2})...(b_{05} + b_{15}z^{-1} + b_{25}z^{-2})}{(a_{11}z^{-1} + a_{21}z^{-2})...(a_{15}z^{-1} + a_{25}z^{-2})} \end{aligned} \tag{6.21}$$

Where H_n represents the transfer function of an individual SOS, and b_{xy} and a_{xy} represent the filter coefficients in which the subindex indicates the coefficient index x of the SOS y[15]. Figure 6.22 shows the diagram for a cascade of SOS in the *Direct Form I.*

[15]For example, b_{01} stands for the coefficient b[0] of the first SOS.

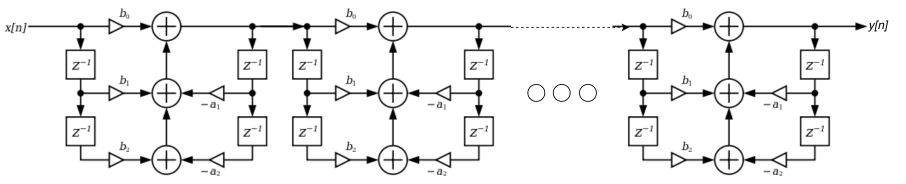


Figure 6.22: IIR filter implemented as a cascade of SOS sections.

6.3.3 Fixed-point implementation

The first step needed to implement a cascaded filter in fixed-point arithmetic is to derive the SOS coefficients from the transfer function of the corresponding higher order filter. This was done by passing the transfer function of the 10^{th} order Butterworth to the function `tf2sos()` of the *scipy.signal* python module.

The obtained coefficients were scaled using 11-bits as it's word-length for a 32-bit accumulator according to the procedure shown in [44]. The reason for choosing 11-bits is that it is the minimum word-length that provides an accurate approximation. Figure 6.23 shows the comparison. It is possible to see that the frequency response of the double-precision is almost identical to that of the 11-bit coefficients. It is also shown how reducing the coefficient-word length by 1-bit already changes the frequency response. Although it might seem a small difference in the frequency domain, the difference in the time domain when 10-bit coefficient are used is easily noted in figure 6.23, again both signals the double precision reference and the signal obtained by using 11-bit coefficients are almost identical, contrary to the signal computed with 10-bit coefficients.

An extra advantage of using 11-bit coefficients is that the resulting scale factor is equal to 256. This means that it is possible to scale-down the output of the filter by right-shifting 8-bits, which fit the available shift instructions in the *Blocks* CGRA[16]. The real approximation of the quantized coefficients for the 5-SOS is shown in the listing below.

```
sos1 = [ 0.5390625    1.08203125   0.54296875  -1.51953125  -0.59765625 ]
sos2 = [ 1.           2.           1.          -1.73828125  -0.828125   ]
sos3 = [ 1.           0.          -1.           0.21484375   0.6796875  ]
sos4 = [ 1.          -2.           1.           1.8984375   -0.90234375 ]
sos5 = [ 1.          -2.           1.           1.95703125  -0.9609375  ]
```

Listing 6.1: Real approximation of the quantized SOS coefficients

6.3.4 Computational complexity

The number of additions and multiplications in a cascaded IIR filter is obtained using the following expression:

$$\begin{aligned} mult_{cascade} &= ((2M-1) * C_n) * N \\ add_{cascade} &= (2(M-1) * C_n) * N \end{aligned} \tag{6.22}$$

Where M is the filter order of the sections, C_n is the number of cascaded sections and N is the number of samples. However, from listing 6.1 it is possible to see that many coefficients are '1' or '0' and hence, unnecessary multiplications and additions can be omitted. This means that only a total of 17 multiplications and 19 additions are required to process a single sample. Therefore, for a 256-sample epoch only $4,352$ multiplications and $4,864$ additions are needed, saving 256 additions and $2,048$ multiplications when compared to the direct implementation of a 10^{th} order IIR filter.

[16]Available shift instructions: shift 1, shift 4, via an ALU and shift 8 via a multiplication by 1.

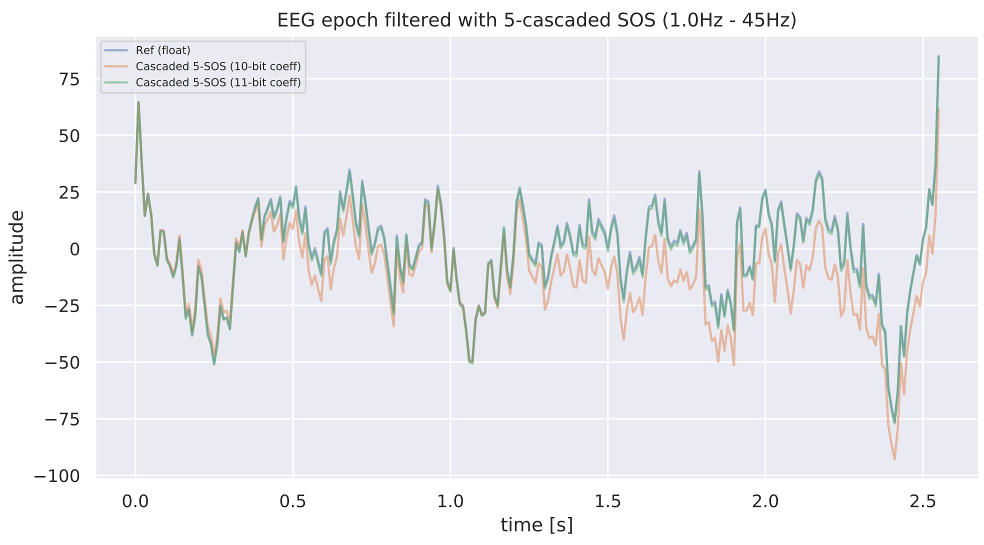
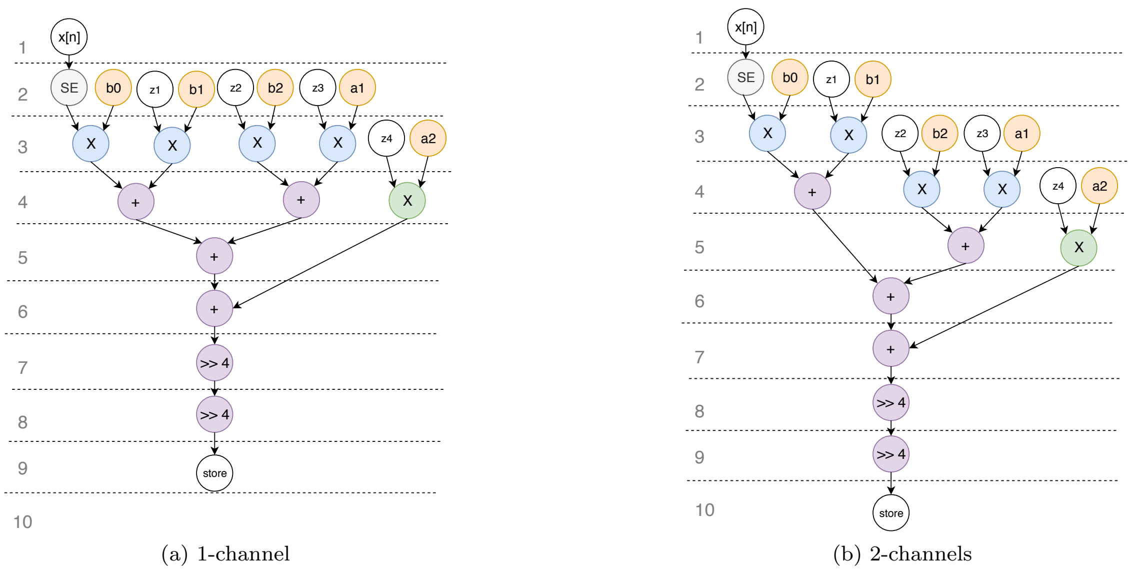


Figure 6.23: Comparison of the output signal using 10-bit, 11-bit and double-precision coefficients.

6.3.5 Expected performance in the *Blocks* CGRA

The analysis starts by deriving the dataflow diagram for the SOS from the transfer function shown in eq.6.21, this is done taking into account the available resources shown in figure 4.1b, figure 6.24a shows the dataflow for the computation of a single channel, where white, blue, yellow and purple circles represent memory operations, multiplications, immediate values and ALU operations respectively. Loading from the global memory and sign-extending the input sample, shown in the first two cycles in figure 6.24a, is done only in the first SOS.



(a) 1-channel (b) 2-channels

Figure 6.24: Dataflow of a SOS

The total cycle count for a pipelined cascaded filter can be estimated with the expression 6.23

shown below:

$$cycles_{filter} = \Big(\Big(\sum_{n=1}^{sos-1} throughput_n\Big) + latency_{sos}\Big) * N \tag{6.23}$$

Where $cycles_{filter}$ represents the expected cycle count of the complete filter, sos is the number of SOS, $throughput_n$ and $latency_n$ are the throughput and latency for the n^{th} SOS and N is the epoch length.

Because of all the SOS have the same structure, the all have the same latency and throughput which are derived from the dataflow graph shown in figure 6.24a. In this case, the throughput is 2 cycles. This is because there are 5x multiplications in the dataflow that need to be done using the 4x available multipliers. The latency is obtained by counting the number of cycles for a complete iteration of the graph, meaning that the latency for the first SOS is equal to 9 cycles and equal to 8 cycles for the remaining four sections. After substituting all the values in equation 6.23, a value of $4,096$ cycles is obtained for an epoch length of 256-samples.

The reference Butterworth filter has a cycle count of 38,153 cycles. Hence, the expected speed-up obtained by the use of a cascaded filter is equal to 9.3x, equation 6.24 shows the calculation.

$$Ideal\ speed\ up = \frac{referenceButterworth_{cycles}}{EstimatedSOS_{cycles}} = \frac{38,153}{4,096} = 9.3 \tag{6.24}$$

6.3.6 Expected performance Parallel Channels

The analysis for filtering two channels in parallel is done in a similar way. But this time the resources have to be shared between two channels, and therefore, the dataflow has to be slightly adjusted as shown in figure 6.24b. The throughput of the graph is 2 cycles and it is still determined by the number of multiplications while the latency is equal to 9 cycles. The values are substituted in equation 6.23 giving a total of $4,352$ cycles and a speed-up of 8.76x, but because two channels are computed in parallel then the speed-up becomes $8.76\ \times\ 2 = 17.53$ compared to the reference. From this, it is obvious that filtering two channels in parallel is more efficient in terms of number of cycles and therefore it was implemented, the results are presented in section 6.3.7.

6.3.7 Cascaded SOS Butterworth mapping results

The cascaded SOS filter was mapped to the *Blocks* CGRA, the results are compared against the existing 10^{th} order Butterworth *Blocks* implementation and against the reference running on the *RISC-V* core.

Table 6.7 presents the results, which yield an energy consumption of 0.876uJ for the computation of two EEG channels in parallel using the cascaded filter and 1.037uJ for the computation of four channels in parallel using the 10^{th} order IIR filter. In order to make a fair comparison, it is necessary to normalize both speed-up and energy consumption numbers. This is done by accounting for the number of processed channels in each mapping. The energy and speed-up per channel are shown in the last two columns in table 6.7. Looking at the energy consumption per channel it is obvious that both mappings provide an improvement compared to the *RISC-V*. However, the direct 10^{th} order Butterworth is the most efficient implementation consuming 0.269uJ per channel, this is roughly half the energy consumption of the cascaded SOS filter. This is correlated with the speed-up per channel, which is equal to 9.36x and 4.64x for the direct 10^{th} order Butterworth and for the cascaded SOS filter respectively. The explanation of the relatively small speed-up and high energy consumption for the cascaded SOS is explained next in more detail.

Energy breakdown and utilization

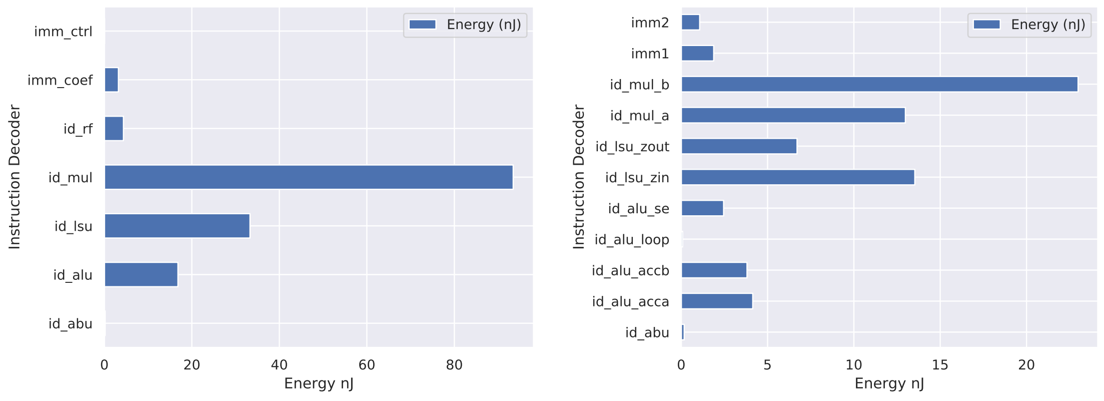
The energy breakdown for both versions are shown in figures 6.25a and 6.25b. The labels in the vertical axis of both figures represent the instruction decoders and their corresponding FU. The horizontal axis shows the energy consumption in nano Joules. In the figures, it is possible to see that in both designs the multipliers and the LSUs are consuming most of the energy.

Table 6.7: Simulation results

	Channels	Cycle count	Speed-up	Energy (uJ)	Speed-up per channel	Energy per channel (uJ)
10 order	4	16,258	2.34	1.037	9.36	0.269
Cascaded-SOS	2	16,448	2.32	0.876	4.64	0.438
RISC-V	1	38,153			1	0.741

The figure 6.26 shows the utilization plots for both designs. On the vertical axis appear the labels of the instruction decoder, the horizontal axis shows the utilization percentage of each unit. Figure 6.26a shows the utilization for the 10^{th} order IIR filter, where all the instruction decoders have an utilization higher than 60%, being the LSUs the bottleneck with an utilization of 80%.

This is contrary to the utilization of the units in the cascaded SOS filter, where the utilization of the units is 30% or below. This is due to the dependencies between SOS, since the hardware was multiplexed to implement the 5-SOS and because of an input sample has to go through the entire cascade, it is not possible to start the computation of the next section before the result of the previous section is ready. This problem can be seen in the dataflow diagram of a SOS shown in figure 6.24b in the long dependency chain of ALU operations after the fifth cycle. A possible solution to this problem is presented in section 6.3.9.



(a) Energy breakdown 10^{th} order Butterworth.

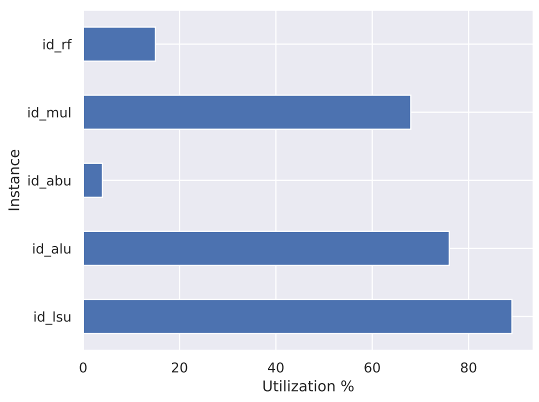
(b) Energy breakdown SOS.

Figure 6.25: Energy breakdown Butterworth filters.

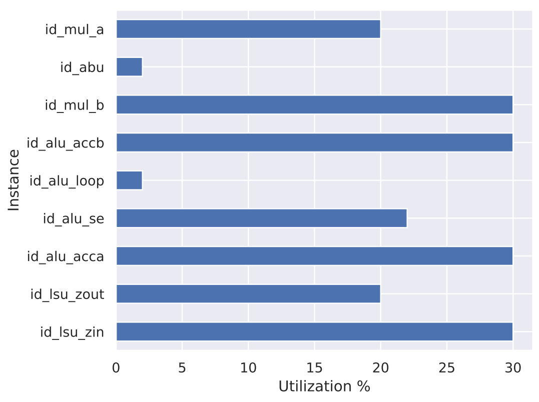
6.3.8 Butterworth filters in the literature

In [37], an evaluation and comparison of software and hardware based optimizations is presented. At the software level, multirate filtering, coefficient optimization, coefficient ordering and coefficient scaling were explored. The coefficient optimization techniques try to reduce the switching activity of the hardware (Hamming distance in the datapath) by modifying and ordering the filter coefficients. The tests were done in a microcontroller with DPS extensions. However, their results show that in average a reduction of only 2% in the energy consumption was achieved by the coefficient optimization methods. Experiments with the multirate filtering technique, which consist of decimating the filter into subfilters in order to reduce the computational complexity, showed that it is possible to reduce the multiplications and additions by 25% when compared to the direct approach.

As for the hardware based optimizations, they explored multiplierless implementations based on distributed arithmetic and shift-add operations. For the measurements, they have implemented a



(a) Instruction decoder utilization 10th order Butterworth.



(b) Instruction decoder utilization SOS.

Figure 6.26: Utilization comparison of the Butterworth filters.

sequential and a parallel reference filters in a *Cyclone 3* FPGA. The results show that the shift-add based implementation already achieves a reduction in the energy consumption of 94% compared to the sequential reference filter and 15% compared to the parallel reference filter.

The approach based on distributed arithmetic performed poorly, consuming more energy than the reference filters. The authors claim that the memory technology they used was not suitable for the implementation.

Distributed arithmetic approaches present another problem, namely, the amount of memory required increases exponentially with the filter order, for a 10th order IIR filter a total of 2^{10} memory locations would be required to store the precomputed values. An attempt to reduce the memory requirements of distributed arithmetic is presented in [8], here the proposed approach reduces the memory requirements to $2^{filter_order-1}$ locations, meaning that 512 locations would be required for a 10th order IIR filter.

6.3.9 Possible optimizations

The chapter finalizes by suggesting the following optimization for the cascaded filter:

- Optimize unnecessary operations: As shown in section 6.3.4, redundant operations can be optimized away. However, this comes at the cost of a more elaborated scheduling being performed by hand.

- Modifications to the computation flow: Software pipelining was not efficiently applied in this kernel because of the cascaded structure on which every input sample flows through the entire cascade, meaning that adjacent SOS depend on the previous SOS output, figure 6.27 shows the current flow of a sample, where the node labeled with 'GM' represents the global memory. A possible solution is to modify the order of the computation as shown in figure 6.28, the node labeled with 'LM' represents the local memory of a LSU. Here, the entire epoch is processed in every SOS before going to the next section. Although this would require 256 extra locations for the storage of the partial results, it would allow better software pipelining having as a result better utilization of the FUs.

- Use the *Transposed Direct Form II* for each SOS: According to [30], the *Direct Form I* is a better alternative for Digital Signal Processing (DSP) processors that count with a wide-bit accumulator, this is because the wide accumulator can handle possible overflows in the single summation point of the *Direct Form I* structure. In the cascaded implementation presented

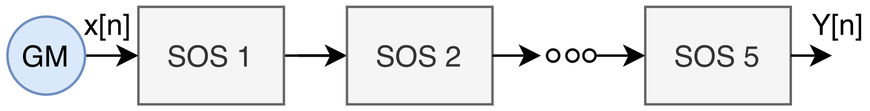


Figure 6.27: Every input sample $x[n]$ goes through the entire cascade.

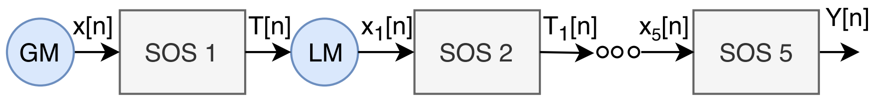


Figure 6.28: The entire epoch is passed through each SOS, locally storing the partial results.

in this thesis, a possible overflow was accounted at the scaling of the coefficients and data previous to the computation, therefore it is not strictly required to use the *Direct Form I*.

The main improvement of using the *Transposed Direct Form II* structure is that only two memory locations per SOS are needed for the storage of the previous input/output values, making a total of 10 memory locations for the five SOS that compose the cascade used in this thesis.

Chapter 7

Performance comparison

This chapter presents the performance comparison of all the available kernels running in the *Blocks* CGRA. It is organized as follows, in section 7.1 the cycle count of the kernels are compared. Section 7.2 presents the energy consumption of the kernels, which is shown separately for loading a kernel into the *Blocks* instruction memory and for the computation phase of the entire *Blocks* instance. The chapter concludes by presenting an area comparison of the mappings.

7.1 Cycle count

The cycle count obtained from the netlist simulations is presented in figure 7.1, in the vertical axis the labels indicate the mapping and the horizontal axis represents the cycle count. The blue bars show the cycle count used for loading a kernel into the *Blocks* instruction memory, whilst the orange bars represent the cycle count used in the computation of the kernel itself.

It is possible to see that the number of cycles spend in loading a kernel is directly related to the program size, which is shown in table 7.1 for all the kernels. The *Cooley-Tukey DIF* is the mapping that takes the longer to load due to the fact that it is fully unrolled in order to reduce the branching overhead.

A similar case is seen in the LWT mapping, on which the large code size is mainly due to the *Periodization* method used for the signal extension at the boundaries. As explained before, this method require special handling near the signal borders and can not be efficiently software pipelined which result in increased code size.

Although a smaller cycle count would be preferred when loading a kernel to the *Blocks* instruction memory, larger kernels can be amortized among the computation of several EEG channels.

Regarding the cycles spent for the actual processing, most of the mappings achieved a similar count except for the Butterworth and the cascaded SOS filter that use more than twice the amount of cycles compared to the other mappings.

Table 7.1: Program size per kernel.

Kernnel	Butterworth	Cascaded SOS	FFT-Korn	FFT-DIF	DWT-Mallat	DWT-Lifting
Program size (Kb)	2.30	3.70	2.88	7.17	6.32	6.77

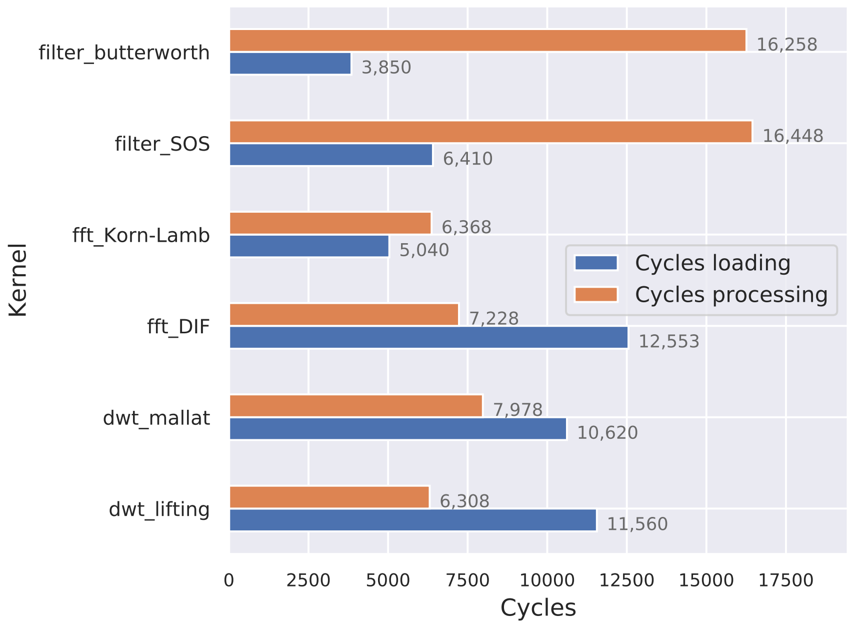


Figure 7.1: Cycle count comparison.

7.2 Energy comparison

The energy results presented in this section were obtained using the timing and power information from the netlist simulations. It is important to keep in mind that some of the mappings compute more than one EEG channel in parallel, as shown in table 7.2, this has to be taken into account during the analysis. Figure 7.2a shows the energy consumption for loading a kernel into the *Blocks* instruction memory, and for processing the kernel represented by the blue and orange bars respectively. It is important to note that the energy results shown in this section include the *Blocks* top module whereas the results presented in chapter 6 include only the energy consumption of the FUs directly involved in the computation.

As expected, larger kernels consume more energy when loaded to the instruction memory, an example of this is the LWT on which it is 2.4x more expensive to load the kernel than the actual computation. Another example is the *Cooley-Tukey DIF*, which program is fully unrolled consuming an equivalent of half the computation energy just for loading the kernel. However, the energy for loading a kernel is not considered critical, and similar to the loading cycle count, it can be amortized among the computation of several EEG channels.

Figure 7.2b presents the normalized energy obtained from the division of the processing energy by the number of channels computed in parallel. It is possible to see that the *Cooley-Tukey DIF* consumes the most energy when compared to the rest of the kernels. This is due to the fact that both FFT algorithms implemented in this thesis are complex-valued algorithms. The *Korn-Lambiotte* FFT consumes less energy because of two previously mentioned reasons, the first is that all the multiplications by ’1’ on the last stage of the transform were optimized away. The second reason is that due to the structure of the *Korn-Lambiotte* algorithm, it is possible to have a maximum reuse of the twiddle factors at every FFT stage avoiding the need of reloading them.

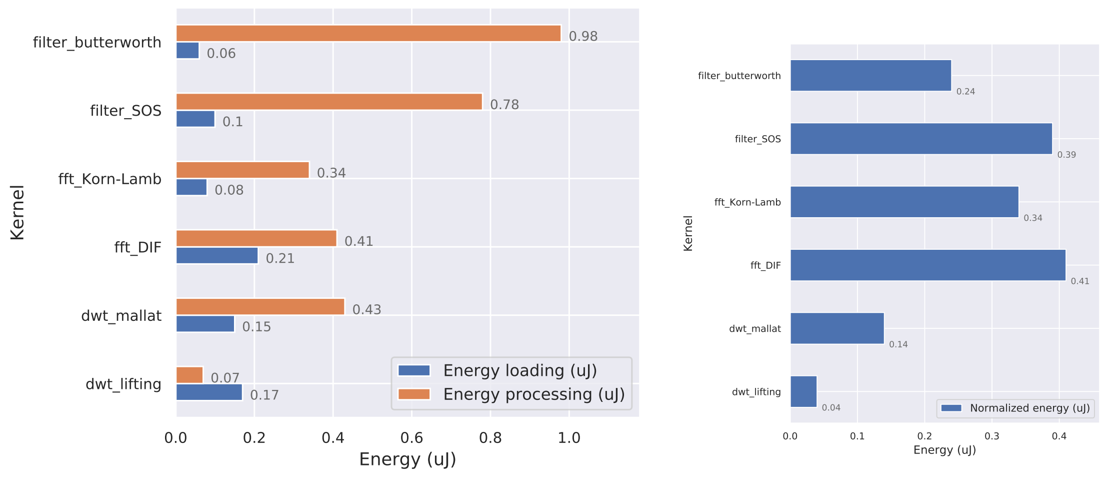
The second most energy consuming mapping is the cascaded SOS filter consuming 0.39uJ. This high energy consumption comes from the large dependency chain that prevents the pipelining of SOS.

Finally, and contrary to the expectations, both DWT kernels consume the least amount of energy compared to the rest. Being the LWT the most efficient with an energy consumption of

0.04uJ, 3.5x less compared to the FWT mapping. There are two reasons for this, the first is the reduced computational complexity of the LWT, which is only half the computational complexity of the FWT. The second reason is that, since both algorithms require previous samples to compute the current one, it is necessary to store them in a shift-register fashion, updating them every time a new sample enters the pipeline. In the LWT, the lifting steps require at most two previous samples to compute the current one, contrary to the FWT which is filter based and requires eight previous samples to compute the same output and thus, more energy is spend shuffling previous samples.

Table 7.2: Channels processed in parallel per kernel

Kernnel	Butterworth	Cascaded SOS	FFT-Korn	FFT-DIF	DWT-Mallat	DWT-Lifting
Processed ch.	3	2	1	1	4	2



(a) Energy comparison

(b) Normalized processing energy

Figure 7.2: Energy results per kernel

7.3 Area comparison

This section presents the area occupied by each *Blocks* mapping which includes the cell and net area obtained from the synthesis reports in the 40nm technology. To understand the differences in the area utilization, it is necessary to take into account the area cost per FU and the number of FUs per mapping.

The table 7.3 shows the area cost in um^2 per FU in the *Blocks* CGRA. The first column of the table represents the area cost for an Instruction Decoder (ID) plus its corresponding instruction memory. Similarly, the second column of the table represents the cost for a LSU plus its local memory. The rest of the columns are self explanatory. It is possible to see that the IDs are the most expensive units in terms of area using $9,883um^2$, even more expensive compared to the multipliers that use $8,865um^2$. On top of this, each LSU counts with a local memory that adds $63,096um^2$. The big area used by a memory is also seen in the ID instruction memory that uses $22,603um^2$ compared to the $503um^2$ used by the ID itself.

The table 7.4 shows the amount of resources used in each mapping. The header of the table indicates the FU and the first column indicates the mapped algorithm.

Finally, the area results per mapping are shown in figure 7.3, it is possible to see that both FFT mappings have the largest area. The difference is mainly caused by the number of IDs (13), which are one of the most expensive units in terms of area. The rest of the mappings are relatively close to each other having an area utilization in the 600k *um* range.

Table 7.3: Area cost per FU in the *Blocks* CGRA

	ID + IM_ID	LSU + LM_LSU	Mul	RF	ALU	IMM	ABU
Area cost (um2)	503 + 22,603	9,883 + 63,096	8,865	5,144	2,763	404	396

Table 7.4: *Blocks* functional units per mapping

	IDs	LSUs	ALUs	Muls	ABU	IMM	RF
Cooley-Tukey DIT	13	4	8	4	1	2	1
Korn-Lambiotte	13	4	8	4	1	2	1
LWT	10	4	3	2	1	2	2
FWT	12	2	9	4	1	3	1
10 order IIR	7	4	4	4	1	2	4
Cascaded SOS	11	4	7	4	1	2	1

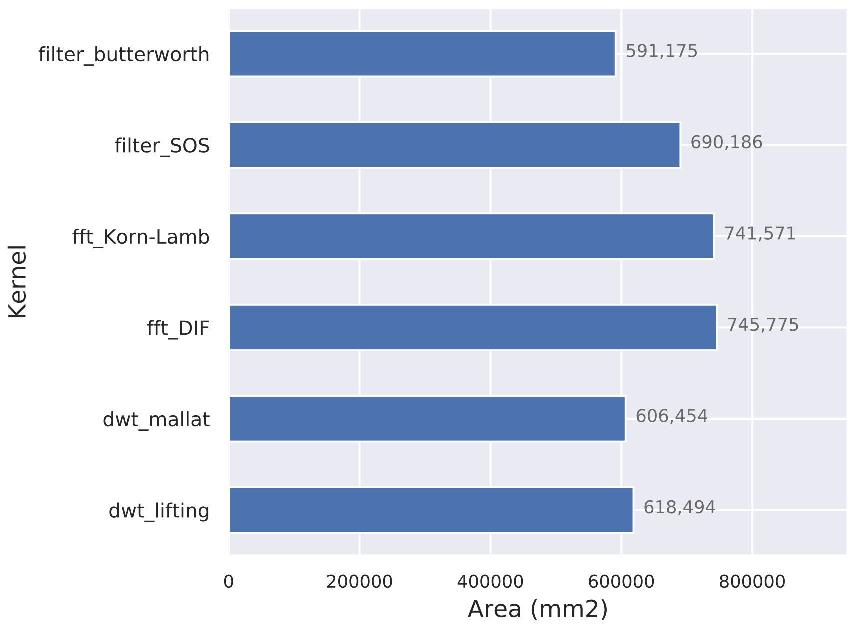


Figure 7.3: Area comparison.

Chapter 8

Blocks instance sizing and shortcomings

This chapter proposes the size of the *Blocks* CGRA for computing the FFT, LWT and the cascaded SOS filter. The chapter ends by presenting possible optimizations for the EEG platform used in this thesis.

8.1 Blocks sizing

In table 8.1 both the initial and the proposed *Blocks* template are shown. In the initial template, the number of IDs are marked with an 'X' indicating that they where still an open design parameter. It is possible to see that the proposed template is similar to the initial template, being the only differences the number of ID and RFs that were defined to 13 and 2 units respectively, This design choice implied an energy-area trade-off, meaning that achieving the lowest energy consumption per channel and the smallest set of FUs that can compute most of the kernels were taken into account.

Looking at table 7.4, it is possible to see that the number of IDs were determined by the FFT mappings. Regarding the RFs, the selection was done in order to fit the LWT. An alternative option would be to support the FWT, which requires only one RF and an extra ALU and IMM compared to the initial template. However, the LWT proved to be more efficient, consuming 1/3 of the energy per channel when compared to the FWT.

For the Butterworth filter, the use of the 10^{th} order IIR was not considered since it is the only kernel that requires four RFs, thus adding them to the template would negatively impact the area and leakage energy for the rest of the kernels.

Table 8.1: Blocks proposed sizing

	IDs	LSUs	ALUs	Muls	ABU	IMM	RF
Initial template	X	4	8	4	1	2	1
Proposed template	**13**	4	8	4	1	2	**2**

8.2 Blocks shortcomings

The architectural shortcomings of the EEG processing platform that were identified during the realization of this thesis are briefly explained next.

- **Sign-extension of loaded values:** This shortcoming involves the LSU in the *Blocks* CGRA and was already shown in figure 6.9. To illustrate this, we assume a 32-bit data-path and a 16-bit sample x with a negative value stored in a LSU local memory. Suppose that a multiplication of the sample x by another operand is required.

  If the sample x has a positive value, it would be possible to perform the multiplication straight after x is loaded. However, if the sample x has a negative value, it would be necessary to first use an ALU to sign-extend it and then perform the multiplication. This may sound trivial since it can be worked around easily using an ALU or a different algorithm. However, the first option is undesired from the energy-area point of view since it would involve the addition of an ALU just for sign-extension in a compute pipeline and the second option would limit the range of algorithms that can run on the platform. A possible improvement is to add a way of loading and sign extending the input data using a single *Blocks* instruction.

- **Global memory access cost:** Currently, even non-conflicting memory operations cost 3 cycles, i.e. 1 cycle to execute the instruction and 2 cycles caused by the interface used to access the global memory. This means that $1,024$ stall cycles are added for reading/writing a 256-sample signal which represents around 16.25% of the total cycle count of kernels such as the *Korn-Lambiotte* FFT and the LWT.

- **Shifting support:** As of now, the *Blocks* ISA supports shifting operations by 1 and 4 bits in the ALU and by 8, 16 and 24 bits by means of a multiplication+shift operation in a multiplier[1]. It was observed that, since in fix-point designs it is usually required to scale the data or the filter coefficients to achieve accurate results, better shifting support is needed in order to scale back the output without stalling or increasing the latency of the pipeline. An example of this situation was found in the cascaded SOS filter, where the output needs to be shifted right by 8-bits, figure 6.24a shows the actual implementation. In this case it was not so critical since only 2x shift-4 operations were required. However, it could be a problem when shifting by other than a multiple of four, i.e.shift-7 or when shifting by a large value, i.e. shift-16. In the later case, it could be possible to use the multiplier to perform the shift in a single cycle but it would be undesired from the energy efficiency point of view.

From the shortcomings presented in this section, the global memory access cost is considered to be the most critical since it becomes more significant every time a mapping improve its cycle count, which is one of the parameters targeted for optimization.

[1]Example, supposing that a number x needs to be shifted right by 16 bits, it could be possible to use the available instruction 'mulu_sh16' that performs a multiplication and shift the result by 16 bits in the following way: $(x * 1) >> 16$.

Chapter 9

Energy Model

In this chapter, three approaches taken in the literature that propose a methodology to obtain an energy model are presented. Although many more works that try to predict the energy consumption exits in the literature, they mostly fall within two categories: Measurement-based and Simulation-based which are exemplified by the works presented next. The chapter concludes by arguing why the construction of an energy model was not feasible for this thesis.

9.1 Example of energy models in the literature

The approach taken in [2] starts by explaining the classification of models for energy estimation. In the simulation-based energy models, a model of the hardware is used to run the applications in order to calculate the energy consumption for each component of the system, having as a down side the unavailability and expensiveness of such models. In the measurement-based energy models, the data obtained from measurements in a physical device is used. These models associate the instructions with the corresponding energy cost, having as main advantage a high accuracy in the energy estimation. The proposed method is measurement-based. A 1-ohm resistor at the power supply of the microcontroller was placed to measure the current. The total energy was calculated as the sum of the energy consumed by fetching, decoding and executing an instruction. The energy for executing an instruction was further subdivided into the energy consumed for the memory accesses, the processing energy and the static energy.

In their model, they take into account the *Instruction base energy*, which is the cost for executing the current instruction, and the *inter-instruction* cost which occur when two different instructions are executed sequentially. They show how the complexity of developing an energy model increases due to the fact that the base energy depends on the current instruction type, mode and operands. And similarly the inter-instruction cost depends on the current and previous instruction and their operands hamming distance. They cope with this complexity by ignoring inter-instruction cost estimations. A linear regression analysis was used to compute the coefficients of the model (fitting). The equations required for the regression analysis were obtained by measuring the energy consumption of 60 special programs prepared in such a way that each of them magnifies the effect of a specific model parameter.

Another example of measurement-based approach is shown in [22], here the energy of a sequence of instructions was computed by analysing a set of test programs. To construct the energy model they fist assume a model equation (linear combination of the factors) that can possibly affect the energy behaviour of the instructions. Then the model parameters are derived using a black box approach (test programs) and measuring the energy consumption of the hardware. Similar to [2], a linear regression analysis was used to fit the model to the actual energy behaviour.

The common factor of [2] and [22] is that both focus on microcontrollers/microprocessors. An attempt to model the energy consumption in a VLIW architecture is done in [31], where an instruction-level energy model is proposed for the data-path of VLIW pipelined processors. This work is similar to [2] and [22] in the fact that a regression analysis is used to find the final model. This model takes into account several software level parameters, such as instruction ordering, pipeline stall probability, and instruction cache miss probability. The authors have explained the problem of how the complexity in instruction-level power characterization of a k-issue VLIW processor grows to $O(N^{2k})$, where N is the number of operations in the ISA and k is the number of parallel instructions composing the instruction. This is because contrary to the microprocessor energy model, models for VLIW architectures should account for all the possible combinations of operations in an instruction. To cope with this, they use a cluster approach in which they group instructions with similar power cost in order to reduce model parameters. The proposed model is simulation-based and require a gate level description of the target processor to derive the energy values and a suitable Instruction-Set Simulator (ISS) to record the information needed for the model. In order to determine the model parameters, 250 experiments were automatically generated varying the number of registers, the values in the IMMs and the number and type of operations.

9.2 The energy model construction problem

This section explains the reasons for which an energy model was not proposed in this thesis. In section 9.1 three examples of simulation-based and measurement-based models were presented. Regardless of the method to obtain the model parameters, they both use a regression analysis to fit the model. Theoretically, it would be possible to use either the simulation-based or measurement-based method to obtain an energy model for the *Blocks* CGRA. The requirements are an architecture description or the physical device for the simulation-based or the measurement-based methods respectively. As seen in the works presented in section 9.1, the amount of variables that influence the energy consumption is large, and the use of automatic program generation was required. The difficulty of proposing and energy model for the *Blocks* CGRA lays in the derivation of the model parameters due to all the possible configurations, instructions, operands, immediate values and the lack of a compiler to generate test programs.

Therefore, an accurate model at the instruction level is infeasible for the time span of this thesis taking into account that all the scheduling of the test programs has to be done manually. It could be possible to make an energy model based on a currently available mapping, however there are two drawbacks of this approach, the first is that it would be inaccurate for the energy consumption estimation of a different mapping, and the second is that the model would need to be recomputed after any modifications to the hardware. For these reasons an energy model for the *Blocks* platform was not done in this thesis.

Chapter 10

Conclusions

In this thesis, a common reconfigurable architecture for the computation of a set of EEG features with a focus on energy efficiency was defined. To do this, an initial benchmark of an EEG application running on a *RISC-V* was performed in order to identify bottlenecks. The EEG application is composed by several features from which the *Power per Band* (FFT), the DWT and the Butterworth filter were selected, analyzed and mapped to the *Blocks* CGRA.

The netlist simulation results show that, on top of the efficiency obtained by the parallel computation in the *Blocks* CGRA, an improvement in the cycle count and energy consumption can be achieved when a suitable algorithm is used for the computation of the features.

The *Korn-Lambiotte* algorithm was used to implement the FFT in the *Power per Band* feature, obtaining a reduction in the energy consumption of 2.22x compared to the reference *RISC-V* reference.

The lifting scheme (LWT) was used to implement the DWT obtaining a reduction in the energy consumption of 3.25x for six levels of wavelet decomposition with respect to the *RISC-V* reference. Since minor modifications to the current LWT mapping would enable to perform the wavelet reconstruction, a potential speed-up of 18.8x compared to the *RISC-V* reference is expected when both wavelet decomposition and reconstruction are computed.

For the Butterworth filter, a cascaded SOS filter was implemented, showing a reduction in energy consumption of 1.7x with respect to the reference running in the *RISC-V* core. This is low when compared to the existing 10^{th} order Butterworth mapping in the *Blocks* CGRA that achieved a better energy improvement of 2.75x compared to the *RISC-V* reference.

For each of the selected algorithms, a literature research of efficient implementations was done in order to propose alternatives that further reduce the cycle count and energy efficiency of the EEG platform.

A global comparison of the mapped features was presented, including the energy consumption involved in the computation of the kernel and the energy spend for copying the program into the *Blocks* instruction memory. The results show that loading the program incurs in a large energy consumption compared to the energy required for the actual computation. However, this is not critical as the same program can be reused when computing several EEG channels.

Initially, the proposition of an energy model was set as one of the contributions of this thesis. However, the many factors that influence the power consumption in a CGRA architecture, the lack of a compiler, the flexibility of the *Blocks* architecture and the limited time span of this thesis are the reasons for which an energy model for the *Blocks* CGRA is not proposed.

With respect to the sizing of the *Blocks* template, 13 IDs are required, and only an extra RF was added when compared to the initial template.

Future work on this thesis includes the implementation of the suggested optimizations, investigate the use of a different interface to interact with the global memory and the extension of the ISA in the *Blocks* CGRA that includes arbitrary shifting in the ALUs and a 'load+sign-extend' operation in the LSUs.

Bibliography

[1] Muhammad Awais Bin Altaf, Chen Zhang, Ljubomir Radakovic, and Jerald Yoo. Design of energy-efficient on-chip eeg classification and recording processors for wearable environments. In *Circuits and Systems (ISCAS), 2016 IEEE International Symposium on*, pages 1126–1129. IEEE, 2016. 1, 4, 5

[2] Mostafa Bazzaz, Mohammad Salehi, and Alireza Ejlali. An Accurate Instruction-Level Energy Estimation Model and Tool for Embedded Systems. *IEEE Transactions on Instrumentation and Measurement*, 62(7):1927–1934, jul 2013. 46, 47

[3] Bingfeng Mei, Andy Lambrechts, Diederik Verkest, J. Mignolet, and Rudy Lauwereins. Architecture Exploration for a Reconfigurable Architecture Template. *IEEE Design and Test of Computers*, 22(2):90–101, feb 2005. 6

[4] Yifan Bo, Renfeng Dou, Jun Han, and Xiaoyang Zeng. A hardware-efficient variable-length FFT processor for low-power applications. In *2013 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference*, pages 1–4. IEEE, oct 2013. 22

[5] S. Alexander Chin, Noriaki Sakamoto, Allan Rui, Jim Zhao, Jin Hee Kim, Yuko Hara-Azumi, and Jason Anderson. CGRA-ME: A unified framework for CGRA modelling and exploration. In *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 184–189. IEEE, jul 2017. 6, 7

[6] W.T. Cochran, J.W. Cooley, D.L. Favin, H.D. Helms, R.A. Kaenel, W.W. Lang, G.C. Maling, D.E. Nelson, C.M. Rader, and P.D. Welch. What is the fast Fourier transform? *Proceedings of the IEEE*, 55(10):1664–1674, 1967. 13

[7] Ingrid Daubechies and Wim Sweldens. Factoring wavelet transforms into lifting steps. *The Journal of Fourier Analysis and Applications*, 4(3):247–269, may 1998. 25, 26, 28

[8] Chunxiao Fan, Fu Li, Xin Cao, Biao Qian, and Peipei Song. A parallel arithmetic for hardware realization of digital filters. *Microelectronics Journal*, 83(December 2018):131–136, jan 2019. 38

[9] Franz Franchetti, Markus Puschel, Yevgen Voronenko, Srinivas Chellappa, and Jose Moura. Discrete fourier transform on multicore. *IEEE Signal Processing Magazine*, 26(6):90–102, nov 2009. 18

[10] Md Mehedi Hasan and Khan A. Wahid. Low-Cost Lifting Architecture and Lossless Implementation of Daubechies-8 Wavelets. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 65(8):2515–2523, aug 2018. 31

[11] Cognionics quick-30 Headset. Availabe at: https://www.cognionics.net/quick-30. *Accessed on: 18th July*, 2018. 1

[12] Emotiv Epoch Headset. Availabe at: http://www. emotiv. com. *Accessed on: 12th July*, 2018. 1

[13] InteraXon Muse Headset. Availabe at: https://eu-store.choosemuse.com/products/muse. *Accessed on: 18th July*, 2018. 1

[14] Neuroelectrics Enobio 8/20/32 Headset. Availabe at: https://www.neuroelectrics.com. *Accessed on: 18th July*, 2018. 1

[15] Neurosky MindWave Headset. Availabe at: https://store.neurosky.com/pages/mindwave. *Accessed on: 18th July*, 2018. 1

[16] Quasar DSI Headset. Availabe at: http://www.quasarusa.com. *Accessed on: 18th July*, 2018. 1

[17] Carl Ingemarsson, Petter Kallstrom, Fahad Qureshi, and Oscar Gustafsson. Efficient FPGA Mapping of Pipeline SDF FFT Cores. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(9):2486–2497, sep 2017. 23

[18] Arne Jensen and Anders la Cour-Harbo. *Ripples in mathematics: the discrete wavelet transform*. Springer Science & Business Media, 2001. 24, 26

[19] Steven G Johnson and Matteo Frigo. Implementing FFTs in practice. *Fast Fourier Transforms*, pages 1–23, 2008. 14

[20] Bojan Kerous, Filip Skola, and Fotis Liarokapis. Eeg-based bci and video games: a progress report. *Virtual Reality*, 22(2):119–135, 2018. 1

[21] Joyce Kwong and Anantha P Chandrakasan. An energy-efficient biomedical signal processing platform. *IEEE Journal of Solid-State Circuits*, 46(7):1742–1753, 2011. 4, 5

[22] Sheayun Lee, Andreas Ermedahl, and Sang Lyul Min. An Accurate Instruction-Level Energy Consumption Model for Embedded RISC Processors. In *Proceedings of the 2001 ACM SIGPLAN workshop on Optimization of middleware and distributed systems - OM '01*, pages 1–10, New York, New York, USA, 2001. ACM Press. 46, 47

[23] Vojkan Mihajlović, Bernard Grundlehner, Ruud Vullers, and Julien Penders. Wearable, wireless eeg solutions in daily life applications: what are we missing? *IEEE journal of biomedical and health informatics*, 19(1):6–21, 2015. 1

[24] Fabio Montagna, Simone Benatti, and Davide Rossi. Flexible, scalable and energy efficient bio-signals processing on the pulp platform: A case study on seizure detection. *Journal of Low Power Electronics and Applications*, 7(2), 2017. 4

[25] Soumak Mookherjee, Linda DeBrunner, and Victor DeBrunner. A low power radix-2 FFT accelerator for FPGA. In *2015 49th Asilomar Conference on Signals, Systems and Computers*, pages 447–451. IEEE, nov 2015. 22

[26] Edras Reily Pacola, Veronica Isabela Quandt, Paulo Breno Noronha Liberalesso, Sergio Francisco Pichorim, Humberto Remigio Gamba, and Miguel Antonio Sovierzoski. Influences of the signal border extension in the discrete wavelet transform in EEG spike detection. *Research on Biomedical Engineering*, 32(3):253–262, sep 2016. 31

[27] Thomas Peyret, Gwenole Corre, Mathieu Thevenin, Kevin Martin, and Philippe Coussy. Efficient application mapping on CGRAs based on backward simultaneous scheduling/binding and dynamic graph transformations. In *2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors*, pages 169–172. IEEE, jun 2014. 7

[28] Senthilkumar Ranganathan, Ravikumar Krishnan, and H S Sriharsha. Efficient hardware implementation of scalable FFT using configurable Radix-4/2. In *2014 2nd International Conference on Devices, Circuits and Systems (ICDCS)*, number 5, pages 1–5. IEEE, mar 2014. 22

[29] K. Deergha Rao and M.N.S. Swamy. *Digital Signal Processing.* Springer Singapore, Singapore, mar 2018. 33

[30] Nigel Redmon. Biquads, Practical digital audio signal processing, 2003. 38

[31] Mariagiovanna Sami, Donatella Sciuto, Cristina Silvano, and Vittorio Zaccaria. An instruction-level energy model for embedded VLIW architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(9):998–1010, sep 2002. 46

[32] Dietrich Schlichthärle. *Digital Filters.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. 31, 32

[33] B.R. Sekhar and K.M.M. Prabhu. Radix-2 decimation-in-frequency algorithm for the computation of the real-valued FFT. *IEEE Transactions on Signal Processing*, 47(4):1181–1184, apr 1999. 23

[34] Guangming Shi, Weifeng Liu, Liu Zhang, and Fu Li. An efficient folded architecture for lifting-based discrete wavelet transform. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 56(4):290–294, 2009. 30

[35] H. Sorensen, D. Jones, M. Heideman, and C. Burrus. Real-valued fast Fourier transform algorithms. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 35(6):849–863, jun 1987. 23

[36] Srinivasa R Sridhara, Michael DiRenzo, Srinivas Lingam, Seok-Jun Lee, Raúl Blázquez, Jay Maxey, Samer Ghanem, Yu-Hung Lee, Rami Abdallah, Prashant Singh, et al. Microwatt embedded processor platform for medical system-on-chip applications. *IEEE JOURNAL OF SOLID-STATE CIRCUITS*, 46(4):1, 2011. 4, 5

[37] Morten Danmark Nielsen Stine Martine Gullaksen. Low-energy fir filter realisations on hardware and software programmable platforms. Master's thesis, Aalborg University Denmark, 2012. 37

[38] Dongkwan Suh, Kiseok Kwon, Sukjin Kim, Soojung Ryu, and Jeongwook Kim. Design space exploration and implementation of a high performance and low area Coarse Grained Reconfigurable Processor. In *2012 International Conference on Field-Programmable Technology*, number Mc, pages 67–70. IEEE, dec 2012. 5, 6

[39] D Sundararajan. *Discrete wavelet transform: a signal processing approach.* John Wiley & Sons, 2016. 24

[40] Paul N. Swarztrauber. FFT algorithms for vector computers. *Parallel Computing*, 1(1):45–63, aug 1984. 18

[41] L Wang, JBAM Arends, X Long, PJM Cluitmans, and JP van Dijk. Seizure pattern-specific epileptic epoch detection in patients with intellectual disability. *Biomedical Signal Processing and Control*, 35:38–49, 2017. 5, 9, 11

[42] Mark Wijtvliet and Luc Waeijen. Blocks architecture. [Online: http://cgra.nl; accessed 21-September-2018]. 10

[43] Wikipedia contributors. Electroencephalography — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Electroencephalography&oldid=847167391`, 2018. [Online; accessed 12-July-2018]. 1

[44] R. Yates. Practical considerations in fixed-point FIR filter implementations, 2000. 34

[45] Jerald Yoo, Long Yan, Dina El-Damak, Muhammad Awais Bin Altaf, Ali H Shoeb, and Anantha P Chandrakasan. An 8-channel scalable eeg acquisition soc with patient-specific seizure classification and recording processor. *IEEE journal of solid-state circuits*, 48(1):214–228, 2013. 5

[46] Jonghee W. Yoon, Aviral Shrivastava, Sanghyun Park, Minwook Ahn, Reiley Jeyapaul, and Yunheung Paek. SPKM : A novel graph drawing based algorithm for application mapping onto coarse-grained reconfigurable architectures. In *2008 Asia and South Pacific Design Automation Conference*, pages 776–782. IEEE, jan 2008. 6, 7

Appendix A

Analysis Polyphase Matrix

Analysis polyphase matrix for the LWT factorization.

$$P(1/z)^{'} = \begin{bmatrix} 0.7341 & 0 \\ 0 & 1.3621 \end{bmatrix} * \begin{bmatrix} 1 & 0 \\ -0.4691z^0 + 0.1401z^{-1} - 0.0248z^{-2} & 1 \end{bmatrix} * \begin{bmatrix} 1 & 2.1318z^0 + 0.6364z^{-1} \\ 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 \\ -0.0189z^2 + 0.1176z^1 & 1 \end{bmatrix} * \begin{bmatrix} 1 & -1.1171z^0 - 0.3001z^{-1} \\ 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 \\ -0.3222z^1 \end{bmatrix} \tag{A.1}$$